Technische Universität München
Lehrstuhl für Integrierte Systeme

Application-driven Development of Flexible Packet-oriented Communication Interfaces

Christian Sauer

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:

Univ.-Prof. Dr.-Ing. Ulf Schlichtmann

Prüfer der Dissertation:

1. Univ.-Prof. Dr. sc. techn. Andreas Herkersdorf
2. Prof. Kurt Keutzer, Ph.D. (University of California, Berkeley, USA)

Die Dissertation wurde am 2.6.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 13.7.2010 angenommen.

Contents

List of Figures

List of Tables

Acknowledgements

Without the support and advice of my colleagues and students this dissertation would not have been possible. Over the years, many of them have become friends. At UC Berkeley, I would like to thank Michael Shilman, Scott Weber, Andrew Mihal, Niraj Shah, Matt Moskewicz, Will Plishker, Kaushik Ravidran, N.R. Satish, Jose-Ignacio Gomez, Jörn Janneck, Chidamber Kulkarni, and Matthias Gries. At Infineon in Munich, I thank Matthias Gries (once more), Andreas Gräfe, Milena Dimitrova, Sören Sonntag, Sebastian Dirk, Hans-Peter Löb, Xiaoyan Jia, Winthir Brunnbauer, Chinapirom Teerapat, and Hans-Martin Blüthgen.

There have been collaborations that came to life through the people executing them. My thanks go to Jörg-Christian Niemann, Mario Pormann, Christian Liss, Ulrich Rückert (all at U Paderborn), Dietmar Tölle, Bo Wu, Helmut Steckenbiller, Rudi Knorr (at FhG Munich), Daniel Lorente, and Thomas Wild (at TU Munich).

Infineon Technologies consistently supported my research. I especially thank my superiors Ulrich Ramacher (Corporate Research) and Friedrich Geissler (Communication Solutions) for their enthusiasm and interest.

I thank my advisors Kurt Keutzer at UC Berkeley, and also Andreas Herkersdorf at TU Munich, for sharing their experience and wisdom.

But most of all, I would like to thank my wife Katja for standing by me with her unending patience and love.

Chapter 1

Introduction

Rapid progress in embedded systems such as network-connected information appliances has given new impetus to a wide and rapidly growing range of input-output (I/O) communication standards such as PCI Express, USB, and IEEE 802.11. This in turn has resulted in system architectures combining customized blocks for communication interfaces with programmable processor cores. Processors, memory hierarchies, interconnection networks, communication interfaces and other peripherals are the main building blocks for these application-specific programmable system-on-a-chip architectures.

During the design process a great deal of emphasis is placed on the first three building block types but communication interfaces and other peripherals are often neglected. This is mainly due to the general perception that these modules constitute standard intellectual property (IP) blocks that can be integrated quickly at design time.

However, due to their heterogeneity and ad-hoc integration practice these modules are increasingly becoming bottlenecks to both the design process and the complexity of the end system as will be discussed below.

1.1 The reality of the interface zoo

We define *communication interfaces* as system-on-chip building blocks which are connected to the pins of the chip. Auxiliary functions that primarily support the interaction between the communication interface and the core system such as a direct memory access (DMA) controllers are also included.

A large amount of effort is put in processors, memories, and on-chip interconnect during the SoC design process whereas communication interfaces are often neglected and their complexity underestimated. They are perceived as standard intellectual property to be acquired as needed. These blocks are therefore integrated into the system-on-chip separately and relatively late during the design process.

Contrary to common perception, communication interfaces play a significant role in contemporary SoCs as illustrated by two representative systems.

The first, Intel's IXP1200 network processor, is shown in Figure 1.1. The concurrent and heterogeneous multiprocessor system is aimed at processing packets in network nodes at wire speed. Its die photograph (taken from [60]) identifies the individual building blocks. The system comprises six RISC-like processing cores (Microengines) that perform the majority of the network processing tasks, a supervising processor for control and maintenance tasks (StrongARM), and a set of dedicated communication interfaces for different purposes (SRAM, SDRAM, PCI, and IX-Bus). All these building blocks are connected by a diverse on-chip network consisting of multiple buses and point-to-point connections.

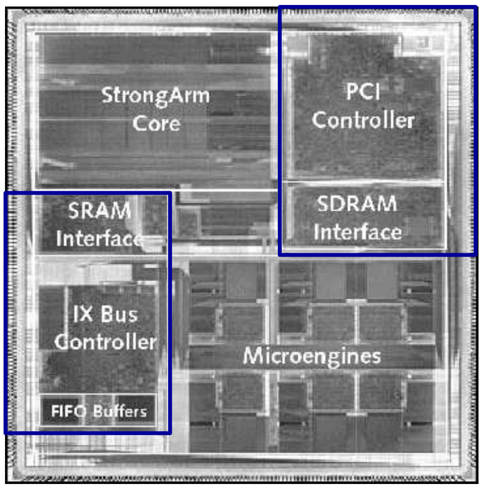


Figure 1.1: IXP1200 die photograph [60].

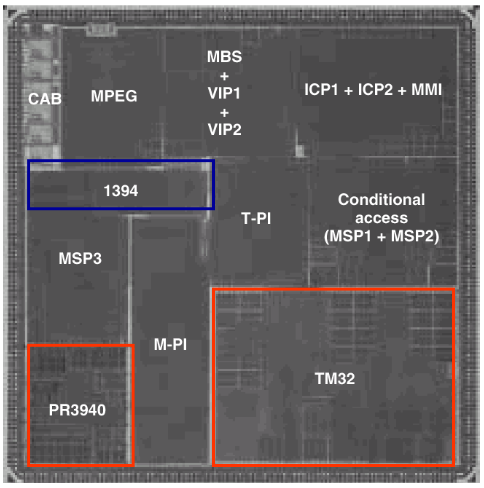


Figure 1.2: Nexperia die photograph [44].

The other example is the Philips Viper system (Fig. 1.2) that targets digital set-top boxes and residential gateways. The system contains a MIPS processing core (PR9340) for standard tasks and the Trimedia VLIW (TM32) core for audio and video processing. The VLIW core is supported by several coprocessors for 2D rendering, MPEG2 encode/decode, and video processing. In the die

photograph the Firewire interface (IEEE1394) is identifiable. Other peripherals are grouped together into larger structures. According to [44], the Viper system implements a total of 50 peripherals and accelerators in 82 different clock domains.
From these two examples it is already evident that:

- Interfaces account for a significant share of the total system, which in these examples amounts to roughly 1/3rd to 2/3rd of the die areas.
- There are a number of different communication interfaces integrated separately on one SoC.
- Individual interfaces such as the PCI interface in Figure 1.1 or the Firewire interface (IEEE 1394) in Figure 1.2 can be quite large compared to other structures on the die, especially the microengines.

Two trends in evolving communication protocols aggravate the listed observations: (1) the complexity of interfaces is increasing, and (2) the protocol landscape is becoming more diverse.

Due to the increasing bandwidth and more sophisticated communication protocols, the *complexity* of communication interfaces increases over time. In Figure 1.3, the interface sizes are shown for different PCI protocol versions (adapted from [28]). The most recent PCI Express interface is 3.6 times larger than the version that was used for the Intel network processor in Figure 1.1 while its performance in terms of bandwidth/pin increases by 64X (cf. Fig 1.4).

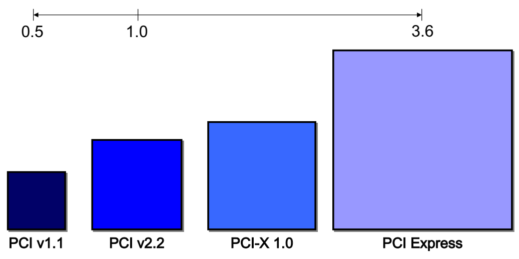


Figure 1.3: Relative increase in interface area for different PCI protocol standards (Scaled to identical technology).

At the same time, the landscape of communication protocols is becoming more *diverse*. Recently announced new and existing enhancements to several packet-oriented link-to-link communication protocols, for instance, are all aimed at network processors. It is, however, unclear, which of these protocols should be supported. Not only do these standards represent different market alliances, but they also provide (or will provide as they evolve) comparable features at a similar performance/cost ratio, as Figure 1.4 illustrates. The figure uses the pin count for cost indication since communication centric systems tend to be dominated by IO pins.

Furthermore, the software interface for such communication interfaces, which comprises device drivers and often the related protocol management, is becom-

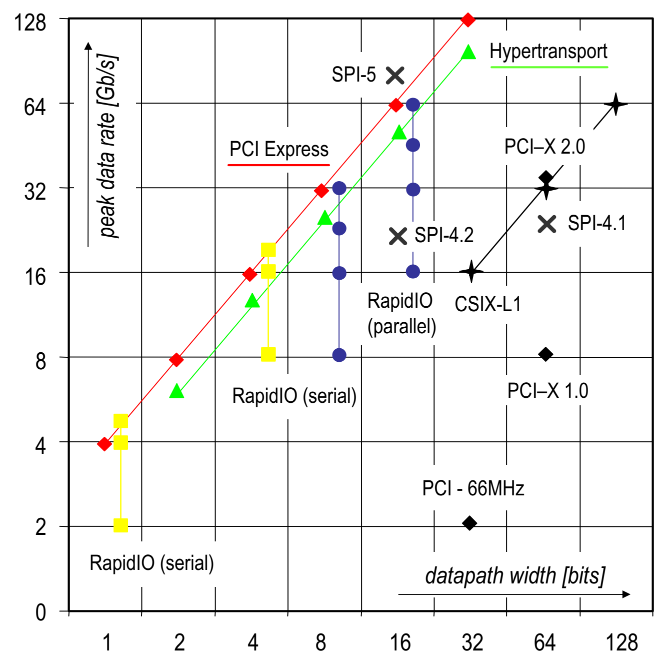


Figure 1.4: Bandwidth vs. pin count of recent network processor interfaces.

ing complex and more heterogeneous, resulting in a challenge for designing reliable software [185]. Finally, due to an unfavorable task partitioning, the interaction between device driver, processor and communication interface is often very fine-grained and dominated by the protocol overhead [170]. Due to the dynamic nature of such interactions, the overall system behavior is difficult to predict, therefore, making a over-provisioning of the involved resources necessary.

These observations reveal the necessity of a disciplined approach to the design and integration of communication interfaces. Interfaces must become a *first class citizen* during the design process of SoCs. The premise from which our research is conducted is that a flexible solution can reduce diversity while still delivering reasonable performance, consequently alleviate the communication interface problem. Using a flexible interface module is the first key concept of our work. The second concept is the recursive deployment of a programmable platform for providing the desired flexibility. Both are discussed next.

1.2 The promise of flexible interfaces

The first key idea for the reduction of the diversity of communication interfaces is the concept of a *flexible interface module* that can be targeted at multiple IO communication standards by configuration. Please note that the specification of a particular interface function is here called *configuration* without implying a particular implementation style (i.e., reconfigurable logic).

Such a flexible architecture for a set of communication protocols would help solve the problems described in the previous section in the following ways:

Reduced heterogeneity and diversity — If the number of required architectures is small enough, the problem of diverse and heterogeneous interface modules in hardware will be alleviated. Instead of numerous different modules, the SoC will now contain multiple instances of only a few modules, which are *configured* differently to provide the desired functionality. Thus, the SoC becomes more regular and the design process easier.

The number of architectures required for capturing the interface variety will be determined by a trade-off between efficiency of an implementation, for instance in terms of performance or energy consumption, and flexibility. There are two obvious extremes. The minimal set of architectures is a set which contains just one class that can be configured to provide any function – but most likely not very efficiently. The other extreme is the maximal set which contains a single architecture per interface – thus optimized and most efficient.

Improved area utilization and reuse — The *reconfigurability* of modules actually allows reusing the silicon area. This, depending on the implementation costs for the flexible architectures, may lead to substantial area reductions since the number of instantiated modules can be reduced to the number of simultaneously active modules. Additionally, the reconfiguration of system interfaces also makes possible reusing the complete system in environments with different communication requirements as well as the support for fast-evolving standards as, for instance, observable in the network processing arena.

Easier system integration — The architectural complexity necessary to provide flexibility allows for a repartitioning of the functionality between main processor(s) and IO module. By allocating more low-level communication tasks to the IO resource itself, the main system can be relieved of fine-grain communication, and the communication protocol processing can be kept on the local processing resource. This makes very thin device driver layers possible, which only deal with the operating system specifics and the general processor-device interface. In addition, low-level device drivers will become less diverse since all IO protocols within an interface class can be accessed via the same HW/SW interface.

An established way for providing flexibility is by means of a reconfigurable fabric, as, e.g., seen in [67]. Different I/O interfaces can be synthesized from HDL descriptions and loaded as configuration into the fabric using mature tool chains. In fact, companies such as Altera and Xilinx offer communication interfaces as IP modules for their FPGA platform products.

However, the flexibility of FPGA platforms comes at the price of several weaknesses. First, there are higher implementation costs. In [56] factors of 2-4X in speed decrease and 10X or more in area increase compared to application-specific standard parts are mentioned. Second, when compared to embedded processors, they are often less efficient. Although their computational density is higher (in terms of peak computations per area), their program, i.e., fabric configuration, is much less dense than the one of a processor [41]. Consequently, processors can solve a problem that requires many relatively infrequently used computations or moderate computational throughput in a smaller area than

FPGAs. For this reason companies such as Triscend [61] (acquired by Xilinx) and ST Microelectronics [13] embed FPGA fabrics into their programmable SoCs. FPGA vendors, on the other hand, harden their products by embedding processor cores into the fabric. Third, there is the lack of an integral and productive programming environment. A flexible system-on-chip which combines hardware on an FPGA fabric with programmable cores (be it as hard or soft macro) requires multiple tool flows and disjoint programming models for hardware synthesis and program compilation. Hence, can we do better, e.g., if we leverage the building blocks of programmable platforms?

1.3 Leveraging the programmable platform

Motivated by the need to provide efficient and economical solutions for systems of growing heterogeneity and complexity, platforms have emerged that are built upon software programmable building blocks, i.e. processing elements. Such programmable platforms promise higher design productivity since they are built more abstractly from coarse-grain processing elements rather than designed on the fine-grained RT level. By software programmability, they also make late or in-field adaptations of a system's function possible, thus reducing the first-time-right risk and increasing the longevity of systems in evolving markets.

Contemporary platforms are multiprocessor systems. In the domain of network processors, for instance, dozens of different platforms have been created [177, 205]. They instantiate multiple processing elements in increasing numbers ranging from two to several hundred. This trend is illustrated in Figure 1.5 which shows the number of processing elements used by selected network processors and their topology.

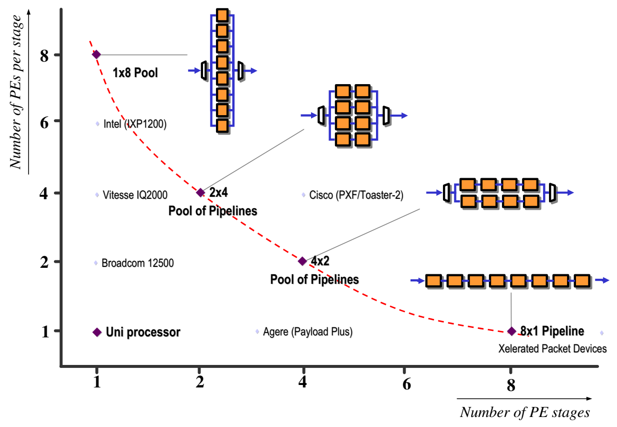


Figure 1.5: Processing core topologies in MPSoC platforms.

This leads to the second key idea of this dissertation. Since contemporary platforms deploy many processing elements for flexibility anyway, it would make sense to use them for a flexible interface solution as well. Figure 1.6 shows the concept. Instead of dedicated interface macros in the left part of the figure, processing elements, shown in the right-hand part, are deployed which execute the particular interface function in software. In a recursive manner, the IO building block is composed of other building blocks. Only a fraction of the original module concerned with the physical IO (the PHY) may remain as IO specific unit.

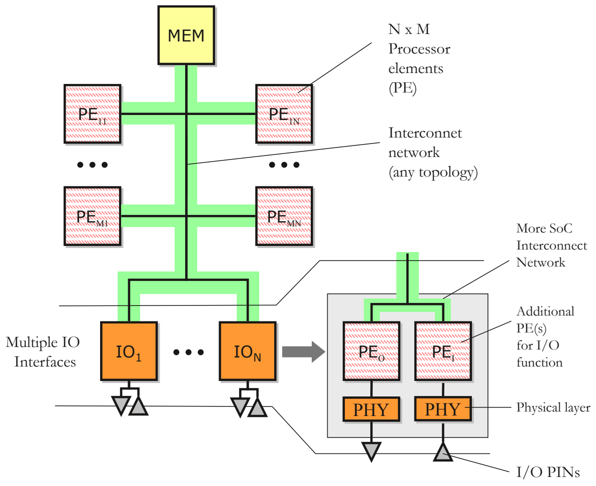

MEM
N x M Processor elements (PE)
PE_{11}
PE_{1N}
Interconnet network (any topology)
PE_{M1}
PE_{MN}
More SoC Interconnect Network
Additional PE(s) for I/O function
Multiple IO Interfaces
IO_1
IO_N
PE_0
PE_i
PHY
PHY
Physical layer
I/O PINs

Figure 1.6: SoC platform deploying programmable platform building blocks (PEs) to replace its hardwired IO functions.

Compared to a reconfigurable fabric, such a solution has two main advantages. First, the *integral* programming model of the platform can be extended to the interface functions. Identical tools and the same programming environment can be used. Second, the platform becomes more homogenous and better scalable since the interface function could be mapped onto any PE.

However, to date, no established method for the systematic exploration of a platform's design space exists. Platform development and deployment still remain an art.

1.4 Objectives of research

The objective of this research is to contribute to the advent of fully programmable platforms by exploring the feasibility of a programmable solution for communication interfaces. Being among the first pursuits specifically addressing communication interfaces this dissertation's central assumption is that there is a set of communication interfaces that can be implemented by such a solution. To prove this hypothesis, this dissertation proposes:

1. A methodology and a set of domain-specific tools for the application-driven development and later deployment of programmable communication interfaces. The *SystemClick* performance evaluation framework generates functional and timed SystemC models from high-level descriptions that can be used to evaluate a particular design point quickly. From the same source, the *CRACC* framework generates efficient implementations for embedded software platforms.

2. The quantitative exploration of the interface design space based on a modular platform that is optimized for packet processing. Using the *NOVA*

platform, performance/area trade-offs are quantified with respect to processor type and configuration, application-specific instruction sets, hardware accelerators, number of processors, and communication topologies.

Adopting a depth-first approach, the focus is placed on the emerging class of packet-oriented communication interfaces. They are essential for networked devices (see Chapter 2) such as wireless access points and network processors. The requirements of these devices will be used throughout this dissertation.

This dissertation will be successful if a flexible solution is found that is a) programmable using an integral programming model, and b) applicable with reasonable costs in terms of performance and area.

1.5 Structure of this dissertation

Three steps are necessary to confirm the feasibility of a programmable interface solution. First, an essential set of communication interfaces must be selected and analyzed. Second, a methodology for the systematic exploration of the design space is required. Last, a programable architecture must be developed and evaluated. According to these steps, this dissertation is structured as follows:

Survey of communication interfaces in contemporary SoCs — In order to capture the state-of-the-art, a representative set of contemporary SoC architectures in different application domains is analyzed and classified in its interface specifics. Existing techniques towards a flexible implementation are discussed. The set of essential packet-oriented communication interfaces is derived, which is then used for the subsequent steps.

Interface analysis and comparison — The interfaces are analyzed and their similarities/dissimilarities are discussed using a common structure and a set of elementary tasks. For this purpose, each interface is modeled using Click, a framework for packet processing applications. The executable models are functionally correct and capture the essentials of each interface.

Application-driven methodology — For the design space exploration, a methodology is developed taking into account the importance of the application domain. The Y-chart-based methodology leverages the Click framework and uses our code generators to derive performance models (SystemClick) and the actual implementation code (CRACC). As the target for the exploration, the modular NOVA platform is introduced.

Architecture exploration — Following our application-driven methodology, we first look at a fully programmable single-PE solution. This solution is one corner case of an implementation and provides the necessary data for the exploration of the design space along several axis. Analyzing the performance, we quantitatively explore the design space with respect to number and type of cores, instruction set extensions, application-specific hardware accelerators, and communication topology. This results in an optimized architecture.

Conclusion — The conclusion summarizes the main results and contributions of this dissertation and provides some starting points for future work.

Chapter 2

Review of Communication Interfaces in SoCs

This chapter captures the state-of-the-art in the deployment of communication interfaces. In order for them to be representative of today's application-specific System-on-Chip architectures (SoC), we are examining more than 30 contemporary architectures in fourteen product families and discussing their interface specifics. The SoC examples come from three application domains with a focus on communication-intense network and networked multimedia applications.

In the following, we will first characterize the selected application domains. Next, we will examine the architectures in each domain applying a usage-based classification scheme to their interfaces. Last, we will compile the data on interface deployment into a set of 35+ essential communication interfaces. Furthermore we will evaluate the implementations of interfaces to identify techniques that support flexibility and reuse and place the focus of our subsequent research on packet-oriented communication interfaces for which we will select a collection of communication standards.

2.1 Application domains

Architectures from three domains were chosen to reveal the different aspects of communication interfaces:

Core and access networking — This domain consists of network processors that target packet processing and forwarding at high throughput rates in the network backbone and access equipment. Their architectures often contain multiple specialized processing elements (application-specific instruction processors - ASIPs) to meet the performance requirements. They require only a limited number of different interfaces.

Integrated access devices; Residential gateways — Residential gateways (aka access points) link the external broadband network and the internal home network and also serve as in-house multimedia aggregation point. They therefore have to deal with a plethora of different network protocols and communication interfaces. Due to moderate throughput requirements their architectures, however, are based on a limited number of standard processors. Integrated access devices are similar in their base architecture and incorporate only a limited number of network interfaces. In addition, they integrate functions for user interaction.

Handheld devices — Wireless application processors handle the application part of cellular phones and hand-held devices. This part often includes everything (e.g. audio/video, security or storage) except the baseband processing (physical layer) of the device. For wireless devices, low power consumption is of the essence.

2.2 Usage-based classification

We classify the communication interfaces based on their usage and group them according to their purpose. Although fairly natural and intuitive, such a classification is ambiguous in some cases since interfaces can be multi-functional or be used for different purposes, see infrared and Bluetooth. The classification scheme uses nine classes. The majority of the interfaces fall into seven of these classes, two additional classes cover the remaining communication interfaces:

- **Wired network interfaces**— This class contains all peripherals that interface to an external network based on wires and fibers; this includes interfaces for home networking, last/first mile connections, and local (LAN) and wide area network (WAN) interfaces.
- **Wireless network interfaces**— These peripherals interface to a network over the air via different communication techniques, which often require a substantially more complex physical (PHY) layer than wired interfaces. Typical interfaces are HomeRF, HIPERLAN, and wireless LAN. Infrared (IrDA) and Bluetooth - if used in combination with LAN protocol stacks - are also wireless interfaces.
- **Local peripheral interfaces**— This class includes communication interfaces that are primarily used to connect 'standard' peripherals such as printers, scanners, or cameras to a system. Typical interfaces are USB, Firewire, IrDA, and Bluetooth.
- **User interfaces**— Specialized interfaces for human user interaction such as keypads, LCDs, or cameras are accounted here.
- **Memory interfaces**— This class includes memory controllers and interfaces to external memories, e.g. SRAM, SDRAM, compact flash, or multimedia memory cards.
- **System extension interfaces**— This class contains interfaces that extend a system locally such as external buses or co-processor interfaces. Such interfaces are, for instance, PCI, RapidIO, Hypertransport, and Infiniband.
- **Test and debug interfaces**— This class covers interfaces that are specialized in test and debug purposes such as JTAG. General purpose interfaces may be used as well, but are accounted for in a separate class.

- **Common standard interfaces**— Some interfaces are integrated in virtually every SoC system but are not system-dependent. They also serve multiple purposes. So far, we have identified General Purpose IOs (GPIOs) and UARTs as class members.

- **Specialized auxiliary interfaces**— If an interface does not fit in any of the other classes, it is considered in this class, e.g., the synchronous serial interfaces.

In the following, we will examine the groups of interfaces for every one of our application classes and discuss their specifics.

2.3 Interfaces deployed with network processors

Network processors (NPUs) target packet processing and forwarding at high throughput rates in the network core and in access equipment. In order to achieve the required performance-per-cost efficiency, they typically contain multiple processing elements that are specialized in packet processing and supported by co-processors. For this survey, we examined the network processors by AMCC, IBM (now HiFn), Motorola (now Freescale), and Intel. Together they represented more than the top 4/5th of the NPU market in '02/03 [107] and cover the spectrum of deployed communication interfaces quite well. Comprehensive overviews of the different systems can be found, e.g., in [177, 205].

In network core and access equipment network processors are often deployed on line cards [199]. System examples are core routers and digital-subscriber-line access multiplexors (DSLAMs). DSLAMs aggregate thousands of individual DSL lines and forward their traffic to the core network. Routers in the core process packets at high throughput rates, e.g., perform complex lookups to determine the destination of packets.

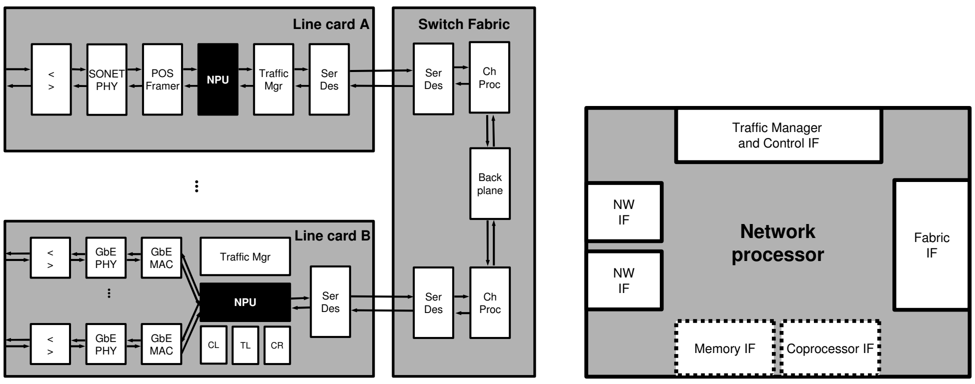


Figure 2.1: Deployment scenarios of network processors on line cards (left) and typical interfaces of a NPU (right).

In both systems, line cards manage the connection of the equipment to the physical network. They are connected to a backplane that allows packets to be distributed to other line cards or to control processors, see Fig 2.1 (left). Due to this, NPUs require a number of dedicated interfaces: *Network interfaces* connect the processor to the physical network, a *switch fabric interface* accesses the backplane, a *control plane interface* connects to an external processor that handles the control plane and also maintains the NPU and its peers on the line card, *memory interfaces* handle packets and table lookup data, and *co-processor interfaces* are used for classification, quality-of-service, and cryptography accelerators. While some of these interfaces (e.g. memory and network interfaces) are well covered and documented by mature I/O standards, others, especially the switch fabric interface, still lack sufficient standardization.

The interfaces deployed by the examined network processors are listed in Table 2.1, grouped according to our usage classes. The table confirms the fact that network processors deploy a large number of quite diverse interfaces. The table also reveals the diversity in memory and network interfaces among the processors. In the following these interfaces will be discussed more closely.

Table 2.1: Network processors and their I/O interfaces.

SoC Interface	Intel IXP 12/24/2800	Freescale C-5	IBM/HiFn PowerNP	Broadcom BCM 1250	AMCC nP3700
Network interfaces					
Number of IFs	1	16 CP[a]+ 1 SW[b]	4 PMM[c]+ 2 SW[b]	3	4
IX-Bus	(yes)	—	—	—	—
Utopia	yes	yes	(yes)	—	—
CSIX	yes	yes	yes	—	—
SPI/POS-PHY	yes	unknown	yes	—	yes
MII/GMII/TBI	—	yes	yes	yes	yes
Flexbus	—	—	yes	—	—
Memory interfaces					
SRAM	1x /2xQDR	2xZBT	2x QDR/LA-1/ZBT		2x QDR
SDRAM	1x /DDR/3xRDRAM	SDRAM (PC100)	8x(DDR)	DDR	4x DDR
Extension interfaces					
PCI	yes	yes	yes	yes	—
NPLA-1	—	—	(yes)	—	yes
Hypertransport	—	—	—	yes	—
PowerPC IF	—	—	—	—	yes
Test and debug					
JTAG	1x	1x	1x	1x	1x
Common					
Core/Type	yes/ARM	no/—	yes/PowerPC	2x/MIPS	no/—
GPIO	4-8x	—	3x	yes	—
UART	1x	(MDIO)	no	2x	(1x)
Auxiliary					
MDIO	0/(yes)	yes	(SPM)	—	unknown
Serial boot ROM	1x	1x	(SPM)	2x(SMB)	unknown

[a] one per channel processor [b] switch interface [c] physically multiplexed macros for pin-sharing

Network Interfaces. Network processors implement two sets of interfaces to connect to the physical network: Ethernet and ATM. These interfaces usually perform OSI Layer 2 functions: media access control (MAC) in case of Ethernet and segmentation and reassembly (SAR) for ATM. The physical layer (PHY layer, layer one), which is normally provided externally, is connected via one of the following three interfaces:

- The *Media Independent Interface* [82] (MII) or one of its variants (RMII, SMII) is commonly used for Ethernet and supported virtually by every NPU. These interfaces come in three performance classes: 10/100 Mb/s (MII), 1 Gb/s (GMII), and 10 Gb/s (XGMII).

- The *Utopia interface* [6, 7, 8, 9] is used for ATM environments to connect external PHY devices. Utopia has four throughput classes: 155Mb/s (L1), 622 Mb/s (L2), 3.2Gb/s (L3), and 10Gb/s (L4).

- *System-packet interfaces* [137] (SPI), which are similar to Utopia, are used for ATM and packet-over-SONET (POS) environments to interface to external framers. SPI/POS-PHY classes range in their throughput from 622 Mb/s to 40 Gb/s (SPI-5).

In order to manage the PHY modules connected to the networking interfaces, an auxiliary PHY management interface (MDIO [82]) is provided. IBMs SPM module, for instance, requires an external FPGA for this. This way, the SPM module can support different interface types.

In addition to Utopia and SPI network processors that dedicate ports for interfacing an external switch fabric use one of the following two options:

- The *common switch interface* [135] (CSIX-L1), a relatively wide interface that needs 64 data pins at max. 250 MHz to support 10Gb/s and in-band flow control.

- The *network processor streaming interface* [135] (NPSI) based on 16 LVDS data pins at 311-650 MHz and two to four additional flow control bits.

Proprietary interfaces are becoming more and more obsolete and are successively being replaced by standardized interfaces. Intel, for instance, has decided to replace the IX bus by SPI/Utopia interfaces in its second generation NPUs.

System extension interfaces. The system extension interfaces of network processors fall into two categories: control plane and co-processor interfaces. In theory, both types can be use equal IO functionality. The control plane interface, however, is more likely to be generic and well-established. The co-processor interface, on the other hand, can be optimized for low latency interactions. In practice, we find only one standard control plane interface commonly used:

- The *peripheral component interface* [149] (PCI) provides a local bus interface with a peak bandwidth ranging from 1 Gb/s to 4.2 Gb/s depending on its version. The newer PCI-X and PCI-Express specifications provide higher point-to-point peak bandwidths but are not yet integrated into products.

Some network processors such as AMCC's, integrate proprietary interfaces to connect to peer and control processors. These interfaces may be replaced in the future as more standardized alternatives to PCI (e.g. RapidIO [160] and Hypertransport [76]) become available.

Besides proprietary or network interfaces (SPI), co-processors are often used memory-mapped and connected via modified memory interfaces. For this reason, the Network Processor Forum specified the NPLA-1 [135] interface:

- The *look aside interface* is modeled based on a synchronous DDR SRAM interface and provides a bandwidth of 6.4 Gbit/s per direction at 2x 18 bit x 200 Mhz.

Memory interfaces. Although NPUs often integrate considerable on-chip memories, they in addition need at least two different kinds of external memories: large packet memory and fast memory for auxiliary data. Our NPU examples deploy the most recent memory interfaces, often in multiple instances, to meet their memory bandwidth requirements:

- *Double data rate* [90] (DDR) SDRAM supports data transfers on both edges of each clock cycle, effectively doubling the data throughput of the memory device. The ones used in our examples provide between 17 and 19.2 Gb/s @ 64 bit x {133, 150} MHz.
- *Rambus' pipelined* DRAM (RDRAM) [159] uses a 16 bit-wide memory bus. Command and data are transferred in multiple cycles across the bus (packet-based protocol). Components are available ranging from 800MHz to 1.2GHz, providing 12.8 Gb/s - 19.2 Gb/s peak memory bandwidth.
- *Zero Bus Turnaround* (ZBT) SRAM [183] do not need turnaround bus cycles when switching between read and write cycles. This is beneficial in applications with many random, interleaved read and write operations such as accessing routing and lookup tables. Motorola's NPU, e.g., deploys a 8.5 Gb/s at 64 bit x 133MHz interface.
- The *Quad Datarate* (QDR) SRAM [158] uses two separate memory ports for read and write accesses that operate independently with DDR technology, thus effectively doubling the memory bandwidth and avoiding bus turnaround. The IXP 2800 implements a total of 4x 12.8 Gb/s at 18 bit x 200 MHz on memory bandwidth based on QDR.

These memory interfaces are standard interfaces. However, memory controllers embedded in NPUs are more specialized compared with their general purpose counterparts. They need to at least support multiple processing elements, which may access a memory concurrently, e.g. by providing individual command queues. In addition, application-specific knowledge is exploited to tailor controllers to the specific usage. In Freescale's example, the queue management, buffer management, and table lookup units are specialized to their specific data structures and access functions.

Other interfaces. NPUs that implement a standard control processor on-chip also implement most of the common GPIO and UART interfaces.

2.4 Residential gateways & Integrated access devices

Residential gateways (RGs) connect the external broadband network to the internal home or small office network. They perform network-related services such as line termination, protocol and address translation, firewall security, and in-home routing. Increasingly, they are also becoming the in-house focal point for applications such as home automation and control, audio and video streaming, and file/printer sharing. Therefore, they have to deal with a plethora of different network protocols and communication interfaces. Due to the moderate throughput requirements, however, their system architectures are typically based on a limited number of embedded standard processor cores, which, in some cases, can be combined with accelerators.

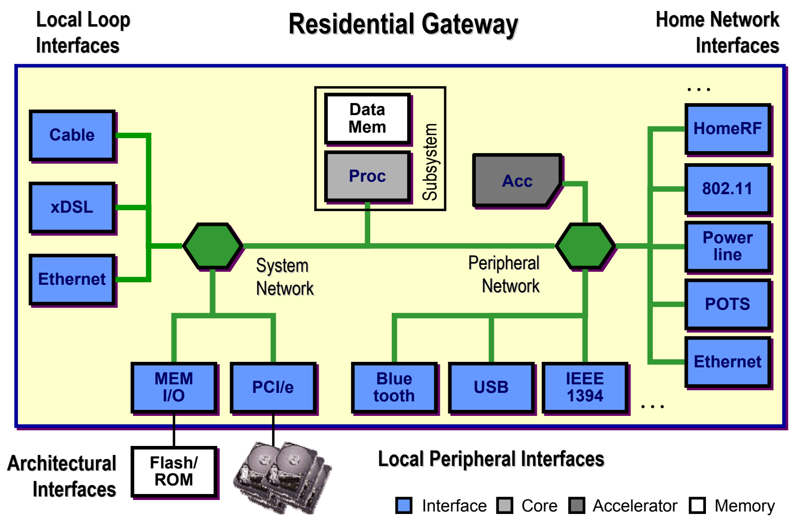


Figure 2.2: Extended RG scenario with exemplary interfaces.

In Figure 2.2, an exemplary RG is shown with its three system-level interfaces (local loop, home network, and local peripherals) and the architectural interfaces. The *local loop interface* connects to the service provider's external broadband connection (e.g. xDSL or cable). The *home network interfaces* provide a choice of wired (e.g. Ethernet, HomeRF, or power line), wireless (e.g. 802.11a/b/g) network protocols as well as plain old telephony service (POTS). *Local peripheral interfaces*, e.g. USB, IEE1394, and Bluetooth, allow for connecting directly to I/O devices without a personal computer such as to cameras, scanners, and printers. *Architectural interfaces* comprise memory interface(s) and system extension interfaces, e.g. PCI and PCMCIA. Wireless access points, for example, are an instance of the RG shown in Figure 2.2. They bridge wireless LAN and Ethernet networks and provide, e.g., a PCI/PCI Express interface for interfacing to companion chips.

Integrated access devices (IADs) are similar to RGs in their base architecture but incorporate only a limited number of network interfaces. Instead, they deploy functions for user interaction. Both systems integrate the same sets of memory interface(s) and system extension interfaces. In fact, IADs and RGs members are often of the same product families, which have the processor subsystem in common and vary only the set of integrated peripherals.

For the survey, we examined Systems-on-Chip by AMD, Intel, Samsung, Broadcom, Motorola, and Philips. Their deployed interfaces are summarized in Table 2.2, grouped according to our usage classes, and discussed in more detail in the following sections.

Table 2.2: Interfaces of residential gateways and integrated access devices.

SoC Peripheral	Samsung S3C2510 RG	Motorola PowerQUICC RG/NE	Broadcom BCM47xx RG	AMD Au1x00 IAD/RG	Philips Nexperia IAD/STB	Intel IXP42x RG/NE
Wired network interfaces						
Utopia	1x	0-2xFCC	—	—	—	0-2x
FE/GbE MAC	2x/-	0-3xFCC/2x	1-4x/-	1-2x/-	—	1-2x/—
HSS/TDM	—/—	2x/8x(SSC)	—/—	—/—	—/—	0-2x/—
HPNA	—	—	0/1x	—	—	—
DSL/Phone	—	—	0/1x	—	—	—
Local peripheral interfaces						
USB 1.1 host/ports	1x	?	1x	1/?	1x	—
USB 1.1 function	1+4	0/1	1x	1x		1x
IrDA SIR/FIR		(SSC)/-	1/-	-/0-1	(GPIO)	—
IEEE 1394					1x	
User interfaces						
LCD IF	—	—	—	0/1	—	—
I^2S/AC97, link	—	—	—	0-1/1	3x	—
SPDIF	—	—	—	—	1x	—
Memory interfaces						
Smart Card	—	—	—	—	2x	—
SDCARD/MMC	—	—	—	0/2	—	—
NAND/Fl/CF	yes	^{m}	(yes)	—	(EBI)	—
SRAM	1x	^{m}	—	1x		—
SDRAM	1x	(DDR)^m	1x	1x	1x	1x
Expansion interfaces						
PCI/PCMCIA/Cardbus	1x	PCI/PCI-X	1x	1-2	PCI	PCI
Other external Bus IF	—	60x/LocalB	1x/?	—	1x/?	1x^i
RapidI/O	—	0/1	—	—	—	—
Test and debug						
JTAG	1x	1x	1x	1x	2x	1x
Common						
GPIOS	64x	53-120/PIO	yes	32-48x	yes	16x
UART	3x	(SSC)	2x	2-4	3x	2x^l
Auxiliary						
I^2C	yes	yes	—	—	2x	—
SSI	—	—	—	0/2 SSI	1x	—
SPI	—	yes	—	0/2(SSI)^n	—	

^{i} multiple standards ^{l} 2x64B FIFOs ^{m} 1x EPROM/FLASH/EDO/SRAM/SDRAM via external bus unit interfaces. ^{n} subset of SPI NE - Network Edge, STB - Set Top Box

Network interfaces. Almost all examples deploy at least one Ethernet MAC with MII interface (cf. p. 13). Systems that implement switching functionality provide multiple Ethernet interfaces. Often, Utopia Level 2 and 3 interfaces (cf. p. 13), including SAR, are implemented that, e.g., connect to external Digital Subscriber Line (DSL) PHYs. In addition, physical layer interfaces to connect to telecommunication WAN environments are found.

- The *High-speed Serial Interface* (HSS) is a PHY interface for primary rate ISDN (E3/T3 WAN, 45Mb/s) or OC-1/SDH (52Mb/s) point-to-point links. Additionally, it can provide connectivity between LANs such as Token Ring and Ethernet. Multiple interfaces may be supported by using time division multiplex (TDM).

The only home network interfaces integrated by the examples are Ethernet and HPNA. Other protocols such as Bluetooth or wireless LAN are provided using external PHY chips and connected by system expansion interfaces.

- The *Home Phoneline Network Alliance* (HPNA) [73] interface is logically based on Ethernet. It basically provides a different PHY layer that in Version 2.0 provides 10Mb/s via telephone lines. Version 3.0 adds Quality-of-Service at 128Mb/s. With optional extensions, this version can deliver up to 240Mb/s. Version 3.1 enables transfer rates of up to 320Mb/s and adds multi-spectrum operations for network coexistence.

Some vendors integrate network PHYs on-chip. This PHY integration saves system costs for the most likely deployment scenarios. We found the following PHYs integrated: 1) 10/100 Ethernet (BCM6345), 2) HPNA 2.0 (BCM4710), and 3) ADSL transceiver and analog front end (BCM6345).

Local peripheral interfaces. There are three deployed local peripheral interfaces: USB, Infrared and Firewire. All three protocol stacks implement higher protocol layers that allow for their integration in a home or office IP/LAN network:

- *Universal Serial Bus* (USB) [200] interfaces enable serial data transfers from 12Mb/s (v1.1) to 480 Mb/s (v2.0). A USB 1.1 function, which allows the system to respond to an external master, seems common among our examples. RGs often implement an additional host function. This way, the RG can work stand-alone and access external peripherals actively without the help of a PC and can also serve as an intelligent bridge between peripherals. Similar to the physical layer of network interfaces, we also found some systems that implement the USB PHY layer on-chip.

- *Infrared interfaces* by the Infrared Data Association (IrDA) [85] are also implemented throughout, although in different versions. RGs and network edge systems often integrate the low end serial infrared version (SIR) only, which uses an UART and provides speeds of up to 115kb/s. The higher speed versions MIR (1.5 Mb/s) and FIR (4.0 Mb/s) are more likely to be deployed by IAD architectures. The new VFIR (12Mb/s) version does not yet seem deployed.

- The *Firewire/IEEE1394* interface [4, 81] is deployed only by Philips' Viper system. This protocol seems to be used particularly in combination with image and audio data streams due to its relatively high bandwidth of up to 400 Mb/s (1394a) or 800 MB/s (1394b).

User interfaces. AMD's Au1100 and Philips Viper devices are the only devices that implement a standard LCD interface, video function, and audio codec controller. There are no other dedicated peripherals for user interaction. Instead, user I/O devices are connected via multi-function interfaces such as UARTs or

GPIO if necessary. Even a I^2S [152] interface, a 3-wire interface dedicated to transport audio data, requires such a general purpose channel to transfer control to the external audio devices.

Memory interfaces. All devices provide interfaces and controllers for standard SDRAM. Intel, Broadcom, Philips, and Samsung implement a dedicated interface for this purpose. Motorola even implements controllers for each of its two external buses. Some systems provide a second controller for static memory devices. AMD's controller, for instance, supports SRAM, Flash, ROM, page mode ROM, PCMCIA/Compact Flash devices, and an external LCD controller via its memory interface. Others use system extension buses instead of individual interfaces. In addition, secure digital card interfaces have been implemented by AMD.

System extension interfaces. IADs and RGs overlap memory and system interface functions to some extent. Both interfaces seem interchangeable. In some cases, external peripherals, e.g. an LCD controller, are connected to the memory interface, and in other cases, memory devices, e.g. flash memory, are accessed via external bus interfaces. All systems implement either PCI or PCMCIA/Cardbus interfaces. Motorola implements PCI-X and RapidI/O in its PowerQUICC III system. These interfaces are used, for instance, to connect to external wireless LAN modules.

Common interfaces. Multiple UARTs and a large number of general purpose I/O are implemented by the examples. They are used to provide additional interfaces, e.g., the chip select in case of Motorola's memory controller, or for sideband interfaces for external PHYs.

Other interfaces. There is a fair number of different serial interfaces (I2C, SPI, and SII) for moderate communication bandwidth. These interfaces are general purpose and are used for connecting to external system components, mostly on a board-level.

2.5 Wireless application processors

Wireless application processors (WAPs) handle the application part of cellular phones and hand-held devices. Apart from the radio baseband processing, this part often comprises other functions, e.g. audio/video, security, and storage. Wireless application processors can be seen as IADs with an (external) wireless network interface. Since they are used in portables, much more emphasis is put on energy efficient architectures than seen before in Section 2.4.

In the following product families by TI, ST, and NeoMagic are being examined. The systems commonly contain at least one ARM9 core and additional programmable accelerators. In addition, TI integrates a DSPs for GSM support. Table 2.3 summarizes the deployed interfaces grouped by our usage classes.

Network interfaces. Wireless application processors do not integrate specific network interfaces. Only TI's OMAP mentions a wireless LAN interface. It is, however, a standard port that is capable of handling the 54 Mb/s bitstream (cf. TNETW1130 [193]). Similarly, other wireless chips, e.g. baseband processor and Bluetooth PHY, are connected via standard interfaces such as UARTs or multi-channel serial interfaces.

Table 2.3: Interfaces of wireless application processors.

SoC Peripheral	NeoMagic MiMagic6 PDA/AP	TI OMAP AP	ST Nomadik AP
Wireless network interfaces			
wireless LAN a/b/g	—	(1x)	—
Local bus interfaces			
USB host/ports	—	1x/3	1x
USB function	1x	1x	?
IrDA SIR/FIR	1x/1x	1x/1x	?/1x
User interfaces			
Camera IF	1x+acc.	1x[i]	1x+acc.
LCD IF	1x+acc.	1x[i]	1x+acc.
I2S/AC97, link	1x/1x	0-1x/1x	MSP/MSP[j]
Keypad	—	1x	—
Memory interfaces			
Smart Card/SIM card	—	0-1x	—
SDCARD/MMC	2x	1-2x/SPI	1x
NAND/Flash/CF	1x	1x	1x
SDRAM/DRAM	1x	1x/DDR	1x/DDR
Expansion interfaces			
PCMCIA	(1x)	—	—
Test and debug			
JTAG	?	1x	1x
EMT9	??	1x	??
Common			
GPIOS	yes	14x	76x
UART	3x	2-3x	2x
Auxiliary			
I^{2}C	1x	1x	2x
/u-wire	—	1x	—
SPI	2x	2x	MSP[j]
Pulse-Width-Light	—	1x	—
Pulse-Width-Tone	—	1x	—
HDQ/1Wire (Batt)	—	1x	—
LED Pulse Gen.	—	2x	—

[i] multiple standards, but no details provided [j] SPI, I^{2}S, AC97

Local peripheral interfaces. Like IADs, application processors deploy two local interfaces: USB and Infrared. In case of USB, both, function and host interface, are commonly implemented based on FIR (4 Mb/s). The new VFIR (12Mb/s) version seems not yet have been deployed in the wireless domain.

User interfaces. Specialized LCD and camera interfaces are implemented by all processors. Often, they are associated with accelerators for better performance/power trade-offs. AC97 and I^{2}S interfaces are provided by virtually all examples. In some cases, multi-channel serial peripherals are used, which also support other kinds of serial interfaces. TI's keypad interface seems a proprietary solution.

Memory interfaces. All processors implement a DRAM memory port. ST and TI use DDR technology for lower power consumption. In TI's case, the interface is used for flash, compact flash, and NAND flash memories as well, whereas other vendors implement a static memory controller for this purpose. A second common peripheral is the secure digital card/multimedia card interface.

Furthermore, TI lists a special SIM card interface. Such SIM cards (or smart cards) are used in GSM phones to identify user accounts and to provide data storage. They also contain processors that execute programs and communicate over a low speed three wire serial interface.

System extension interfaces. In general, WAPs do not integrate system extension interfaces but use other types for similar purposes such as various serial and auxiliary interfaces to link to baseband part, audio codecs, and wireless PHYs. As an alternative for higher bandwidth, the memory interface is used. The only PCMCIA port among our examples, for instance, is provided by Neomagic's static memory interface.

Common interfaces. The systems implement a similar number of UARTs in all examples, but quite a different number of general purpose I/O. The small number of only 14 GPIOs in TIs case, is certainly compensated by the larger number of small auxiliary interfaces.

Other interfaces. Serial interfaces such as I^{2}C and SPI are implemented in all processors. TI and ST implement them partly in multi-channel serial peripherals that support multiple serial standards. TI's SD card interface can be used as SPI port, too. TI also implements a number of small auxiliary serial interfaces for display lights, battery status, and sound/speaker control.

2.6 Discussion and selection of interfaces

In the previous sections we surveyed a number of domain-specific system-on-chip and summarized their use of IO interfaces. In this section, we are compiling the data on the deployment, discussing trends towards flexibility and selecting an essential set for the subsequent analysis of this dissertation.

Deployed interfaces per die

Depending on the application domain, our systems implement between 8 and 25 different interfaces on a single die. Among the three domains, network processors integrate the least number of interfaces with an average of 10, followed by residential gateways with 12 in average; application processors with an average of 18 use the most communication interfaces, as shown in Table 2.4.

Table 2.4: Number of different interfaces per SoC and domain.

Domain	Network Processors	Residential Gateways Internet Attached Dev.	Wireless Processors	Across domains
per SoC Average	9, 8, 10, 8, 11 10	13, 10, 10, 13, 10, 15 12	14, 25, 14 18	8 - 25 12

Table 2.5 shows the set of essential interfaces derived from the domain-specific lists of deployed interfaces. The criteria used for deciding whether an interface is representative and should be included in the essential set is its common implementation into one domain or the availability of standards. Such interfaces are normally well documented and their specification is publicly available. Basically, the final set is the superset of the domain-specific lists and has approximately three dozen members. Due to their disjoint function and the lack of standardization are display and camera interfaces excluded. Although not integrated in the examples, PCI Express has been included due to its foreseeable deployment in next generation devices. We have also included the wireless LAN (IEEE 802.11) family of standards due to their increasing relevance for home and hand held devices.

Table 2.5: Essential set of deployed communication interfaces.

Interface class	Deployed standard interfaces
Network	Utopia, SPI, Ethernet (MII), HSS, TDM wireless LAN
Local peripheral	USB, IrDA, IEEE1394
User	AC97, link, I^{2}S
Memory	SDRAM, DDR, RDRAM, SRAM, QDR, ZBT, NAND, Flash, CF, SDCARD, MMC, SmartCard
System extension	PCI, PCMCIA, Cardbus PCI Express, Hypertransport, RapidIO
Test and debug	JTAG
Common	GPIO, UART
Other	MDIO, I^{2}C, SPI, SSI

From the set of 35+ essential communication interfaces, our systems-on-chip integrate between 10 and 20 diverse IO functions of varying performance requirements. Their heterogeneity has lead to several different implementation techniques for IO interfaces.

Implementation techniques for communication interfaces

The examination of interfaces in the preceding subsections revealed a tendency toward the support of different communication standards. This is especially true for the network interfaces of network processors; they are configurable for multiple standards. Other interfaces such as serial interfaces or memory controllers can be configured and parameterized as well. We have observed three approaches towards flexibility that differ in their degree of configurability from the hardwired solutions mentioned first.

- *Individual IP* – Every interface is implemented individually and connected to dedicated pins. Although this is the common case, it leads to a larger number of heterogeneous blocks in the system and to high area consumption. Memory interfaces, for instance, are usually customized for a particular RAM technology.
- *Multiple Macros/pin sharing* – The simplest technique to provide flexibility is the implementation of multiple macros, which can be used alternatively in a multiplexed fashion. This way, the macros can share the expensive pins. Obvious drawbacks of such a solution, however, are the heterogeneity and the area consumption.
  - The physically multiplexed modules of IBM's PowerNP integrate four different network interfaces.
  - Concurrent implementations of USB host and function macros can be found (e.g. in Samsung's residential gateway S3C2510 system). However, they do not necessarily share pins.
  - Freescale's PowerQUICC contains a memory controller with three different modules: a high performance SDRAM interface module, a more power-efficient lower performance module (e.g. without burst mode), and three parameterizable modules.
- *Parameterizable Interfaces* – Virtually all peripherals can be adjusted to some extent by specifying a well-defined set of parameters. The degree to which a peripheral is parameterizable (the set and range of parameters) varies with the interface. Here are several examples:
  - UARTs – Serial interfaces can be configured, for instance, for different transmission speeds and the use of hardware support for protocols (e.g. Xon/Xoff).
  - Memory Controller – Timing parameters and organization of the external memory modules can be set up at boot time.
  - PCI Interface – Some PCI interfaces support multiple versions that differ in bus frequency and width.
  - IBM's Physical MAC Multiplexer (PMM) – The selection signal for multiple (fixed) macros is also a parameter.

  Since such parameters are configured (i.e. programmed) by the processor core and the number of parameters may be substantial, these interfaces are sometimes wrongly considered to be programmable.

- *Programmable Interfaces* – In some interfaces, parts of the functions are carried out by specialized co-processing elements that are micro-programmable. The capabilities of these elements as well as their association to particular interfaces vary:
  - Freescale's Serial Data Processors (SDPs) – For a limited set of protocols (Ethernet and ATM) the SDP engines handle individual low level tasks such as bit field extraction and CRC calculation. Each of the 16 network interfaces included by the C-5 uses two SDPs.
  - Intel's Network Processing Engines (NPE) – The IXP 4xx systems implement these programmable accelerators to support individual network interfaces. The engines are specialized to the needs of a particular interface but are derived from the same core architecture. This helps to offload computation-intensive tasks from the CPU.
  - Freescale's Communication Processor Module (CPM) – The CPM contains a processor core that is shared by multiple interfaces and only executes higher-layer communication protocol layer tasks effectively separating these tasks from the main processing system.

In addition to these approaches, orthogonal reuse techniques are found, which can be combined with any of the approaches above but actually do not require any configurability.

- *Multiple Channels* – Multi-channel interfaces handle multiple (communication) contexts simultaneously. They exploit the fact that some communications (e.g. time slots in a time division multiplexed link) do not require the full performance of a module. Some of our examples also support multiple protocols:
  - TI's multi-channel serial interfaces (MCSIs) – In TI's OMAP processors, configurable MCSIs are used (e.g. in clock frequency, master/slave mode, frame structures or word lengths). Their buffered version (McBSP) allows for continuous data streams for a number of different communication protocols (T1/E1, AC97, I2S, SPI).
  - Freescale's multi-channel controllers (MCCs) – Each MCC supports up to 128 independent TDM channels (HDLC, transparent or SS7).
- *Use of generic interfaces* – An additional approach to increase reuse and flexibility is the use of generic interfaces instead of specialized ones. We observe three different applications:
  - Wireless application processors use interfaces such as UARTs rather than proprietary solutions (e.g. to connect to companion chips (e.g. Bluetooth PHY)).
  - Almost every system provides a number of general purpose I/Os that can be used under software control for any I/O function at the expense of CPU performance.
  - Some systems implement generic interfaces such as the serial management interface in IBM's network processor that may require additional external logic to support less common usage scenarios.

Focus and set of interfaces for this thesis

This thesis adopts a depth-first approach to the problem of interface diversity. We are not attempting to address a solution for every communication interface of the essential set. Instead, the focus is placed on the emerging class of packet-oriented communication interfaces.

Recent communication standards put much effort in better exploitation of the physical channel and service quality addressing the need for high communication bandwidths at low costs, see Figure 1.4 on page 4. This in turn has lead to increasingly complex modules which must rely on packet-oriented communication protocols to ensure reliable transfers. Packet-oriented interfaces cover the better part of the essential interface set. Virtually all network (Utopia, SPI, Ethernet, wireless LAN), system extension (PCI Express, Hypertransport, RapidIO), and local peripheral interfaces (USB, IrDA, Firewire) belong to this type. The remaining IO modules either are memory interfaces or require only limited performance.

From the class of packet-oriented communication interfaces, we have chosen

PCI Express, Hypertransport, RapidIO, Ethernet, and wireless LAN

as examples to be used in this dissertation. The choice of these particular packet interfaces was motivated by the communication needs of two multiprocessor SoC products: first, a *network processor* device as part of a DSL line card which uses Ethernet ports as network interfaces and requires either RapidIO, PCI Express, or Hypertransport as control plane and switch fabric interfaces, and second, a *wireless access point*, which deploys wireless LAN, additional Ethernet ports, and PCI Express as system extension interface.

We are primarily interested in the higher level interface and protocol aspects of the IO modules and intend our work to be complementary to the efforts of the software-defined radio community or initiatives such as the Unified 10Gbps Physical-Layer Initiative [134] which all focus on the physical characteristics of IO protocols.

Chapter 3

Analysis of Packet-oriented Communication Interfaces

The previous chapter identified packet-oriented communication interfaces as the largest subset of essential IO standards. But how similar are these interfaces? What are their elementary tasks? In order to answer these questions, this interface class will now be analyzed on the functional level.

Following a common structure, the five standards PCI Express, RapidIO, Hypertransport, Wireless LAN, and Ethernet are analyzed. Each interface is modeled in Click. Click is an established framework for describing packet processing applications. The models are functionally correct and capture the essentials of each interface, hence enabling a detailed and quantitative comparison.

Before concluding this chapter with the similarities and dissimilarities of packet-oriented interfaces, related work on the modeling, analyzing, and comparing of communication protocols is discussed.

3.1 Fundamentals

As the basis for the comparison and analysis in the subsequent sections of this chapter, this section describes the fundamentals of communication interfaces. The first subsection discusses aspects related to the SoC integration. The next two sections then focus on protocol-related concepts and identify elementary tasks that are common in packet-oriented interfaces.

3.1.1 Interface surroundings

Communication interfaces (or IO modules) are building blocks for Systems-on-a-chip. They interact with their SoC environment in two distinct directions: to the outside world via the physical interface and to the SoC core via the transaction interface. Figure 3.1 shows a communication interface embedded in its environment.

The physical *link* interface is formed by one or more *lanes*. For duplex communication, a lane consists of two opposite unidirectional point-to-point connections. The bandwidth of such links can be scaled by changing the number of lanes (cf. Fig. 1.4). In the case of serial protocols such as RapidIO,

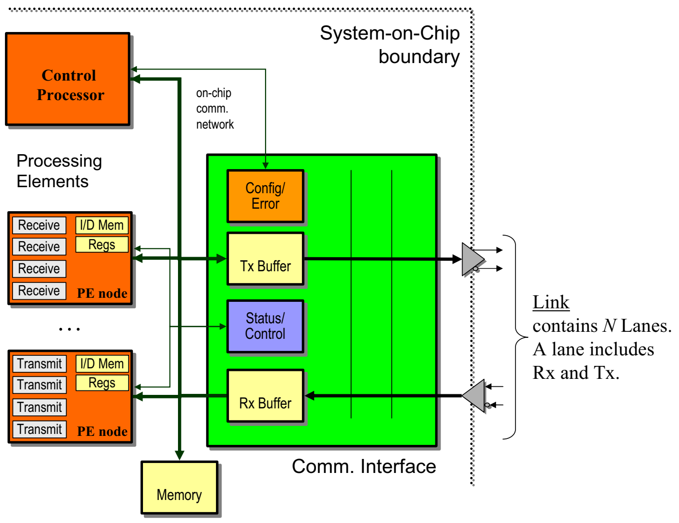


Figure 3.1: A communication interface and its environment.

PCI Express, and Gigabit Ethernet (1000 BaseT) a lane contains only the four differential data signals. Hypertransport may have an explicit clock signal per lane (synchronous mode) and an additional control signal per link.

The *transaction interface* allows the SoC cores to control the sending and receiving of data units, the transactions. The SoC may issue requests or respond to incoming requests by returning completion transactions (split transaction model). For interaction efficiency, the module interface may incorporate building blocks such as masters for the on-chip communication network, DMA engines for accessing the core system's memories, and local transfer buffers. Distinct address spaces (memory, IO, configuration, and message) may be defined to support different communication semantics in the system.

Each interface provides a *configuration space* which contains the parameters and status information of the IO module. Access to this space is possible via configuration transactions from both directions. A separate internal interface to the SoC core can be provided for lean and direct access.

3.1.2 Functional layers

The function of IO interfaces can be structured in three layers following the OSI/ISO reference model as shown in Figure 3.2. Each layer provides a service to the above layer by using the service of the layer below. Peer layers on different nodes communicate with each other by using their particular protocol aspects, which implies using the services from the layer below. The three layers are:

- **Physical layer** (PHY) — which transmits data via the physical link by converting bits into electrical signals.
- **Data link layer** (DLL) — which manages the direct link between peer nodes and reliably transmits pieces of information across this link.
- **Transaction layer** (TA) — which establishes a communication stream between a pair of systems and controls the flow of information. Since

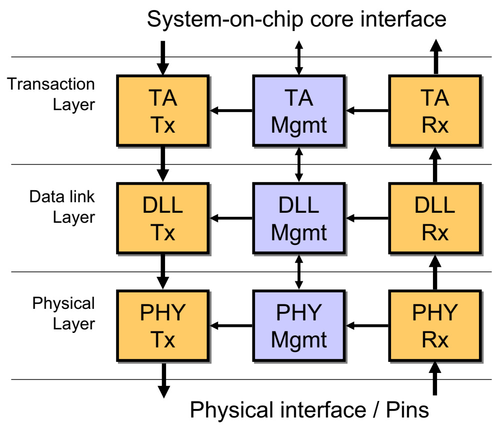


Figure 3.2: Layered structure of packet-oriented communication interfaces.

switched networks require routing, different variants of the transaction layer might be used in endpoints and switches.

Data flows through these layers in two independent directions: the transmit path (Tx), which sends data from the device core to the link, and the receive path (Rx), which propagates data from the link to the device core. At each layer, additional control and status information for the peer layer of the opposite device is inserted (Tx side) into the data stream, or data sent by the peer layer is filtered (Rx side) out of the stream by the layer management block. A transaction at the transaction layer is represented as a packet. The link layer forwards these transaction packets and generates its own data link layer packets for the exchange of state information. The PHY layer first converts all packets into a stream of symbols which are then serialized to the final bitstream.

The terminology of the layers follows the PCI-Express specification. RapidIO uses a similar layered structure (logical, transport, and physical). Hypertransport, however, does not use layers explicitly but relies on the same functions and structural elements. Gigabit Ethernet and wireless LAN functionality is usually described using PHY and MAC layers. We have divided the MAC layer in DLL and TA aspects. Section 3.1.3 will describe the function of each layer for each protocol in more detail.

3.1.3 Elementary tasks

The function of the different packet-oriented communication interfaces is provided by a set of elementary tasks. This section describes the common set, starting from the physical layer upwards. In order to be considered common a task must be used by at least two of our protocols.

Clock recovery Serial interfaces encode the transmit clock into the bitstream. At the receiver, the clock needs to be recovered from data transitions in the bitstream using a Phase-Locked Loop. To establish bit lock, the transmitter sends specialized training sequences at initialization time.

Clock compensation The PHY layer must compensate frequency differences between the received clock and its own transmit clock to avoid clock shifts.

Lane de-skewing Data on different lanes of a multi-lane link may experience different delays. This skew needs to be compensated at the receiver.

Serialization/Deserialization The width of the internal data path must be adjusted to the width of the lane. Serial interfaces also need to lock the bitstream to symbol boundaries.

8b/10b coding/decoding Data bytes are encoded into 10 bit symbols to create the bit transitions necessary for clock recovery.

Scrambling/Descrambling Scrambling removes repetitive patterns in the bitstream. This reduces the EMI noise generation of the link.

Striping/un-striping In multi-lane links, the data is distributed to/gathered from individual lanes byte-wise according to a set of alignment rules.

Framing/Deframing Packets received at the physical layer are framed with start and end symbols. In addition, PHY layer command sequences (e.g., link training) may be added.

Carrier Sensing This is used to determine if a shared medium is available prior to transmission.

Cyclic redundancy check (CRC) The link layer protects data by calculation of a CRC checksum. Different CRC versions may be required depending on data type and protocol version.

Ack/Nack protocol In order to establish a reliable communication link, the receiver acknowledges the error-free packets by using a sequence number. In the case of transmission errors, the packet is not acknowledged (or a not-acknowledge is returned) and the packet is retransmitted.

Classification The classification according to packet types may be based on multiple bit fields (e.g., address, format, type) of the header.

Packet assembly/disassembly The transaction layer assembles payload and header and forms outgoing packets according to the transaction type. The link layer may add an additional envelope (e.g., CRC, sequence number). The link layer may generate information packets (e.g., Ack/Nack, flow control) to update the link status of its peer.

Flow control A transaction is only transmitted to the receiver if sufficient buffer space is available at the receiver. The receiver updates the transmitter periodically with the amount of available buffer space.

Address validation Incoming transactions should only be addressed to the device or its memory spaces, respectively.

Buffers and scheduling A set of individually flow-controlled buffers is required for all transaction types to prevent head of line blocking. Additional sets for quality-of-service may be used.

Configuration and management space The configuration space stores the identity of the device which is determined during the initialization phase and the negotiated link parameter. It also allows access to internal state and error logs.

Table 3.1, shown later in Section 3.5, groups the elementary tasks according to their appearance on the different layers in RapidIO, Hypertransport, PCI-Express, Gigabit Ethernet, and wireless LAN. The table will reveal quite some similarities in occurrence and deployment of these tasks. The functional modeling of end-point devices for the individual standards in the following sections will capture the specifics of each interface.

3.2 Modeling packet-based interfaces in Click

For detailed analysis the full system function of the interfaces and their surroundings are modeled. The comparison of the models provides insights into the similarities of the different protocols and allows for the characterization of the packet-oriented interface application domain.

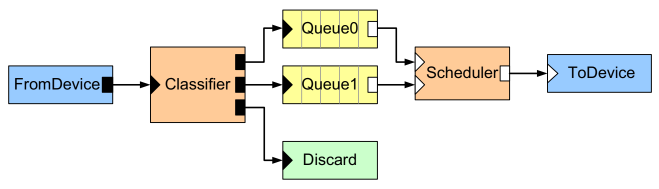
We implement our models in Click [102], a domain-specific framework for describing network applications. Click was chosen for several reasons. Click models are executable, implementation-independent, and capture inherent parallelism in packet flows and dependencies between elements naturally. Furthermore, Click's abstraction level and the rich and extensible element library allowed us to focus on interface specifics. By using Click a functionally correct model can be derived quickly. And even more important, typical use cases, traffic conditions, and the ideal protocol behavior can be validated precisely.

Click is open source software, implemented in C++, and runs on Linux platforms. This section briefly introduces Click's main characteristics and describes concepts for modeling packet interfaces. Further information on Click can be found in [100] and on Click's web site [194]. The Click interface models will be described in subsequent sections.

3.2.1 Click characteristics

In Click, applications are composed in a domain-specific language from elements that can be linked by directed connections. The elements describe common computational network operations whereas connections specify the flow of packets between elements. Packets are the only data type that can be communicated; their generic format supports arbitrary communication protocols. All application state is kept local within elements. Two packet communication patterns are distinguished in Click: push and pull. Push communication is initiated by a source element and models the arrival of packets in the system. Pull communication is initiated by a packet sink and models space that becomes available in an outbound resource.

Figure 3.3 shows a simple Click example using both a graphical syntax and the Click language. Packets are inserted into the system by the *FromDevice* element (push output [black]). The packets flow to the input port of the *Classifier* element. This element forwards a packet to one of its output ports depending on the result of the internal processing, e.g., filtering header fields. Two outputs are connected to queues. In Click, queues are explicit elements that have push inputs and pull outputs (white). Hence, the *ToDevice* can pull packets out of the queues at its own rate and remove them from the system. Pulling the packets happens via the packet *Scheduler*, which selects a packet from its inputs depending on its policy.



```
FromDevice -> cl::Classifier [0] -> q0::Queue -> [0] s::Scheduler -> ToDevice;
                        cl [1] -> q1::Queue -> [1] s;
                        cl [2] -> Discard;
```

Figure 3.3: Click example: graphical and textual representations.

3.2.2 Interface specifics

Click implements software based routers on Linux computers, turning them into network nodes with considerable packet processing capabilities [20]. The software uses OS functions and interacts closely with the OS, e.g., to access the physical network ports and their packet streams. Click runs as a CPU thread inside the Linux kernel. Work is scheduled by maintaining a list of elements that start Click processing chains.

To model the full system function of our communication interfaces in Click, a number of issues have to be addressed:

- Simulation mode — For the functional verification of our interfaces we used Click in user mode only. Our models also describe the environment, namely traffic sources, sinks, and physical communication channels and do not depend on external events.

- Precise timing — Our interfaces handle timeouts and must guarantee response times. Click cannot guarantee precise timing since it relies on cooperative scheduling. Our models do not depend on the absolute system time. Thus, timing accuracy can be achieved by using timing intervals that are large compared to the packet processing time, i.e. the simulation does not achieve real-time.

- Flow of control information — Click supports the annotation of packets with processing state. In some packet flows, however, state information generated downstream needs to be fed back into an upstream element. To achieve the proper granularity, we explicitly modeled such dependencies using tokens. In push connections a token represents the state change event. A pull token indicates reading state access.

- Non-packet data types — Besides transaction and link layer packets, state tokens and symbols are used. Both of them are also represented internally by Click packets. Elements may convert packets into tokens or symbols and vice versa.

- New elements — We used existing click elements whenever possible, given Click's focus which is for standard packet flow and processing functions on layer 3, and above. In some cases, our interfaces required special elements, which, for instance, provide the Layer 2 processing that would be covered by the port hardware of the Linux box.

3.2.3 Model setup

Figure 3.4 shows the most basic simulation setup of the interface models using a directed point-to-point link. An on-chip source communicates via the transmit part of an interface to a peer system with its receive part. The peer responds using a 2nd directed link (not shown). Packet arrivals and departures trigger the execution of processing paths. In the model, transmit and receive paths are timed by the *Channel* element, which pulls symbols out of the transmitter and pushes them into the receiver at a constant rate.

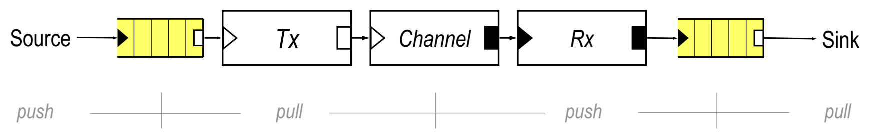


Figure 3.4: Basic simulation setup for interfaces that use point-to-point simplex communication.

A statically configured rate is sufficient if the data is transferred in chunks of the same size, i.e. with constant bandwidth. The PCI Express, RapidIO, Hypertransport, full duplex Gigabit Ethernet models in the next section use this simulation setup for their point-2-point connections.

For such connections the transfer delay is negligible. This is different for communication via a shared medium. The fact that the medium is busy for the time of the transfer is crucial for correctly accessing the medium. Collision detection and avoidance depend on the precise transfer delay, as shown in Figure 3.5. The shared channel receives a packet (P_1) at an input port and delays its transfer to the output port(s). If another packet (P_2) arrives while the first one is still in transmission, the packets collide. Since both will be received scrambled at their scheduled arrival times, the receiver can detect the collision. Collision avoidance is modeled by the extra pull port of the channel. If busy, transmitters pull busy tokens which they can use to delay their transmission.

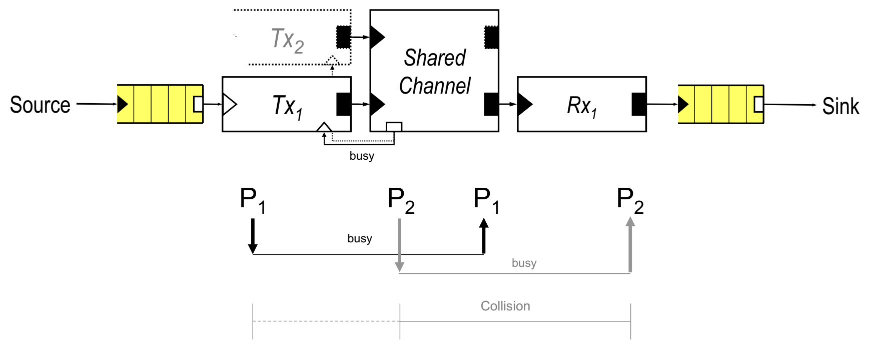


Figure 3.5: Simulation setup for shared communication channels (top). For correct handling of Packet collisions (P_1, P_2) transfer delays must be considered (bottom).

This setup in combination with a statically configured rate can be used by half-duplex Ethernet. Wireless LAN transfer rates change dynamically. Therefore, the shared channel must support variable rates on a per packet base.

3.3 Application models

In this section, the individual models are presented and important communication scenarios are discussed. The models capture the complete data and control flow for the steady state of an interface device. Initialization, configuration, status reporting, and other management related aspects are simplified. Since we are not interested in physical properties, we do not model clock recovery, or any synchronization on the physical layer. We verify our models by simulation using communication patterns that were derived manually from the specifications.

3.3.1 PCI Express model

PCI-Express is a serial, packet-oriented, point-to-point data transfer protocol [25]. There are two different versions of PCI-Express: Base and Advanced Switching. The Base specification [150] preserves the software interface of earlier PCI versions. The Advanced Switching version [1] defines a different transaction layer than the base version to add features important to the networking domain, such as protocol encapsulation, multicast, and peer-to-peer communication.

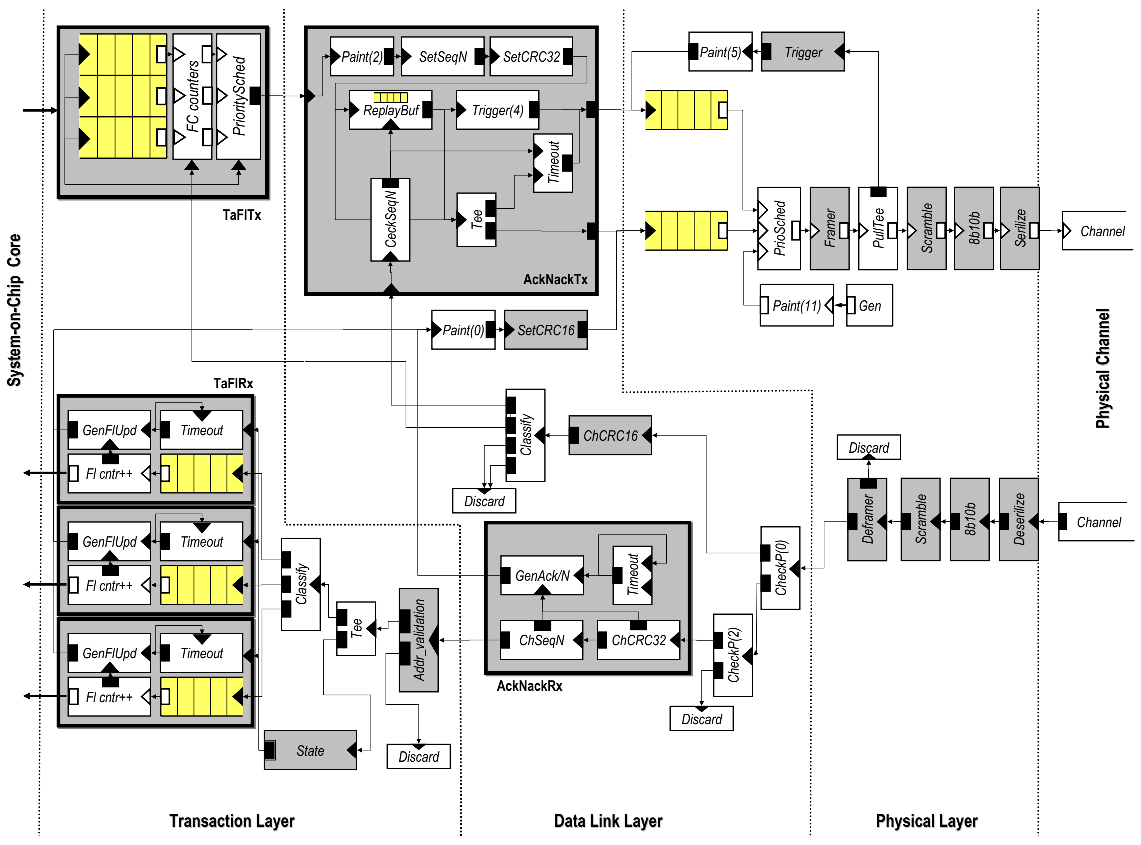


Figure 3.6: PCI-Express end-point device interface model.

For the purpose of this analysis, we use the Base specification [150] for modeling the critical path of an end-point device. The Click diagram for our implementation is shown in Figure 3.6. The functionality of the elements was described in Section 3.1.3 on page 27. Based on the model, six cases (A–F) of information flow through the interface can be identified:

A) Outbound transactions. The SoC core initiates a transaction (e.g., a read request) by transferring data and parameters into the transaction buffer, which is a part of the transmit transaction layer (TaFlTx element in the figure). This buffer implements at least three queues to distinguish between posted, non-posted, and completion transactions, which together represent one virtual channel. Posted transactions such as memory writes do not require a response; non-posted transactions, e.g. memory reads, require a response; completion transactions are the response for non-posted transactions. From the buffer, transactions are forwarded to the data link layer, depending on the priority and the availability of buffer space at the receiver side of the link. When a transaction leaves, flow control counters are updated. The data link layer (AckNackTx), adds a sequence number, encapsulates the transaction packet with a CRC, stores a copy in the replay buffer, and forwards it to the PHY layer. At the PHY layer, the packet is framed and converted into a stream of symbols. The symbols are, if necessary, distributed onto multiple lanes, encoded and serialized into a bitstream before they are transferred to the channel. The serialization shown in the figure is not modeled. Therefore, the channel runs at symbol time resolution (cf. Sec. 3.2.2).

B) Inbound transactions. A stream of encoded symbols enters the receive side of the PHY layer and is decoded, assuming that clock recovery, compensation, lane de-skewing, and de-serialization have already been performed. The Deframer detects and assembles symbol sequences to PHY layer commands and packets. Packets are forwarded to the link layer. The link layer classifies incoming packets into transaction packets, link layer packets, and erroneous packets. Transaction packets that pass the CRC and have a valid sequence number are forwarded to the transaction layer (AckNackRx). Erroneous packets are discarded. For each received transaction an acknowledge or not-acknowledge response is scheduled. At the transaction layer, the received transaction (e.g., a read completion) is stored into the appropriate receive buffer queue and the SoC core is notified. As soon as the transaction is pulled from the queue, the receive flow control counters can be updated, and the transfer is completed.

C) Outbound acknowledge packets. The link layer generates confirmation packets to acknowledge/not acknowledge the reception of transaction packets (AckNackRx). To preserve bandwidth, they are issued in scheduled intervals rather than after every packet. Besides the ack/nack type, a packet contains the lastest valid sequence number and is CRC protected.

D) Inbound acknowledge packets. If the received link layer packet has a valid CRC and is an ack/nack, its sequence number *SN* is verified (AckNackTx). When a valid acknowledge has been received, all transactions with sequence numbers not larger than *SN* can be purged from the replay buffer. Otherwise, transactions with larger numbers are retransmitted. If there are too many retransmissions (more than four) or no ack/nack packet has been received, a link retraining command will be issued to the PHY layer.

E) Outbound flow control packets. After having read a transaction from the receive buffer and changed the receive flow control counters, the transmitter

has to be updated. For this purpose, the link layer issues a flow update packet, which is generated from the counter values provided by the TA layer. In the initialization state, init packets instead of updates are issued to the transmitter.
F) Inbound flow control packets. Inbound flow control packets are forwarded by the receiving link layer to the transaction layer. The TA layer updates its transmit flow control counters and schedules the next packet for transmission from the pending transaction buffer.

3.3.2 RapidIO model

RapidIO is a packet-oriented, point-to-point data transfer protocol. Like PCI-Express, RapidIO is a layered architecture [161, 48, 180]: The logical layer (our transaction layer) specifies the transaction models of RapidIO, i.e. I/O, message passing, and globally shared memory; The transport layer specifies the routing of packets through the network (our transaction layer covers the part which concerns end devices: the device identification); The physical layer defines the interface between two devices (our physical layer) as well as the packet transport and flow control mechanisms (our data link layer).

Since there are only minor differences between PCI-Express and RapidIO in terms of tasks and packet flow, we are refraining from presenting the complete implementation here. Instead, the differences on each layer are listed.

- The transport layer implements a different buffer scheme with four prioritized transaction queues that are flow-controlled together.

- At the link layer, an explicit acknowledge for each packet is required, whereas PCI Express allows the acknowledge of a packet sequence. The not-acknowledge provides the cause for an error, which is used for individual reactions at the transmitter.

  RapidIO appends a 16 bit CRC at the end of packets. In the case of long packets, an interim CRC value is added after the first 80 bytes of the packet. The much shorter control packets are protected by a 5bit CRC. PCI Express uses 32 bit CRCs for transaction packets and 16bit CRC for link layer packets.

- The PHY layer uses slightly different control symbols than PCI Express.

3.3.3 Hypertransport model

Hypertransport is a parallel, packet-oriented, point-to-point data transfer protocol for chip-to-chip links [198, 79]. Version 1.1 of the standard [77] extends the protocol with communication system-specific features such as link-level error recovery, message passing semantics, and direct peer-to-peer transfer. In our work, we primarily used the version 1.04 described in [198].

Unlike RapidIO and PCI-Express, the packet transfer portion of a link comprises groups of parallel, uni-directional data signals with explicit clocks and an additional sideband signal to separate control from data packets. Control packets are used to exchange information, including the request and response transactions, between the two communicating nodes. Data packets that just carry the raw payload are always associated with a leading control packet. To improve the information exchange, the transmitter can insert certain independent control packets into a long data transfer.

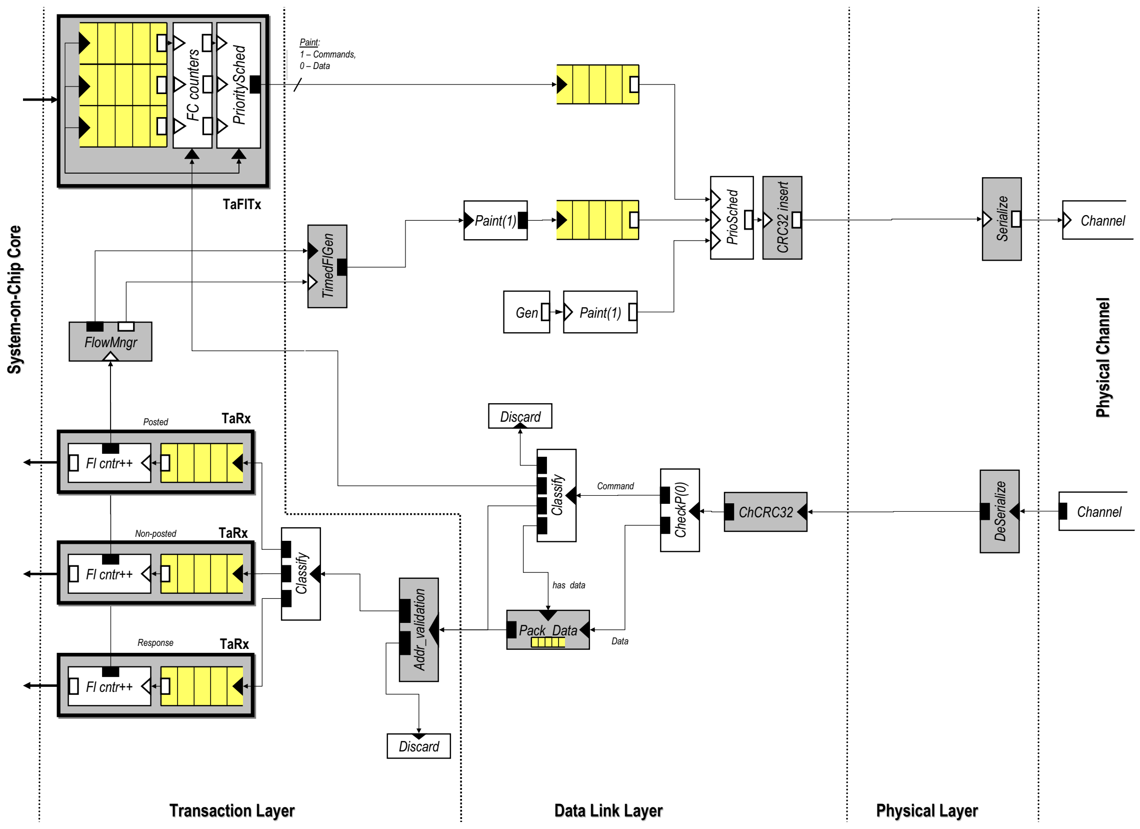


Figure 3.7: Hypertransport end-point device interface model.

The base functionality of the Hypertransport protocol is comparable to PCI-Express and RapidIO. However, Table 3.1 on page 48 reveals two main differences: 1) At the PHY layer, Hypertransport does not require framing, channel coding, clock recovery due to the synchronous interface[1], and 2) in non-retry mode[2], there is no acknowledge/not-acknowledge protocol at the link layer, and a periodic CRC inserted every 512 transferred bytes is used.

The Click implementation of an interface for a single-link end device is shown in Figure 3.7. For the model, we partition the Hypertransport protocol logically among our protocol layers as defined in Section 3.1.2, although layers are not used by the specification. Due to the absence of an ack/nack protocol, only four paths through the interface are important:

A) Outbound transactions. Similar to PCI-Express, a transaction is written into one of the transaction buffer queues (posted, non-posted, response). Transactions are forwarded depending on the priority and the availability of receiver space (flow control). When a transaction leaves, flow control counters

[1]Rev. 3.0a [79] adds an asynchronous mode (Gen3) using scrambling and 8b/10b coding.
[2]In retry mode [77] the link layer uses per-packet CRCs and an Ack/Nack protocol.

are updated. The link layer performs the periodic CRC. The CRC value is inserted into the packet stream every 512 bytes. If necessary, the data link layer would interleave the current outgoing transaction with control packets, e.g. for flow control. If there are neither waiting transactions nor control packets, idle control packets are issued to the PHY layer. At the PHY layer, the packet is serialized and distributed according to the link width.
B) Inbound transactions. The PHY layer de-serializes the stream of incoming data into 4 byte fragments, colors them as control or data, and sends them to the link layer. At the link layer, the CRC check is performed and the fragments are assembled to packets. After a classification step, transaction packets are passed on to the next layer. At the TA layer the address check is performed that discards invalid transactions. Valid transactions are stored in the appropriate receive buffer queue, and the network processor core is notified. When the transaction is pulled from the queue, the receive flow control counters are updated and the transfer is completed.

E) Outbound flow control packets. The link layer issues NOP packets that include flow control information provided by the transaction layer. In Hypertransport, only the counter differences (max. 2bit per flow counter) are transferred. During initialization, multiple packets are therefore necessary to transfer the absolute value of the receive buffer.
F) Inbound flow control packets. Inbound flow control information is forwarded by the receiving link layer to the transaction layer. The transaction layer updates its transmit flow control counters and schedules the next packet for transmission from the pending transaction buffer.

3.3.4 Gigabit Ethernet model

Gigabit Ethernet (IEEE 802.3) [83] is a serial packet-oriented data transfer protocol for local area networks. Increasingly, it can be found as backplane in communication systems [174]. The IEEE 802.3ap task force currently discusses the standardization. Traditionally, Ethernet has been used in half-duplex mode with a shared medium and collision detection. Like the preceding protocols it can be used in switched point-to-point settings in full-duplex mode as well.

The Click implementation of a Gigabit Ethernet controller is shown in Figure 3.8. The function is partitioned logically among our protocol layers as defined in Section 3.1.2. Our TA and DLL layers split the function of the Ethernet standard's MAC layer. The PHY layer classification is identical to the standard. The 1000Base-X PHY shown in the figure requires 8b/10b coding similar to PCI Express and RapidIO.

In Figure 3.8, the black portion of the graph is used in full-duplex mode. The additional red processing paths (its elements are marked by a *) are required for half duplex. Both modes must be supported since the mode can be negotiated dynamically. There is no flow control in half-duplex mode; in duplex mode the framer supports burst transfers, collision handling, and retransmissions.

Due to the absence of an ack/nack protocol, only four paths through the interface are important:

A) Outbound transactions. The SoC core stores outgoing frames in the input queue. As soon as the link is idle, the frame is read and stored in the framer while an FCS is added. The framer adds pre- and postfixes (SOP, EOP,

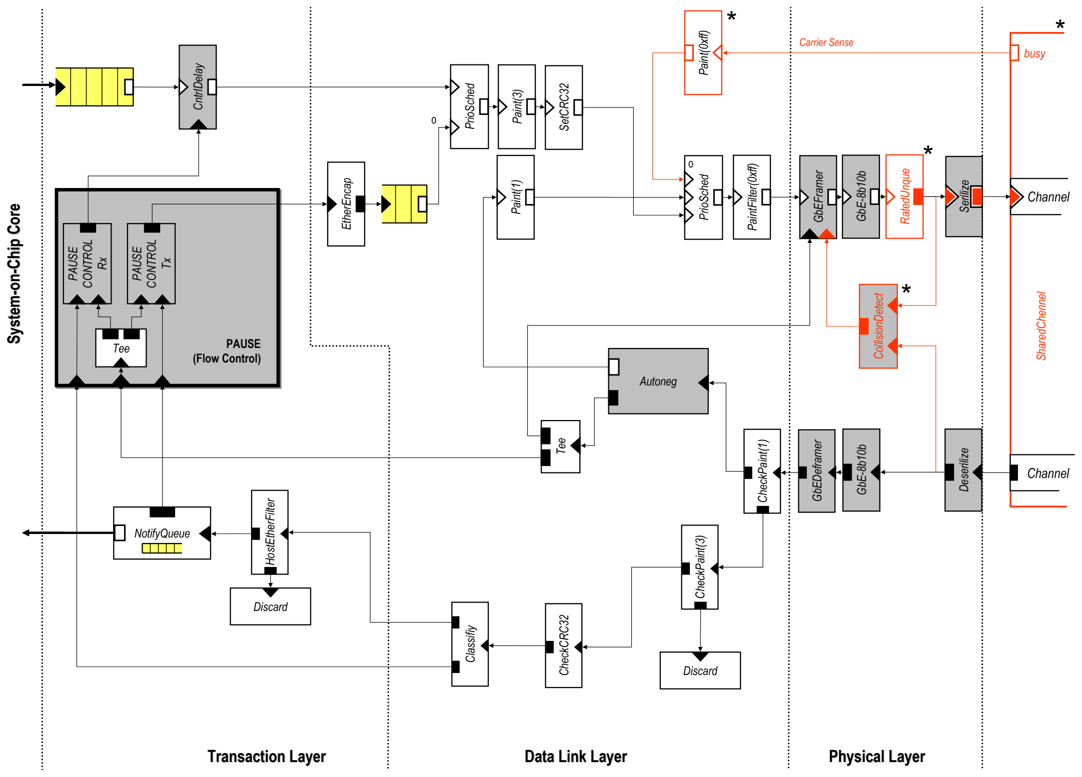


Figure 3.8: Gigabit Ethernet MAC and PHY (1000Base-X) model.

extension symbols) and breaks the packet into a stream of symbols that are sent over the link. After the packet, the framer generates a sequence of idle symbols to ensure the required inter-frame gap of 12 bytes. In case of a collision (half-duplex, collision detect), the framer interrupts the current packet transfer, sends JAM symbols, backs off, and retransmits the stored packet (or discard it if retransmitted too often).

B) Inbound transactions. Inbound streams of symbols are received, decoded, and assembled to frames at the PHY layer. At the DLL layer, the frames are classified. Configuration frames are sent to the Autoneg element. Frames without PHY errors and valid CRC are forwarded to the transaction layer and stored in the output queue (NotifyQueue), given their destination address matches.

E) Outbound flow control packets. When particular full or empty thresholds are reached, the output queue notifies the flow control unit (PauseControlTx), which generates PAUSE or CONTINUE frames depending on the type of message (almost full/empty). These specially typed ethernet frames have a higher priority than other frames and are transmitted as soon as the link is idle.

F) Inbound flow control packets. At the receiver's transaction layer, the flow control frames are fed into the flow control unit (PAUSE CONTROL RX). This unit extracts the delay value from the payload and sets the timer of the ControlledDelay element to delay the transfer of the next transaction frame accordingly. CONTINUE frames actually cancel the timed delay to improve performance.

3.3.5 Wireless LAN model

Wireless LAN (IEEE 802.11) [49, 36] is a packet-oriented protocol that connects stations to Ethernet networks using radio waves. Stations are usually mobile. A central device, the access point, manages the wireless network and performs the wired-to-wireless bridging. Communication over the air is quite different compared to shared medias in wired networks. Air channels are open to interception, highly unreliable, and dynamically change their properties due to effects such as mobility and multi path interference. The wLAN protocol has to cope with this channel properties at the price of higher protocol complexity. For this reason a wireless MAC extends and adds features compared to the protocols discussed before:

- **Virtual carrier sensing and collision avoidance** — In order to determine if the medium is available, virtual carrier sensing is used. 802.11 frames carry a duration field, the Network Allocation Vector (NAV), that is used to reserve the medium for a time period. Stations set the NAV to the time they will be using the medium. Other stations receive this value and set their NAV timers accordingly. A station knows the medium is busy if its NAV counter is not zero and can delay its own transmission. To avoid collisions caused by hidden nodes stations may use Request-to-send/Clear-to-send (RTS/CTS) control frame handshakes with extended NAV values prior to the actual data transmission (cf. Fig. 3.10).

- **Interframe spacing for priorities and contention resolution** — Access to the air is controlled by a set of coordination functions such as the one for contention-based service (DCF). They use varying interframe spaces to provide prioritized access. The higher the priority of the traffic the shorter the interframe gap. Four spaces are defined: SIFS (shortest) for highest priority transmissions (e.g. ACK, RTS/CTS), PIFS for contention free transmissions, DIFS for contention-based transmissions, and EIFS for recovery from erroneous transmissions[3]. Contention resolution is achieved by means of a backoff window that follows the DIFS interval.

- **Periodic exchange of network state** — Wireless networks require the exchange of network state and management information at regular intervals, e.g. to identify the network and to announce its capabilities. This is achieved by beacons sent by the access point. Similar to the negotiation of the Ethernet link speed, stations can explicitly exchange capability information such as a list of available transfer rates by sending a ProbeRequest. The access point responds if the parameter sets are compatible. From the negotiated set of transfer rates, for instance, the transmission rate of a packet may be selected dynamically, depending on feedback on the qualities of previous transfers.

- **Extra processing functions** — Two essential functions, which would be left to higher layers in other protocols, are integrated into wLAN: fragmentation and encryption. Fragmentation of transaction packets and management frames into a burst of small chunks increases reliability. Encryption methods defined at the MAC layer enable secure authentication,

[3]11n uses another access time, RIFS, which is shorter than SIFS and is used in burst mode

key management, and private communication usually at the expense of considerable computational effort.

For this analysis, the Click model focuses on the MAC layer (our transaction and link layers) as specified by the 802.11a/b/g standard. The air model receives complete frames and their transfer rate and generates the desired transmission delay and half-duplex/busy behavior of the MAC/PHY interface.

Figure 3.9 shows the main processing paths of the model. Wireless LAN does not implement flow control but adds additional control and management frame processing path for channel reservation and the permanent exchange of state:

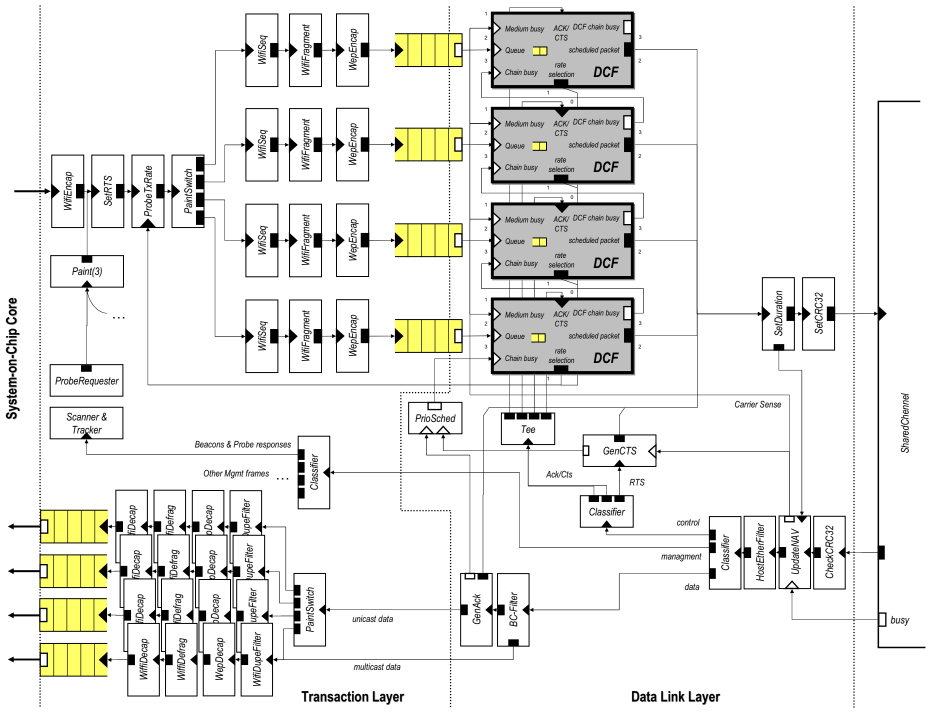


Figure 3.9: Click model of a wLAN 11n a/b/g station. The model includes service differentiation as specified by the 11e standard. Not all management processing paths are shown.

A) Outbound transactions. Ethernet framed transactions from the SoC core experience several processing steps before they are stored in a QoS queue: the Ethernet header is replaced by a wireless header, a sequence number is added, and, if necessary, the frame is fragmented and encrypted. Also, the transfer rate is selected based on the channel feedback for this connection. The DCF module, which handles the timed access to the medium, reads the transaction from the queue as soon as the link state permits. This means, that the frame is read if the medium has been idle for at least DIFS time, no retransmission or frames of higher priority are waiting, and no other station has won the contention. In this case, the duration field is set, the FCS is appended, and the frame plus its sideband information are sent to the PHY layer.

B) Inbound transactions. Inbound wireless frames are received and forwarded to the link layer. The link layer checks the CRC, updates its NAV timer with the frame's duration value, discards frames addressed to other stations, and classifies the frames left into data, control, and management frames. Data frames are forwarded to the transaction layer which separates them by QoS class, removes duplicates from the stream, decrypts and reassembles if necessary, and replaces the wLAN header with an Ethernet header. Then, they are stored in the output queue.

C) Outbound acknowledge. Valid unicast data frames that were accepted by the DCF element schedule the generation of an acknowledge control frame. As soon as the SIFS timer expires, the frame is pushed into the regular processing path for outbound transactions (see path A), i.e. its duration field is set and the FCS is added before it is sent to the PHY layer. In the model, acknowledges are always transmitted at the rate the data frame was received.
D) Inbound acknowledge. Inbound acknowledges experience the same processing path as inbound transaction frames (see case B) until they are classified and forwarded to the DCF. Here, retry and back-off counters are reset if the link expected the acknowledge. There is a timeout for not received acks.
G) Outbound RTS/CTS. If an outbound transaction frame is larger than a threshold (SetRTS), the generation of an outbound RTS frame is triggered as soon as the frame is read by the DCF element (cf. case A). The generation of CTS frames is triggered by reception of RTS. Similar to outbound acknowledges, RTS and CTS frames just follow the regular processing path of case A, after DIFS or SIFS time, respectively.
H) Inbound RTS/CTS. Inbound RTS and CTS frames travel the processing path F to the DCF. A RTS triggers the generation of the associated CTS (cf. path G). A received CTS frame triggers the transmission of an outbound transaction after SIFS time.

I) Outbound management frames. Outbound management frames follow the processing path of outbound transaction (path A) except that they do not need the wireless header encapsulation step. In the model, the generation of selected frames, e.g., a ProbeRequest, is triggered by the SoC core.
J) Inbound management frames. Received management frames travel up to the transaction layer similar to inbound transactions (path B). Here they are classified and either handled directly or forwarded to the SoC core. Received beacons, for instance, are fed into BeaconScanner/Tracker elements in order to monitor and update the interface's network state.

Protocol timing constraints

Different frames and the coordination of shared medium access have to follow strict timing rules. In Figure 3.10, an atomic data frame transfer including an optional medium reservation sequence is shown. In this example, station STA has gained access to the air and sends a reservation request (RTS frame) to the access point (AP). The AP has to reply after a defined time interval (SIFS period) with a CTS control frame. After reception of CTS, the STA starts the data transfer after another SIFS period. At the end, the AP acknowledges the successful reception of the data with an ACK control frame, again after one SIFS period. This sequence forms an atomic transfer. The short SIFS period prevents other stations from interrupting the transfer. After recognizing the free medium they must wait considerably longer (DIFS period and random backoff) before they may transmit.

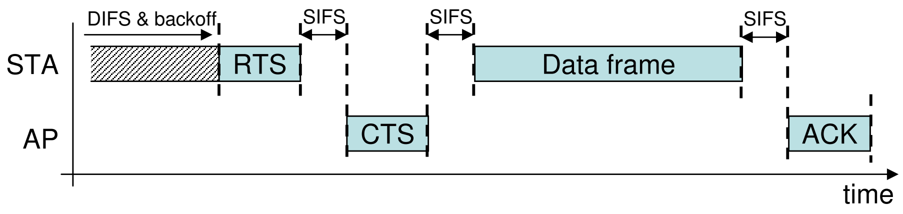


Figure 3.10: Atomic data frame transfer (IEEE 802.11).

All these strict timing requirements dealing with inter-frame spaces, response times, and transmission opportunities are in the low μs range. As a result, the cycle budget for the required frame processing is tight, as will also be pointed out later in Section 5.1.3.

3.4 Related work

Related work for this chapter can be found in two domains: 1) the comparison of communication protocols and their I/O interfaces, and 2) modeling and analysis of packet processing applications. References to the individual standards were already provided in Section 3.3.

3.4.1 Comparing protocols

The rising interest in new interconnect standards has been covered by several articles that provide overviews on changing sets of standards [16, 38, 46, 78, 125, 148, 163, 134, 162, 181].

- [38, 46, 148, 181] are introductional and do not conclude on similarities.
- A thorough survey of I/O adapters and network technologies in general is presented in [162]. The paper particularly focuses on requirements for server I/O.
- Selected issues of the physical layers of PEX/ASI, RIO, and 10Gb/s Ethernet are discussed by Noel et al. in [134].
- In [163], Reinemo et al. discuss the quality-of-service features of three protocols Infiniband, Ethernet, and PEX/ASI. They conclude that each of the protocols may have its advantages, depending on the use case.
- Although in [16] and partly in [78] some details of HT, PEX, and RIO are compared, a comprehensive picture has not been drawn of the essential elements and processing paths of the interfaces.

We are interested in the similarities and dissimilarities of the higher level interface and protocol aspects, i.e. the transaction and link layers of our protocols.

The different physical characteristics of the protocols are already addressed by initiatives such as the Unified 10Gbps Physical-Layer Initiative (UXPi) [134] and individual companies such as Rambus, [125]. An early version of our work [169] compared the interfaces for PCI Express, RapidIO, and Hypertransport. The current version adds Ethernet and wireless LAN to the comparison and extends the modeling concepts to shared medias and precise timing.

The article by Bees and Holden [16] comparing Hypertransport, PCI Express, and RapidIO complements our analysis.

3.4.2 Modeling packet processing applications

There is a large body of work that uses network and packet processing models for many purposes such as the analysis of protocols or the evaluation of an application's performance for a given traffic and network setup.

Given the diverse purposes, modeling approaches for packet processing applications fall into two broad categories: analytical methods and simulation based techniques. Analytical methods such as network calculus [110] usually abstract the system function. They are used, e.g. in [195], to reason about selected properties of the network system or to conduct static/worst-case performance analyses for generalized traffic scenarios. Simulation-based techniques on the other hand rely on characteristic workloads but may capture dynamic system effects as well. Depending on the abstraction level, they also capture the actual system function.

For the functional comparison of our interfaces, we are focusing on simulation-based approaches since they enable the verification of the models system functions. This is one of our six requirements that are listed and discussed next.

Requirements

For the functional comparison, the models must fulfil six requirements:

- Interface device focus — We are focusing on the communication protocol interface and its surroundings, meaning the protocol endpoint that sends/receives data, not the communication network itself.

- Full system function — For the functional analysis, our models must capture the flow of data with its processing functions and the flow of control including the state machines of the communication interface, correctly.

- Executable model — An executable model enables verification and validation of the system function, e.g. by simulation or emulation.

- Architecture independent — For the comparison, we are interested in the function only. No assumptions whatsoever must be made in the model about the platform the system will be running on.

- Component-based model — To expose concurrency and dependencies within a protocol function, the models should be composed from components. A library of elements facilitates reuse among protocols.

- Timed models — Communication interface are reactive systems that need to meet real-time requirements. An explicit and metric notion of time is required to model time-outs, deadlines, and rates in the system.

These requirements are driven by the goals of the interface analysis. However, we want to reuse the device models as input for the architecture exploration. The models should serve as golden units and hopefully provide a path to the evaluation of an embedded platform as well.

Models of Computation

Models of Computation (MoC) are an established approach to capture the behavior of concurrent and heterogeneous systems for the specification and design of embedded systems [24, 89]. MoC basically are design patterns that 1) separate computation, communication, and control, 2) describe the system behavior as interaction of components, 3) guarantee desired system properties, and 4) can be executed.

A number of MoCs are relevant for describing packet processing applications and are therefore discussed here:

- Process Models — Components in process models are sequential processes that run concurrently and communicate via messages either synchronously (CSP – Communicating Sequential Processes [71]) or via FIFO channels with blocking read (Kahn process networks [93], extended in [141] to bound fifo sizes by using blocking writes). CSPs fit particulary well to resource sharing and client/server applications. In [182] for instance, they are used to study intrusion detection in a TCP network. Kahn process networks are used, e.g. in [191], to model streaming signal processing applications. Stream processing exhibits strong similarities to the flow of packets through our communication interfaces. However, process models are untimed.

- Petri Nets— Petri nets are directed graphs of nodes connected by edges. Two different types of graph nodes exist: *places* and *transitions.* Places represent state. They store *tokens.* Transitions represent actions of the system. The modify state by consuming and producing tokens depending on enabling and firing rules. Places are always connected to transitions and vice versa.
  Petri nets are a graphical formalism to express the concurrency of systems. They are analyzable and can be used to prove certain properties of the modeled system. A review of the fundamental concepts is given in [131]. High-level Petri nets extend the fundamental place/transition systems. Coloured Petri nets [91] make tokens distinguishable to enable data-dependent modeling. Time and timed Petri nets add notions of time to the system (see references in [131, 91]). The work in [88] adds hierarchy and parameters to enable component-based modeling techniques.
  Petri nets are powerful enough to model most of the other MoCs such as the Kahn process networks above [114]. High-level Petri nets have been applied successfully to many domains [131], including performance modeling and the analysis of communication protocols. They are suited to model and simulate our communication devices. However, there expressiveness is not required for the interface comparison. Also, functionally correct models that capture control and dataflow grow complex, quickly.

- Finite State Machines — Finite state machines (FSMs) model the behavior of control-dominated reactive systems by composing events, a finite number of states, and actions. A FSM sequentially responds to external events with actions. An action may depend on the current state and may generate outputs and transitions of the state.
  Hierarchical finite state machines, as first introduced by the Statecharts model by Harel [66], enable the concurrent and hierarchical composition of FSMs by defining *and* states (system is in state a and b) and *or* states (system is in state a or b). This handles the problem of exploding state spaces for larger systems and significantly increases usability. In general, hierarchical finite state machines are an useful abstraction for describing the control facet of communication protocols. They can be executed, and there is a clear path to hardware and software implementation. However, they cannot express data flow and processing well. For this reason, FSMs are often combined with other computation models in hybrid models. The **charts* study by Girault et al. [52], for instance, discusses the combination of hierarchical FSMs with, e.g., process networks, to capture the dataflow and computational facets of applications as well. The Click model of computation encapsulates state and state machines within individual elements. In addition, our packet-flow oriented interface models use state tokens that flow between elements to explicitly achieve FSM-like behavior.

- Queueing Networks — Queueing networks are composed of systems of queues and servers [99]. Queues store transactions until a server becomes available. The server creates, processes, or deletes a transaction. This consumes a specific amount of time after which the transaction may be forwarded to the next queue. Service time and route can depend on the class of a transaction [15]. Queues store transactions according to their queuing discipline. With first-in-first-out (FIFO) queues and servers with processing function, queuing networks become similar to (timed) Kahn process networks.
  Using queuing networks one can determine, e. g., the residence time of transactions in the system, the average queue length, and the server utilization. In [197, 139], for example, queuing networks are used to model the distributed coordination function of the IEEE standard. The authors reason about delays and queue lengths in the system, using analytical results that were verified by discrete event simulation. Simulation frameworks for queueing networks include the SimEvents [123] extension of Simulink and SystemQ [189]. For capturing the system function of our communication protocol interfaces, queueing networks are less suited. The strict queue/server composition scheme limits the expressiveness of the model and restricts its granularity.

- Discrete Event — In discrete event models, components communicate via events that are associated with a time stamp. All events in the system can be ordered following a global time. A component may generate events with a future time stamp but can receive only those available at the current time. Discrete-event is a quite general model of computation and frequently used in general purpose simulators for applications such as network simulation.

Today, there exist a number of established network simulators. Prominent representatives[4] with large user communities include ns-2 [23], OpNet [138], and OMNeT [201]. Network simulators are usually targeted at a range of protocols. Models are composed hierarchically from library elements, often in a special specification language. Typically, the library is implemented using a regular programming language such as C++ or Java. The focus of network simulation is on simulating communicating nodes and observing the dynamic protocol behavior within the network, e.g., the TCP feedback mechanism or the effects of hidden nodes in wireless networks. Little attention is paid to the details of individual network devices. To speed up simulation performance, nodes are significantly abstracted. This is illustrated by a study on the effects of detail in wireless network simulation [69]. The most detailed cases in this study use a black box cost model of the MAC layer for analyzing the energy consumption of the network.

Each MoC has its own strengths and weaknesses. In fact, general purpose modeling and simulation frameworks often combine multiple MoCs for hybrid system models [156, 143] or provide extensions to support other MoCs on top of their built-in one (cf. Simulink's StateFlow and SimEvent modules). However, they all provide only little support for our communication interfaces, in terms of domain-specific primitives and data types.

System-level modeling languages

System-level modeling languages are aimed at models that address two essential needs [89]: firstly, the need for a specification that captures the system's function and requirements in a precise, unambiguous and analyzable way, especially during the early design phases of a system, and secondly, the need for a system description that provides a path to an efficient implementation without overconstraining the solution.

According to Edwards et al., the distinction between a language and the underlying model of computation is important [45]. The authors point out that a model of computation can be supported by different languages. Its MoC effects the expressiveness of a language. A language can be too expressive so that many analysis and synthesis problems become practically unsolvable. A language also can be incomplete or abstract. A coordination language as used in some of the network simulators, for instance, only describes the interaction-of-components facet of a model of computation but not the computation contained within the components. Instead, it provides an interface to another language for the components.

In the context of packet processing applications, two modeling languages seem most relevant [112, 157]:

- SDL [176] — The Specification and Description Language characterizes communication protocols by control sequences and I/O actions [17]. The most recent standard [86] provides a graphical representation and an action language. SDL's underlying models of communication are extended

[4]For other examples see the references in [23, 32].

FSMs (a flavor of FSM that folds state into variables), Kahn process networks of concurrent processes (with FSMs) that communicate via fifo signals of unbound size, and an imperative sequential model to be used in functions that manipulate variables and IO.
SDL's primary area of application is the formal protocol description, the verification of the protocol, e.g. in form of message sequence charts, and the (automated) generation of test patterns. The 802.11a wireless standard included an SDL description of the protocol function. In [108] this description is used for a performance analysis similar to the DE network simulation discussed earlier. Newer versions of 802.11, however, do not provide a formal description anymore, probably for reasons of complexity.
The language focuses on the control flow of an application, as the full implementation of TCP in [218] demonstrates. Dataflow and computation as required by our lower layer communication interfaces are not first class citizens, see the wLAN example in [65].
Attempts that use SDL for performance analysis note the lack of time-related features for expressing aspects such as timeouts, time-dependent enablers, or timing constraints imposed by the environment [54, 190].
There is quite some work on generating implementations (in hard- and software) from SDL descriptions, e.g., in [65, 119, 140, 186, 202]. However, the results so far indicate a lack of efficiency and report significant overhead and restrictions.

- UML [136] — The Unified Modeling Language is a general-purpose modeling language that provides structure and describes dependencies between elements but lacks formal semantics. So called profiles help to make UML applicable in a particular application domain by defining semantics for a subset of UML relevant to the domain. The use of UML for the multi-faceted design of embedded real-time systems is a vivid area of reasearch [109].

  There is ongoing work on making UML usable in the context of protocol specification and implementation [142, 129]. Pärssinen et al. describe a UML profile for communication protocols in [142]. Similar to SDL, the authors provide a textual language extension to describe actions in UML statechart transitions. This enables the model translation from UML to SDL by proprietary tools for a narrow protocol-specific subset of UML. There is also an ITU recommendation [129, 87] that specifies a profile to provide the same abstractions and semantics that can be found in SDL embedded into UML. Based on the UML/SDL recommendation, De Wet and Kritzinger [40] provide an approach for design and performance analysis. They also consider the UML real-time profile for expressing time related aspects.

There are many other network protocol languages (including Promela, LOTOS, Estelle, and others, see the references in [101]), which are of less interest to our functional comparison. According to [101], most of them focus on verification. Pragmatic goals such as real-world implementation or, as in our case, the simple comparison of the system function, are difficult to achieve.

Click: Model of computation and domain-specific language

As explained in Section 3.2, we use Click as a modeling framework for the comparison of our communication protocols. It is widely used for describing and implementing real-world packet-flow oriented applications that run as software on Linux systems. Click is both, a model of computation and a packet processing language.

A Click model describes the flow of packets through a system, see the Overview in Section 3.2.1. The Click MoC interprets the connection between elements as method bindings. In case of push connections, the thread of execution flows with the data, i.e. the upstream element calls a push method at the associated downstream element with the packet as parameter. In the case of pull connections, the downstream element triggers the generation of a packet by calling the associated pull method at the upstream element. In this case, the packet is the return value. This pattern results in a direct and efficient implementation. Lee and Neuendorffer [113] classify this as imperative MoC, since a command (the function call) is given. In contrast, in declarative MoCs, components just offer the data. When and how it is processed by the receiver is not relevant. Most of the other MoCs discussed before are declarative. Click's notion of time is explicit and metric. Packets in Click can carry a time stamp. Elements may use the time stamp and access the real time of the Linux host system to implement time-dependent behavior. This dependency on the host time can be compensated for our interface models (see Sec. 3.2.2). Many MoCs such as process models and FSMs are untimed.

The Click language is a graphical and textual coordination language. It is used to instantiate and configure elements and to describe the packet flow between them. Compound elements provide hierarchy within a model. The elements themselves come from a library and are implemented in C++. Click is sufficiently expressive for a natural and intuitive description. The focus, however, is on efficient implementation, not on formal aspects. In contrast, SDL or other FSM oriented languages have a somewhat 'unusual' programming model, and their models can be less intuitive and more difficult to understand. Also, since Click elements describe a particular packet processing task, encapsulate all relevant state, and have the simple packet passing interface, reuse and modularity of the elements are easy and natural.

3.5 Comparison and conclusion

Table 3.1 groups the elementary tasks as described in Section 3.1.3 according to their appearance at the different layers in RapidIO, Hypertransport, PCI-Express, Gigabit Ethernet, and wireless LAN. Table 3.2 shows the different packet processing paths as implemented in our interface models (cf. Sec. 3.3). The tables reveal quite a few similarities in the occurrence and deployment of tasks.

- The physical layer of RapidIO and PCI Express is almost identical. PCI just uses the additional scrambling for EMI reduction. Gigabit Ethernet and Hypertransport in its most recent version may deploy similar functions as well. There are, however, protocol-specific deviations in the implementation. The control characters of the 8b/10b coding scheme, for instance,

Table 3.1: Tasks on protocol layers for the different interfaces.

Function	RapidIO			PCI-Express			Hypertransport			GB Ethernet			wLAN		
	PHY	DLL	TA	PHY	DLL	TA	PHY	DLL	TA	PHY	DLL	TA	PHY	DLL	TA
Clock recovery	+			+			$+^4$			+			+		
Clock compensation	+			+			+			+			+		
Lane de-skewing[1]	(+)			(+)			(+)								
8b/10b coding	+			+			$+^4$			$+^5$					
Scrambling				+			$+^4$			$+^6$					
Striping[1]	(+)			(+)			(+)								
Framing	+			+						+			+		
Carrier Sensing										+			+	+	
CRC protection		+			+	$+^2$		$+^{3,7}$			+		+	+	
Ack/Nack protocol		+			+			$+^7$						+	
Classification		+	+		+	+		+	+		+			+	+
Packet assembly		+	+		+	+		+	+		+	+		+	+
Flow control			+			+			+			+			
Address validation			+			+			+			+		+	+
Buffers&Scheduling			+			+			+			+		+	+
Configuration Space			+			+			+	+				+	+

[1]Required only for multiple-lane links. [2]Optional end-to-end CRC for transactions.
[3]Periodic instead of per-packet CRC in non-retry mode. [4]Optional asynchronous operation mode.
[5]1000 Base-X (fiber),[6]1000 Base-T PHYs. [7]Per-packet CRC and Ack/Nack in optional retry mode.

Table 3.2: Packet processing paths for different IO interface models.

Path	Function	PEX	RIO	HT	GbE	wLAN
A	Outbound transactions	+	+	+	+	+
B	Inbound transactions	+	+	+	+	+
C	Acknowledge generation	+	+	(+)[1]	–	+
D	Acknowledge reception	+	+	(+)[1]	–	+
E	Outbound flow control	+	+	+	+	–
F	Inbound flow control	+	+	+	+	–
G	Outbound RTS or CTS	–	–	–	–	+
H	Inbound RTS or CTS	–	–	–	–	+
I	Outbound management	–	–	–	–	+
J	Inbound management	–	–	–	–	+

[1] Not supported by the modeled version of Hypertransport (1.04)

are interpreted differently. Wireless LAN uses different encoding schemes.

- RapidIO, Hypertransport, and PCI Express support multiple-lane links. Gigabit Ethernet and wireless LAN rely on higher layer link aggregation schemes.

- All protocols except Hypertransport require framing at the physical layer. The frame formats differ. Wireless LAN has the most complex one. It includes data fields and a 8bit hec value (+ mark in CRC row). Hypertransport uses extra control signal instead.

- Carrier sensing is used by all shared medium communication protocols, namely wireless LAN and GbE.

- All protocols use CRC functions for error protection. Length and polynomial may differ between protocols. CRCs of 32bit (PEX, GbE, HT, wLAN), 16bit (RIO, PEX), 8bit (wLAN), and 5bit (RIO) can be observed. Most protocols append the CRC value at the end of the header and/or payload. RapidIO allows for the insertion of an interim CRC value for the first 80 bytes of large packets in addition to the final one at the end of the packet. Ignoring packet boundaries in the non-retry mode, Hypertransport inserts its CRC periodically in the data stream.

- An acknowledge/not-acknowledge protocol is used by all standards except Ethernet. The protocol requires sequence numbers and a replay buffer for proper operation. Details differ among protocols, e.g., the implementation per QoS class or for all of them together.

- Flow control is used by all protocols except wireless LAN. GbE uses only a simple on/off scheme for the link. PEX and RIO deploy sophisticated credit-based schemes that distinguish QoS classes.

- All communication protocols use different packet formats depending on the purpose of the communication. This means that generic packet assembly and packet classification are common tasks across layers and protocols.

A further analysis of the application models, i.e., an architecture-independent profiling, reveals some implementation insights:

- *Data types.* In the models, most processing is done on header fields, 32-bit or less. Only Ethernet and wireless LAN addresses are 48-bit. Packet descriptors [32B] that contain addresses, packet length, packet/payload ID, flow ID, and so on are exchanged between tasks. The payload is rarely touched.

- *Primitive operations.* The models use frequent field matching operations that require bit masks and shifts in addition to primitive arithmetic and logic operations, e.g., for classification.

- *Concurrency.* The different processing paths are independent tasks that may execute in parallel. Full duplex interfaces require two truly concurrent processes for receive and transmit functions.

- *Activation patterns.* Packet arrivals (push), an idle channel (pull), and expired timers may trigger the execution of a processing chain. Most protocols are delay-insensitive, only wireless LAN has hard real time requirements.

The packet-oriented communication protocols exhibit similarities in the occurrence and deployment of the elementary tasks. Activation schemes and packet processing flows follow similar patterns. However, the implementation details of the same task may vary from interface to interface due to protocol specifics. This means that a programmable architecture actually is beneficial for the implementation of the different standards.

Chapter 4

A Methodology for Exploring the Programmable Interface Design Space

After analyzing and comparing the packet-oriented communication interfaces in the previous chapter, we will now look into a methodology for exploring a flexible interface solution based on a common and programmable platform.

Programmable platforms are concurrent multiprocessor architectures that enable software-based implementations of particular system functions. They were introduced for coping with the increased time-to-market pressures as well as with design and manufacturing costs [97]. To date, there exists no established method for the systematic exploration of a platform's design space. Platform development and deployment still remain an art.

In the following, we are developing a methodology that takes into account the importance of the application domain. A programmable interface platform has to trade off flexibility against cost and performance. Our design flow therefore should characterize the application domain as early as possible. With this input, the design space can be narrowed to one major design trajectory that starts with the most flexible solution and systematically refines the platform architecture to meet performance and cost constraints.

Ideally, the Click models described in the previous chapter will become an input into the flow. These models are the result of an elaborate domain analysis and represent the application domain as a set of functionally correct and architecture-independent specifications. A path from the specification to an implementation in hard- and software would enable the fast and systematic exploration of platform alternatives based on a quantitative performance feedback.

4.1 Requirements

The exploration of the programmable interface design space requires the fast and systematic exploration of platform alternatives based on quantitative performance feedback. Taking into account the interfaces specifics, a methodology must fulfil seven requirements, which are grouped together in three challenges.

1. Rapid generation of platform instances for exploration — Individual design points must be creatable and modifiable rapidly, without the need for time-consuming reimplementations to enable the exploration of larger design spaces. The derived requirements are:
    - Separation of application and platform architecture — During the exploration the set of applications, i.e. the function, remains constant while the platform architecture and the function-to-architecture mapping varies. Separation ensures an identical function and avoids reimplementation and verification efforts. For accuracy, dependencies of both facets must be captured by sufficiently expressive abstractions.
    - Support Click as application description — The Click models described in the previous chapter are functionally correct and architecture-independent. They represent a considerable investment and should be used as description of the system function.
    - Full platform evaluation — The interface design space may use any of the building blocks of a hard- and software platform. Vice versa, the interface functions may be mapped to any resource within a system.
2. Early system-level performance evaluation — The performance of individual design points must be evaluated quantitatively, considering the overall system function under various load conditions, e.g., to rank design alternatives accurately and to identify potential bottlenecks early in the design process. This requires:
    - Feedback from architecture timing to system function — Interfaces are reactive systems in which the architecture timing influences the system's function. This dependency must be captured correctly.
    - Higher level of abstraction — Models are required that abstract from unnecessary detail but expose performance-relevant system facets correctly, e.g., the arbitration of shared resources. Such abstractions reduce the modeling effort and increase evaluation performance.
3. Productive exploration and programming environment — The methodology, abstractions, and tools used for the exploration should be usable for the development and deployment of the final system. This requires:
    - A path to efficient implementations in hard- and software — In order to preserve the investment into the evaluation models, a path to an efficient implementation is required. By providing predictable and correct implementation results such a path enables a first-time-right strategy and increases confidence in abstraction and models.
    - An integral programming model — Modeling and programming abstractions are required that are intuitive for the designer and naturally integrate communication interfaces into the platform's overall programming model.

The next section discusses existing exploration methodologies and frameworks addressing these challenges.

4.2 Methodology fundamentals

This section will discuss the state-of-the-art in design space exploration. It starts with SystemC performance modeling as this language has gained momentum for the description of platforms at higher abstraction levels. Next, there are Click extensions aimed at targeting Click to specific architectures. Finally, the common Y-chart methodology is introduced and existing approaches are examined with respect to the requirements listed in the previous section.

4.2.1 SystemC modeling and performance evaluation

SystemC is widely used for platform modeling early in the design cycle since it unifies the modeling of hardware and software aspects and enables models on different abstraction levels [58, 130]. Several commercial frameworks are available which support SystemC models [37, 192, 124]. However, these frameworks follow top-down methodologies based on refinement and (behavioral) synthesis starting at the transaction-level (see, for instance, the flow described in [196]). There are no separate abstractions for application, architecture, and mapping.

The work in [206] describes the state-of-the-art of platform modeling in SystemC for performance feedback. Components can be represented at transaction and RT levels, and programmable parts can be abstracted up to instruction set simulators. The StepNP network processor platform [145] (cf. Sec. 4.2.2) is an illustrative example. However, the simulation of several architecture building blocks at these levels is too complex to be used for early exploration.

Trace-based simulation

A technique to speed up the simulation is the use of traces that abstract the computational part of the system function by timed sets of communication events, as, for instance, [105]. However, traces have two fundamental issues. First, they do not capture workload and data dependencies well. A trace captures the details of a system execution path for a particular workload. Hence, for event-driven systems with varying workloads, a large amount of traces needs to be generated for a useful analysis. This is why the work in [207] captures different traces at task-level granularity depending on the type and size of input data. Second, it is unclear how to model systems in which the architecture timing actually influences the system function such as in case of periodic tasks and time-outs.

Hostcode execution

For this reason, some frameworks avoid the relatively slow instruction set simulation by executing software tasks natively on the simulation host rather than by using traces [96, 21, 34]. Our framework SystemClick (see Sec. 4.7) applies the same principle. Kempf et al. [96] deploy an instrumentation technique to annotate existing code with fine-granular data on used operations and memory accesses gathered during compilation. Using a data base with primitives, execution and access latencies are derived for the simulation. Avoiding the data base, the timing information can be directly annotated at basic blocks during code generation and compilation [34]. Contrary to that, SystemClick's, performance information is derived by profiling and is dynamically annotated on a coarse granularity, leveraging the well-defined element boundaries of Click tasks.

The execution of native tasks, i.e., C/C++ functions, within SystemC simulation requires wrappers that provide a simulation context in terms of event handling, timing, and execution concurrency. The study by Blaurock [21] focuses on the incorporation of legacy code into SystemC models. Special SystemC channels are used which encapsulate tasks. To model timing, wait statements must be inserted into the legacy code. The scheduling of tasks depends on their connection by SystemC channels and is left to the SystemC scheduler. In [95] virtual processors are introduced, which execute tasks. These processors handle the timing of events, support preemptive scheduling based on statically assigned priorities, and model the context switching overhead. Cheung et al. [34] use a SystemC module to represent the processor and its operating system. The module interfaces with processes, provides services (such as event, mutex, and mailbox) and handles process scheduling and overhead for the system.

SystemClick achieves the native execution by the strict separation of function, timing, scheduling, and arbitration. Wrappers are used to encapsulate tasks and handle timed events. Resources arbitrate the execution of tasks, following an explicitly selected scheduling strategy. The overhead of context switches is handled by the task wrappers so that individual processing times can be considered.

4.2.2 Click extensions for specific architectures

One of our requirements in the platform exploration and deployment process is the support of Click. Therefore, this section examines how Click is targeted to specific platforms.

Click as software on concurrent processors

Click was originally implemented on Linux using C++ [102] and achieves considerable performance compared to the Linux protocol stack [20]. Recent extensions to Click include SMP-Click [33] and NP-Click [178]. In [33] a multi-threaded Linux implementation for general-purpose symmetrical multi-processing (SMP) hardware is described. In [178], Shah et al. show how Click, augmented with some abstracted architectural features, can be used as a programming model for the Intel IXP-1200 network processor. The approach uses a library of Click elements which are implemented in target specific code and thus lack sufficient portability. In Section 4.4, we will provide a way called CRACC to map Click software onto a range of embedded processor cores by generating c-code from the Click source and compiling it using a core's tool chains. In Section 4.6, CRACC will be used to program our NOVA platform.

Click for hardware elements

Click is also used to encapsulate hardware description language content. In the CLIFF work by Kulkarni et al. [103], Click elements are mapped to FPGAs while reusing standard RT level back-end tools. CRACC's code generator can be adapted to generate netlists for CRACC and CLIFF, given a Click input. By incorporating CLIFF, a comprehensive framework for hardware-software partitioning and implementation of packet processing applications based on standard deployment tools could have been provided. Instead, we chose to adapt our code generator to SystemC due to its wide deployment. Our framework enables early simulation-based performance evaluation for hardware-software partitioning and provides a path to implementation based on SystemC and CRACC.

Click as platform programming model

The StepNP network processor platform [145] is closely related to NOVA and CRACC. The platform is implemented in SystemC and relies on cycle-precise hardware models for performance feedback. For the program execution, instruction set simulators are wrapped. Communication is implemented based on OCP communication semantics on the transaction level [145] following NoC schemes [146]. StepNP can use Click for specifying the application [145]. The corresponding application development environment, MultiFlex [147], is based on C++ and provides common SMP and message passing programming abstractions. The implementation relies on hardware support for message passing, object management, and thread scheduling. Contrary to that, we rely on common ANSI-C compilers in our CRACC approach for efficiency reasons. Programming close to hardware using C is the most accepted form of application development due to tight design constraints on latency and code size. It is also often the only programming abstraction above assembler provided by the core manufacturer [122]. The NOVA platform provides support for message-passing in hardware. Object handling and task scheduling are implemented as CRACC firmware functions leveraging the Click MoC communication semantics for runtime efficiency. The StepNP/MultiFlex approach does not provide abstractions for platform and mapping specification as they are required for performance evaluations based on the Y-chart, which is introduced next.

4.2.3 Common Y-chart methodology

In order to assess the feasibility of packet-oriented communication interfaces on one common and flexible module implementation, alternatives must be explored. The Y-chart methodology shown in Figure 4.1 is a common scheme for this task [10, 98, 126, 144]. Application and architecture descriptions are

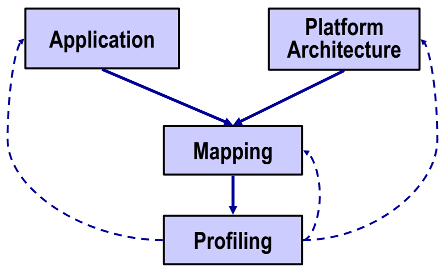


Figure 4.1: The Y-chart methodology for design space exploration.

independent and kept separate. Only an explicit mapping step leads to an analyzable (virtual) system prototype. A consecutive profiling and performance analysis step then provides feedback for optimizing the architecture, application, or mapping.

The distinct descriptions for application, architecture, and mapping separate the concerns of application developers and platform architects. Thus, each facet of the system can be specified using clean abstractions, models, and languages that are natural and productive for the facet. The platform may be modified without forcing the application to be rewritten and vice versa. Potential inconsistencies between architecture and application are exposed in the mapping step and can be addressed explicitly.

The Y-chart constitutes the core methodology of our application-driven flow for exploring a programmable interface architecture as will be presented in Section 4.3. Next, we will examine the state-of-the-art in design space exploration and platform modeling approaches.

4.2.4 Evaluation frameworks based on the Y-chart

There have been many attempts for the early exploration of embedded hardware/software platforms on the system-level.

With respect to their design flow they can be classified into three broad categories according to Mihal [126]: bottom-up, top-down, and meet-in-the-middle approaches. Mihal postulates the concurrency implementation gap between the application (top) and the architecture (bottom), to describe the mismatch between the capabilities of an architecture and the requirements of an application. Design methodologies, which map applications to architectures, aim at solving the gap by following one of the three strategies. Bottom-up methodologies provide architectural abstractions and models on top of the target architecture to ease and simplify the application-to-architecture mapping. Top-down methodologies start with application abstractions that help to model complex applications in a more disciplined and formal way, which enables a refinement path towards implementation. Meet-in-the-middle approaches use abstractions for application and architecture at the same time while solving the implementation gap. Mihal concludes that only the latter are sufficient for application-architecture-mappings since they provide multiple abstractions that can be used parallel to close the implementation gap.

Frameworks based on the Y-chart (cf. Sec. 4.2.3) are meet-in-the-middle approaches. In the following, a representative selection of these frameworks will be discussed. Overviews of additional (non Y-chart) frameworks can be found, e.g., in [55, 42]. Densmore [42] provides a comprehensive listing of frameworks while Gries [55] focuses on the discussion of design space exploration strategies.

Metropolis

Metropolis [11] is a general design framework, where all aspects of a design flow can be expressed at different levels of abstraction by using the meta modeling language. Function, architecture, and mapping facets are represented by separated entities. Applications are modeled as a set of processes that communicate via ports connected by networks of media. Constraints may be used to describe the coordination of processes and the behavior of networks. Architectures are represented by networks of media and processes, as well. In this case, quantity managers are used to arbitrate events and annotate costs. The mapping between application and architecture models is specified by a third network that synchronizes events between the application and the architecture. Its generality, however, makes the modeling and implementation effort less intuitive than by restricting the designer to one consistent representation for a particular application domain. The description of architecture resources and their scheduling policies, for instance, require some rather complex models that use multiple quantity managers and combine performance annotation with arbitration effects in a single description. Some of Metropolis' shortcomings may be addressed by the next generation of the framework [39].

Artemis

The Sesame framework [153] of the Artemis [155] project is targeted at the media processing domain. The application description is based on Kahn process networks (see Sec. 3.4.2). Architecture models are described in the same language as the application. Their implementation in SystemC is facilitated by an add-on-library which provides high-level communication abstractions and support for the XML-based modeling language. A mapping layer is used to associate processes and architectural resources manually. In addition, the layer bounds FIFO sizes and - as the framework is trace-based - virtual processors which feed application traces into the performance simulation [154]. The function of the virtual processors is similar to wrappers (cf. Sec 4.7.3) that interface application and architecture events while ensuring correct timing and access semantics. However, the Sesame framework does not provide a feedback path from architecture timing to the functional model. Thus, Sesame is not suited to model time-dependent behavior as required for our purposes.

PeaCE

The Peace framework [59] extends Ptolemy [156] (see Sec. 3.4.2) as co-design environment for SoCs targeted at multimedia applications. Applications and architectures are modeled using the communicating FSMs and extended SDF (shared variables, fractional rate) models of computation. For design space exploration, application tasks must by characterized for the respective architecture modules similar to a performance database (cf. Sec. 4.3). An automated mapping step uses a task graph, a set of hardware building blocks, and a performance/cost database. Assuming ideal communications, a task-node mapping is identified first fulfilling the design requirements. In a second step, bus-based communication architectures are explored based on the simulation and the analysis of memory traces that were derived for the particular task-node mapping. For the resulting application-architecture mapping, the performance is verified using timing accurate hard- and software co-simulation on the RT level. The authors argue with an exploration speed that is faster than the simulation-based evaluation. However, this trace-based approach cannot capture interdependencies between communication and computation and the timing feedback from architecture to the system's function, correctly. If speed permitted, e.g., by native task execution, we would prefer multiple iterations using a performance simulation with resource characteristics and contention. For the path to implementation, PeaCE uses a library-based approach for code generation and simulation, where efficient implementations come from a module library.

CoFluent

The CoFluent Framework [35] is similar to Artemis. Applications are modeled as sets of concurrent processes that communicate via fifo channels or shared variables. The platform architecture is modeled in SystemC and comprises processing elements with a VxWorks operating system which provides system function calls for communication and also handles the scheduling of tasks. An explicit and graphical mapping abstraction is used to derive implementations. For performance evaluation, the application code is instrumented and traces are derived [27]. The operating system aspect is abstracted to latency and explicit scheduling strategy [111].

Mescal

The Mescal project [127] focuses on the development and deployment of tiny heterogeneous application-specific processors (”SubRISCs”). SubRISC architectures are modeled bottom up from microcoded data paths following the TIPI approach [203]. The Cairn methodology [126] enables correct implementations for clusters of Tipi-elements by providing separate abstractions for architecture, mapping, and application. Applications are described in a specialized language using models of computation and actor-oriented design. Case studies demonstrate the approach for packet processing and signal processing applications. In the context of a programmable platform (cf. Sec. 4.3), the Cairn/Tipi approach can be used to explore flexible application-specific building blocks. Apart from the RTL code generation, the Tipi framework can generate fast simulators, which can be slaved in other simulation environments [204]. There is, however, no suitable compilation technology for TIPI PEs. Cairn/Tipi currently relies on manual mappings and assembly-level programming for the crafted processing elements. The Cairn/Tipi approach cannot be used for embedded standard cores.

Summary

Table 4.1 summarizes the properties of the different frameworks with respect to our requirements. Following the Y-chart methodology, all frameworks separate application and architecture specifications and have an explicit mapping step. Click as the application description is only supported by the Mescal approach, the others rely on non-domain-specific MoCs (Process networks, SDF, FSMs). Mescal, however, is limited to the exploration of Sub-Risc PEs. It does not support the full range of SoC platforms. The other frameworks have deficiencies in capturing performance-relevant system behavior. Support for time-dependent system behavior is limited (Metropolis), or does not exist (Artemis, CoFluent, Peace). Scheduling and arbitration effects, which have a large performance impact, can often not be modeled explicitly and the support for dynamic scheduling is limited. Some modeling abstractions are either too general (Metropolis) or too limited (Peace) for enabling intuitive and productive descriptions of design points. All frameworks provide a path to an implementation. Only Kahn-process networks (Artemis/CoFluent) may be used as a programming model for platforms, but storage and scheduling remains implicit.

In summary, none of these frameworks adequately addresses the requirements for the exploration of communication interfaces as listed in Section 4.1.

Table 4.1: Overview of Y-chart-based design space exploration frameworks

Requirement/ Feature	*Framework*				
	Metropolis	Artemis	Peace	CoFluent	Mescal
Processing Domain	GP	Media	Media	Media	Packet Signal
Separation of application and architecture	++	++	++	++	++
Click as application input	o	—	—	—	++
Full platform eval, including OS	o	++	o	++	—
Modeling abstraction	process networks	KPN/ TLM	FSM/SDF	KPN/ TLM	Click Tipi-PE
Feedback of platform timing into function	o	—	—	—	++
Path to implementation	o	++	++	++	++
Integral prog. model	—	o	—	o	—

++ supported, – not supported, o partially supported, GP - General Purpose

Furthermore, only the CoFluent/Artemis frameworks support SystemC, which has gained momentum as system-level modeling and simulation language.

4.3 Application-driven flow using the Y-chart

For the Y-chart, the application description and the platform specification are equal inputs. They are assumed to be independent of each other. However, a programmable interface platform must trade off flexibility against cost and performance. Our design flow is therefore to focus on characterizing the application domain as early as possible. With this detailed application knowledge, the platform design space can be narrowed quickly to a major design trajectory that starts with the most flexible solution and refines the platform architecture systematically to meet performance and cost constraints.

4.3.1 Five phase methodology

Our design methodology can be described by the following five phases. They can be seen as a realization of the Mescal methodology [56], where constraints on costs, flexibility, and ease of programmability form the tools used in each phase. Particularly "ease of programmability" plays an important role in our case as we are anticipating a programmable architecture based on embedded cores.

1. *Application Domain Analysis.* Identify and define representative and comparable system-level benchmarks and realistic workloads for packet-processing algorithms and protocols by analyzing the application domain.

The goal is to understand the application domain and to derive design requirements and system environments. We need to identify and specify system-level benchmarks, including functional model, traffic model, and environment specifications that enable a quantitative comparison of the architectures being considered. While kernel level benchmarks are simple and used most frequently, they can potentially hide the performance bottlenecks of a heterogeneous platform.

2. *Reference Application Development.* Implement a performance indicative system-level reference application that captures essential system functions.

For performance indicativeness and the evaluation of different partitioning and mapping decisions, executable and modular system-level reference applications are required to determine weight and interaction of function kernels.

An application reference is necessary to explore different partitioning and mapping decisions on the system level. Given the variety in parameters and protocols this reference application needs to be modular so that customizations can be achieved quickly. We model our applications in Click as described in Chapter 3. The reference models comprise the system function and a test bench that contains traffic sources and sinks. This test bench together with the configuration, i.e., the setup of routing tables, classifiers, etc., defines a particular environment the model is working in.

3. *Architecture-independent Profiling.* Derive architecture-independent application properties by analyzing and profiling the reference application.

Our approach emphasizes this step to narrow architectural choices. The goal of this step is to determine architecture-independent characteristics of the reference application by static analysis and/or simulation. Due to the complexity of modern designs it is infeasible to maintain several prototypes of the architecture. Before the actual development of the architecture starts, we are using this phase to find a good starting point for exploring the architecture design space.

For the communication interfaces, the analysis and simulation have been focusing on the features described in Section 3.5. These characteristics are derived by annotating the Click specification of our reference application and executing the model. Several conclusions can be drawn from the results to prune the design space. The affinity to certain hardware building blocks (general purpose cores, DSPs, ASICs, or FPGAs) was formalized in [175, 26]. Design decisions on the communication architecture can be drawn from the analysis of the traffic between components. For example, if a cluster of components shows a high volume of transfers between its members in comparison to the data transfer volume entering and leaving the cluster [151], this setup cannot exploit a high-performance shared global bus to connect to the rest of the system well. The storage requirements can guide decisions on memory hierarchy and data layout, e.g. whether packet payload is to be stored separately from headers and whether fast scratchpad memory is required. Finally, the application analysis reveals concurrency among the branches of the graph that can be exploited by corresponding hardware architectures such as multi-threading and parallel cores.

4. *Architecture Exploration.* Perform a systematic search of the architectural design space based on the reference application and its properties; define and iteratively refine the resulting platform architecture.

As a starting point for a programmable interface architecture, we use a set of embedded general-purpose processors and other commodity parts. Such a generic solution is most flexible and programmable but is likely to fail other optimization objectives. Single core profiles are used to identify optimization potential by refining the platform to meet further objectives, i.e., in terms of area. As a result, a virtual platform prototype with an efficient programming environment is implemented.

We start by determining an upper bound on the required computational resources. We use the reference applications to profile the combination of core and compiler for a range of embedded processors. We start with this most flexible and easiest to program solution due to our primary design criteria (flexibility and ease of programming). We first profile and compare individual processors. We then derive multiprocessor solutions and compare them with respect to their area requirements. We finally optimize and refine the platform to maintain as much flexibility as possible while optimizing the area, including the use of instruction set extensions and dedicated coprocessors. A prerequisite to this quantitative approach is an application that can be mapped to different architectures following the Y-chart of the previous section (4.2.3).

5. *Platform Implementation and Deployment.* Implement the platform and ease the deployment with efficient programming abstractions.

The virtual prototype proves the concept and serves as an executable specification for the actual platform hardware and software implementation. The programming abstractions used during the exploration are reused to ease the deployment of the platform. A successful deployment of a programmable platform can only be achieved if the programmer is relieved of understanding the architecture in full detail. The programmer is faced with the problem of manually coordinating the execution of software portions on different processing elements which must be optimized individually on the assembly level [104]. This burden frequently leads to suboptimal usage of hardware resources and is tedious.

4.3.2 Prerequisites and tools

The Click models described in Chapter 3 can be seen as the result of phase one and two of our methodology. They are the outcome of a domain analysis and represent the application domain as a set of functionally correct and architecture-independent specifications. This means that we chose the Click language and the Click framework for modeling our reference applications in the context of the Y-chart. The Click simulation engine is used to verify the applications.

The third phase of the flow, architecture-independent profiling supported the comparison of the different protocols and led to a fully programmable multi-processor platform as starting point for the design space exploration.

The exploration of the platform's design space in the fourth phase particulary depends on methods for transforming and mapping the Click applications quickly onto architecture variants. We use code generation techniques for this purpose. Our *CRACC* framework generates code for platforms based on embedded processors. In addition, a hard- and software platform is required onto which the application can be mapped. The platform should be modular to ease the modification and the refinement. This is important since the exploration is an iterative process, where the development of a design point is guided by the results of a previous step. We developed the heterogeneous *NOVA* platform for this purpose. Using *NOVA* on the RT level and the *CRACC* code generator, we can quickly optimize Click applications and explore mapping and partitioning alternatives in a cycle-precise and bit-true setting. Figure 4.2 shows the CRACC/NOVA Y-chart. Architecture exploration, however, is restricted due to the design effort required on the RT level.

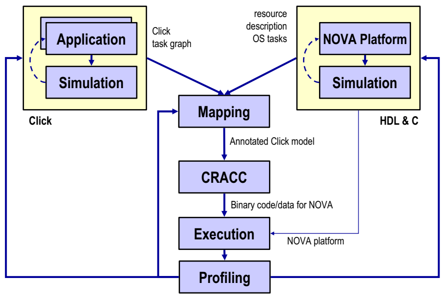


Figure 4.2: The Y-chart based on CRACC and NOVA.

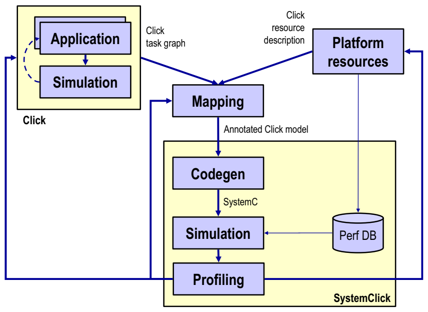


Figure 4.3: The Y-chart based on SystemClick performance models.

For iterative exploration early in the design process, a fast yet performance indicative way to evaluate platform variants is required. We therefore abstract a NOVA architecture as a set of shared computation and communication resources which arbitrate tasks and consume time. Our *SystemClick* framework generates a SystemC model from an annotated Click model that can be simulated using a performance database. Figure 4.3 shows the Y-chart based on SystemClick.

The database associates each task with the costs for a particular resource, e.g., its processing time or communication delay. For indicative results, this database can be populated with realistic data, e.g., derived from partial CRACC/NOVA execution profiles as described above.

The SystemC environment provides a path to implementation based on stepwise refinement and mixed-level simulations. Once the platform is implemented, Click and CRACC (cf. Fig. 4.2) become programming tools for its deployment.

The subsequent sections of this chapter describe and discuss the CRACC code generation, mapping, NOVA platform, and SystemClick performance models.

4.4 Code generation for embedded cores

We are looking for a way to map our Click models of packet-oriented communication interfaces onto a range of programmable architectures.

Application-specific architectures are notoriously difficult to program due to heterogeneity and concurrency. Current best practice in developing software under tight memory and runtime constraints is based on programming in assembler. Clearly, this approach is in conflict with other design criteria such as ease of maintainability and portability among platforms and architecture alternatives.

Due to the complexity of these systems, a more deterministic and reliable design flow is required, both in terms of development time and software quality (e.g., predictable performance). On the other hand, the design of reliable software in general is a vivid area of research [188]. The question arises whether we can bridge the efficiency gap to some extent between embedded software development on the one hand, and general software engineering concepts as embodied by Click on the other hand.

In this context we have to realize that programming in C is the only abstraction above assembly level that is established as a programming model for embedded processors [122]. Therefore, we have to provide a framework that supports a systematic path to efficient implementations based on C (or even assembler) while offering enough abstraction for software reuse, ease of mapping, and performance evaluation.

For this reason, we have developed a framework and code generator called CRACC (Click Rapidly Adapted to C Code) which enables the fast migration of packet processing functionality to different embedded processing cores based on a modular Click application description. This section describes CRACC.

4.4.1 Why not Click for embedded processors?

Click [102] is implemented in C++ and uses Linux OS concepts such as timers and schedulers. Most application-specific embedded processors, however, provide only limited programming environments, poor runtime support, and often do not support C++ [122]. Especially in the case of MPSoCs such as network processors we observe a lack of proper compilers and operating systems for processing elements within the data plane. Instead, they rely on assembler and as many static optimizations as possible in order to achieve fast and lean implementations.

Thus, we need to transform our Click application description into a representation an embedded processor can support, e.g., a 'C' program. Such a solution needs to address the following features, which Click offers:

- A domain-specific language for the hierarchical composition, configuration, and interaction of elements.
- Runtime re-configurability of element connections.
- Modular programming style using object-oriented features.
- Concurrency and scheduling of elements at runtime.
- Timers and timed rescheduling of elements.
- Communication interfaces between elements.

4.4.2 CRACC – Click Rapidly adapted to C Code

The CRACC design flow (see Figure 4.4) starts with an implementation in Click. The application programmer can model the functionality on any host on which Click can be simulated. After a functionally correct model of the application has been derived, which can usually be done quickly, CRACC's Click front-end is used to generate a netlist and the corresponding configurations for CRACC elements. The CRACC ANSI-C source code can then be cross-compiled and profiled on the respective embedded platform. The subsequent performance optimization can focus on individual elements and possibly on the partitioning of elements onto several processing cores. This systematic approach leads to improved design productivity and simplifies reuse.

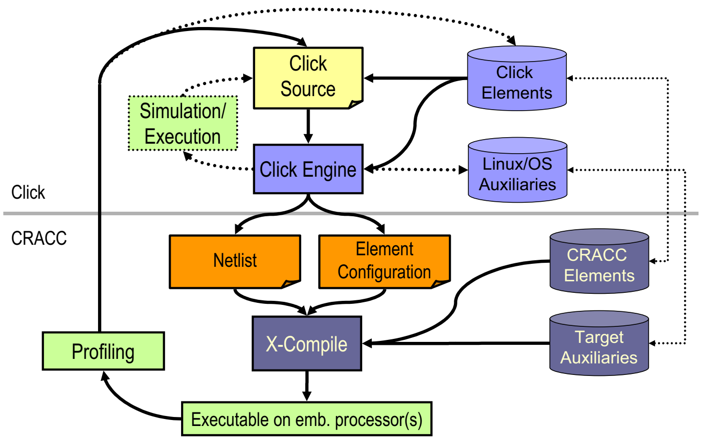


Figure 4.4: The CRACC code generation flow and framework.

Preserving Modularity

CRACC preserves Click's modularity by emulating object-oriented programming. Particularly objects and their dynamic management, inheritance, and virtual function calls are supported using function pointers in structs, e.g., described by Holub in [72]. This means that a Click element is represented in C by a struct that contains pointers to element functions in addition to its element data. That way, Click's push/pull function call semantics can be preserved. In the following, we will discuss these issues using our CRACC implementation.

- *An object in C* - In contrast to C++ object methods require an explicit pointer to the associated element. A struct is declared that can be used similarly to a class in C++. Before it can be used, however, the element's local data needs to be initialized, and the method function pointers need to be set to local methods.

- *Object construction* - Similar to C++'s constructor, we implement a function that creates an object struct, calls its init function, and returns a pointer to the struct. This encapsulates the element initialization and provides a convenient way of instantiation.

- *Inheritance* - Click uses a rather flat class hierarchy. Most elements are directly derived from the element class. In CRACC inheritance is achieved by using macros that declare an element's data and methods. The struct then simply contains an ordered list of all parent class macros followed by the local declaration. For proper initialization, the init functions of all parents followed by the local init must be called.

- *Virtual functions* - Using function pointers in structs actually makes every function virtual since it allows the replacement of an element function with another one even at runtime.

Packets and other data types

CRACC implements a *Packet* data type that allows the modification of its data. In Click, *Packet*s and *WriteablePacket*s are distinguished to preserve memory using a reference-or-copy-before-modification approach. Although more memory-consuming in some rare cases, the CRACC behavior seems more natural for embedded packet processing since we expect most packets (at least their headers) to be modified, anyway. The packet methods are called directly without using function pointers to avoid runtime overhead. Other Click data types that are often used to ease the element configuration such as classification rules and IPv6 addresses, are translated by the Click/CRACC front-end into ANSI-C data types.

Timers

In Click, timed and rated elements require a common time base for their operations. Click uses the operating system's (Linux) time base and exploits OS facilities for timed operations. Click implements two concepts for timed operations: tasks and timers. Tasks are used for operations that need to be called very frequently (tens of thousands of times per second) and therefore should execute fast. Infrequent operations are more efficiently supported by timers. Timed elements in CRACC can only access time by using timers. They are also used instead of tasks for 'fast' scheduling. Timers encapsulate the specifics of a target's implementation, e.g. a hardware timer that is register-mapped, memory mapped, or implemented as a co-processor.

Simulation engine

A DE simulation engine is provided by CRACC to enable the execution of the generated and compiled code for verification purposes. The engine provides a global time, maintains lists of scheduled events, and triggers the execution of push/pull chains. The scheduling order of the CRACC simulation differs from Click due to the different task and timer implementations. This means that the order of packets generated by different sources or pulled by different sinks will not necessarily be the same. However, packet flows generated by the same source/sink pair have an identical sequential order.

Preserving the Click language

The Click language describes the composition of a packet processing application from elements and its configuration. CRACC preserves this language front-end by extending Click. The modified Click first evaluates a Click source description and executes all initialization phases (configure and init functions). Then, it generates a C schematic of an application that instantiates, configures, and connects CRACC elements according to the Click description (see Figure 4.4 on page 64). Since the CRACC configuration is done after parsing and processing of the Click configuration strings, only preprocessed valid data structures are used for configuring the CRACC elements.

4.4.3 Targeting CRACC to embedded cores

CRACC has been targeted to a representative set of ten embedded processors, which among others includes the common 32bit cores MIPS, ARM, and PowerPC. The initial tool chains we used for the cores are based on GNU tools and public domain instruction set simulators, e.g., GDB. In a second phase, we switched to cycle precise simulators provided by the individual core vendors when necessary. For CRACC, the different endian-ness of the embedded cores was the issue with the largest impact on implementation effort and code base. The individual tool chains and programming environments furthermore required the adaptation of makefiles and runtime environments (e.g., startup and initialization code).

As an indication for the required resources, we analyzed the modeling effort for Click model, CRACC implementation, and the efficiency of the resulting code on selected processors using a typical network processor application with packet processing and quality-of-service functions modeled in Click [174]. The case study which has been published in [173], concludes:

- **Modeling effort.** Assuming that no elements must be added to the CRACC library, the software implementation of a full system model can be carried out within a few days. Pure Click descriptions are rather small and require only a few hundred lines of code. This is not surprising since the Click code only describes the composition and configuration of elements, the actual functionality is contained in the element library. For the DSLAM application domain only a few elements had to be added to the library.

- **Code efficiency.** The CRACC element implementations are much smaller than their Click counterparts due to the removed object-oriented overhead and simplified packet abstractions. On average, CRACC's object code size was only 17% of the Click objects's size (g++, gcc, -O3, x86 platform). This resulted in a compact software that required less then 20kB code memory in total.

Applying modularity as provided by Click and CRACC comes at the price of some overhead in runtime and code size. This overhead can be reduced significantly by a number of optimizations such as the static resolution of elements' processing types and the concatenation of elements belonging to the same task chain. The optimization potential is discussed in more detail in [173].

At this point, we have a method for mapping Click applications onto individual embedded cores. CRACC generates modular and portable ANSI-C code suited for a range of processors. Together with Click-based traffic generation, we can execute the code and profile the combination of compiler and individual core for the given reference application. The results are performance-indicative and can be used for benchmarking the different cores. In the next section, we will extend the CRACC concept to heterogeneous and concurrent platforms that deploy multiple processing nodes and tailored hardware.

4.5 Hardware encapsulation, platform mapping

Partitioning and mapping an application onto a heterogeneous and concurrent platform comprising multiple processing nodes and special purpose hardware still remains an art. To support and ease this manual task, we apply the idea of wrappers [217] to fit hardware and software resources together. The CRACC framework provides APIs, specialized library elements, and extended code generators that encapsulate hardware specifics and are responsible for interfacing hardware resources and synchronizing functional components that are mapped on different resources.

A Click application graph is partitioned at element boundaries and mapped onto a heterogeneous multi-processor platform manually, as indicated in Figure 4.5. In the following we will discuss techniques related to targeting and mapping Click/CRACC to a multiprocessor platform in more detail.

4.5.1 Packet I/O

Packet I/O in CRACC uses the same concepts as Click, i.e. From- and ToDevices that encapsulate interface specifics and handle the interaction with the device. A typical FromDevice, for instance, contains an ingress queue followed by an Unqueue mechanism that pushes received packets into the system. Depending on the underlying hardware, polling as well as interrupting schemes may be implemented and encapsulated. In the context of a multi-processor SoC, From- and ToDevices are mapped onto the physical I/O interfaces, e.g. a Gigabit Ethernet module. The communication with neighboring elements then

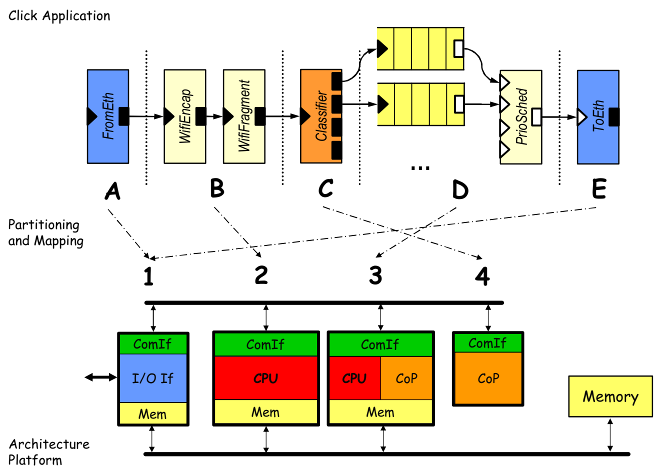


Figure 4.5: Mapping a Click application onto a concurrent platform.

follows the on-chip communication scheme as described in Section 4.5.3. The models of our packet-oriented communication interfaces can be seen as refinements of regular From- and ToDevices since they describe the device function in detail. From- and ToDevices can be replaced by artificial packet sources/sinks if our simulation engine is used.

4.5.2 Special purpose hardware

In order to ease the mapping of CRACC applications onto different platforms, special purpose hardware such as Timers and coprocessors is encapsulated in auxiliary modules that export a hardware-independent API. Depending on the way of coupling, special purpose hardware can be deployed either directly by using the API within CRACC element implementations or indirectly with particular Click/CRACC elements that are mapped onto the special purpose hardware. A register-mapped timer, for instance, requires only the Timer abstraction for its deployment. A loosely coupled CRC accelerator linked into the packet flow, on the other hand, is more naturally addressed by special SetCRC and CheckCRC element implementations. Element encapsulation requires loosely coupled hardware accelerators to support the communication interfaces described in the next section.

4.5.3 Inter-element communication

CRACC elements pass packet descriptors between elements. These descriptors contain extracted packet header and payload data, annotated flow information, processing state, and header and payload pointers. Three communication schemes can be distinguished depending on the mapping of the elements:

- On the same processor — Inter-element communication on the same processor is handled using function calls that pass a pointer to the packet descriptor. Elements on a processor require an task scheduler that fires all sources (sinks) of push (pull) chains. The scheduling of a push (pull) chain is non-preemptive. Its rate is implementation-specific and may depend on priorities. Elements on different processors use special sources (sinks) to communicate.

- Between different processors — The communication of elements on different processors requires message-passing semantics including a destination address. The address must allow the identification of both the target processor (in case of shared connections) and the associated task graph on that processor. For that purpose, special transmit and receive elements in combination with a FIFO queue are inferred that annotate the packet with the required information. The queue compensates potential rate differences that may be caused by scheduling and arbitration effects.

  Furthermore these elements convert push and pull chains as needed by the platform's communication pattern. The Figures 4.6 and 4.7 demonstrate the concept for a target system with push semantics. In this case, the pull chain conversion shown in Figure 4.7 requires user-annotated rate information for the transmitter. This is intended as the pull-to-push conversion changes the behavior of the model, a fact the programmer must be made aware of. Alternatively, a notification scheme can be implemented using a second push channel as shown in Figure 4.8. This solution matches Click's communication semantics exactly but is more expensive.

  Finally, these wrapper elements encapsulate the processor's hardware communication interface similar to other From- and ToDevices. We anticipate using specialized on-chip communication interfaces that provide ingress packet queues. These queues require only a small number of entries depending on the number of task graphs executed simultaneously on a processing node. However, the message passing semantics enables other implementations as well.

- Between elements in software and hardware — The interface of a cluster of hardware elements to the on-chip communication network must support the same packet passing semantics as for inter-processor communication, e.g. ingress queues. Depending on the interface between hardware elements, the hardware side of transmitters and receivers must convert the communication semantics accordingly. In the case of CLIFF hardware elements [103] and queue interfaces, for instance, a three-way handshake needs to be generated using full and empty signals of the queues.

  A special case is the pull access for reading the state of (tightly coupled) hardware as used by our Ethernet and wLAN models. In this case, the packet passing hardware interface can be avoided by a software wrapper which directly accesses the hardware and forms the tiny state packets expected by the model.

More on wrappers can be found in the later sections of this chapter. Section 4.6.2 discusses them in the context of the NOVA platform, Section 4.7.3 describes how they are used for SystemClick's performance simulation.

4.5.4 Data layout and distinct memories

As in Click, communication with memories is encapsulated within CRACC elements. CRACC assumes a unified access to the memory hierarchy. Besides

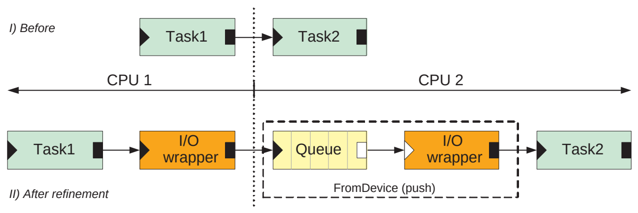


Figure 4.6: Synthesis of push communication wrappers for a push target.

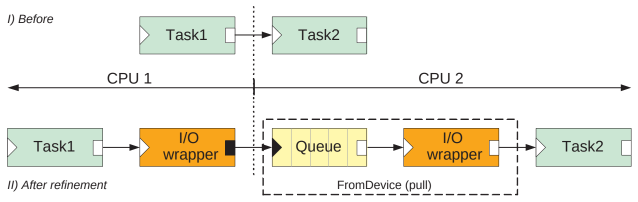


Figure 4.7: Synthesis of pull-to-push communication wrappers.

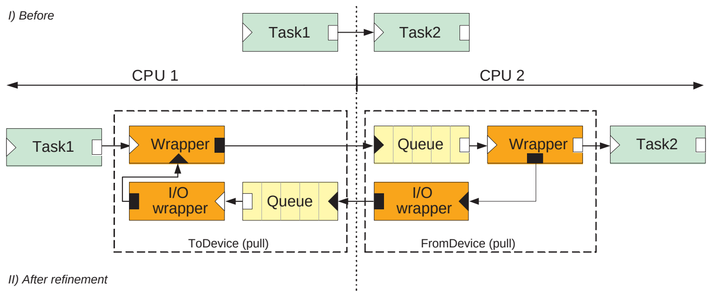


Figure 4.8: Notifying pull communication wrappers for a push target.

the local data memory, all accessible distributed/shared on- and off-chip memories are mapped into the processors memory address space. Data objects are either local or shared and are mapped to a particular memory explicitly by the programmer. In order to provide a hardware-independent solution shared data should be encapsulated into objects that define their access semantics (e.g. blocking/non-blocking) while deploying platform-specific hardware such as semaphore banks. Using such objects different data scopes as described in [178] can be implemented. In case of read-only data the encapsulation can be omitted.

4.5.5 Mapping annotation and multi core code generation

A particular application-to-architecture mapping is specified by the designer by annotating the Click application graph. For this purpose, specialized Click elements are provided which can be instantiated to guide the code generation process and help to describe platform and mapping specifics:

- StaticMapper — StaticMapper associate Click elements with platform resources by accepting a list of <instance name, resource id> pairs. In case of named communication (as implemented by SystemClick), communication links may be associated with communication resources as well. StaticMappers generate no extra code.

- FromIo/ToIo Wrapper — These communication wrappers partition a Click graph into subgraphs that can be mapped onto different resources. Depending on the code generator, they can be inferred automatically in cases where two communicating Click elements are mapped onto different resources. Alternatively, they may be used explicitly by the designer to partition, refine, and map the communication more efficiently, see Sec. 4.5.3. Communication wrappers generate platform-specific code.

- Resource — For mapping, platform resources are represented by resource ids. The Resource element is used to declare a resource and configure its properties. For most platforms, resource elements generate no code. Only in the case of SystemClick, specialized code for the performance simulation is generated.

- Resource manager — Resource managers represent platform-specific code that either initializes or manages a particular platform resource[1]. Memory managers, for instance, generate the software function (if mapped onto a SW resource) and the OS communication code required to address this function from the local as well as distant resources. Resource Manager must be instantiated explicitly.

Using these elements, the designer annotates the Click application source with mapping information and instantiates platform resources. To keep concerns separated, different source files may be used for application description, platform specification, and mapping annotation. From the annotated source, executable code for multiprocessor platforms is generated in several phases:

- Mapping propagation — CRACC allows partial mappings. In such cases, heuristics are used to derive a mapping for other elements automatically. Currently supported are 1) full downstream propagation, i.e., the initial element drags all connected elements onto the same resource, and 2) weak downstream propagation which stops at already mapped elements.

- Multi-pass code generation — The mapped task graph is iterated multiple times to ensure the generation of objects in the order of their (potential) dependencies. All generated C code is encapsulated in macros that enable conditional compilation depending on the resource id.

- Per-resource compilation and linkage — To compile the generated code for a particular resource, the associated resource id is defined and the resource's tool chain is invoked by scripts.

Depending on the platform, the generated per-resource code objects may be processed further, e.g., to compile and link code and data memory images that can be loaded into boot memories. The NOVA platform, described in the next section, for instance, requires a single image that combines the initialization and runtime code segments as well as the different data segments.

[1]In case of SystemClick performance simulation (cf. Sec. 4.7.4) they even *model* a resource.

4.6 NOVA – A Network-Optimized Versatile Architecture

In the previous sections of this chapter we dealt with concepts and tools to map Click applications onto programmable platforms. This present section focuses on the architecture platform facet of the Y-chart by introducing NOVA. NOVA, the Network-Optimized Versatile Architecture, is a programmable platform tailored for packet processing. We developed NOVA for the exploration of design alternatives in hardware and software, where the refinement process is driven by network and communication interface applications. A heterogeneous NOVA platform encapsulates embedded cores, tightly and loosely coupled coprocessors, on-chip memories, and I/O interfaces by special sockets that provide a common communication infrastructure.

4.6.1 NOVA hardware 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.

On-chip communication

NOVA supports two types of on-chip communication in hardware: direct message passing and memory-mapped 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 the means for blocking 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.

NOVA socket

The socket decouples the on-chip communication network from the processing nodes and provides unified interfaces between them. This, for instance, is helpful for the exploration of different communication schemes. Figure 4.9 shows the concept. The socket encapsulates an IP module and provides external interfaces. The figure shows three interfaces: to the packet descriptor (PD), the system message (SM), and the memory access networks. The internal interfaces are specialized for the particular IP module. Usually, NxM 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

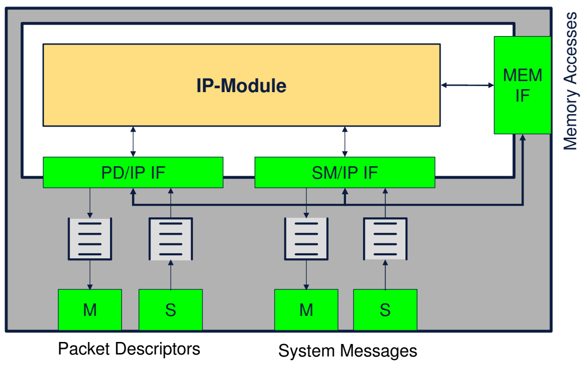


Figure 4.9: NOVA's socket concept.

Figure 4.9, 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, for instance, uses them to transfer packets to/from the memory.

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. Figure 4.10 shows a 32bit MIPS 4k PE with Havard architecture and a memory mapped-subsystem.

- 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 (encryption) and memory management (DMAs, memory manager).

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

- Off-chip interfaces – Off-chip interfaces are mostly off-the-shelf units encapsulated by NOVA sockets. The current emphasis is on network and memory IO. The fast Ethernet MAC wrapper, for instance, contains DMAs that autonomously store and forward packets, and parser/unparser units for the handling of packet descriptors.

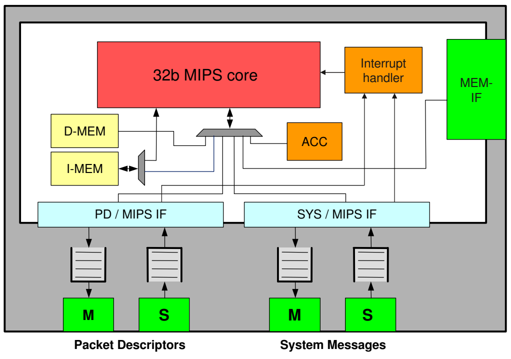


Figure 4.10: A 32b MIPS core and its subsystem encapsulated by NOVA's socket.

These building blocks can be connected using any on-chip communication protocol and arbitrary topologies. For the context of this work, we are using a communication infrastructure based on the OCP protocol (cf. Sec. 4.6.3). NOVA in the context of future interconnects and networks on chip has been discussed elsewhere [167].

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 in either hardware or software. Memories and all resources accessed by a PE are mapped into the individual PE's data memory map.

4.6.2 Programming model

We start programming 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.

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 as discussed in Section 4.5, 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 will be 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 are implemented for encapsulation of the 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 which 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.

Multi-core and OS extensions

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 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 yet 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 that 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.

4.6.3 Platform evaluation

We implemented a 4PE prototype of the NOVA platform. Figure 4.11 shows the block diagram. The device is dimensioned for use, e.g., as DSLAM line card processor and employs four of the MIPS processing elements (see Figure 4.10). Leveraging the FPGA-based Raptor2000 prototyping environment [94] the system implements four Fast Ethernet IOs that are connected to external PHYs and shared off-chip SRAM memory. The on-chip communication is based on three OCP buses for system messages, packet descriptors, and memories. This way, the high priority delivery of system messages is assured.

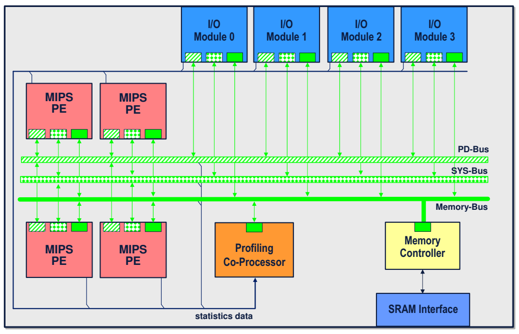


Figure 4.11: NOVA prototype with four processing elements.

The prototype also integrates a statistics and profiling module to derive run-time performance information. The module 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 runtime.

To evaluate 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. The ASIC results are provided and discussed later in Sections 5.3.3 and 5.4.

4.6.4 Remarks

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 (which will be presented in Section 5.4 and were published in [166]) show that the overhead of hardware and software modularity is reasonable for NOVA compared to state-of-the-art techniques and that NOVA is usable for the systematic application-driven design space exploration of network processors.

4.7 SystemClick – Annotated SystemC performance models

Using NOVA and the CRACC code generator, we can quickly optimize Click applications and explore mapping and partitioning alternatives. However, since this approach depends on the cycle-precise and bit-true execution of the generated code for its performance feedback, we are effectively limited to existing architectures. The systematic exploration of architecture variants is barely possible due to the design effort required for each platform instance.

In this chapter, we are therefore developing a way for enabling the early design space exploration by raising the level of abstraction for the architecture model and its performance simulation. For this purpose, we extend our code generation and mapping capabilities towards SystemC so that we are able to produce SystemC performance models of Click applications mapped onto architectural platforms. Figure 4.12 shows the concept of SystemClick [171], the modified CRACC flow for SystemC (cf. Fig 4.4 on page 64).

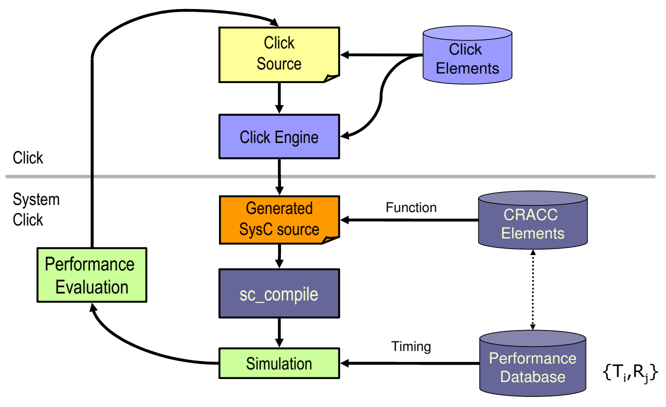


Figure 4.12: SystemClick — SystemC performance models generated from Click.

For fast evaluation, we abstract the architecture to a set of shared resources that arbitrate computation and communication tasks and consume service time. This is reasonable since these aspects alone already have a significant performance impact, see, e.g., [104]. The choice of SystemC, however, enables a path to step-wise refinement and mixed-level simulations as it would be required, for instance, for a virtual prototype.

4.7.1 Platform representation

A platform in SystemClick is represented as a set of resources and communication wrappers adhering to the mapping and encapsulation concepts of CRACC as described earlier in Section 4.5.

The hardware portion of the platform is modeled by resources that arbitrate computation and communication tasks. Computation resources process packets by executing Click task chains. Communication resources pass packets following Click's push and pull communication semantics. A task waits until the resource becomes available. Then it is executed and consumes service time. Wrappers extend Click task chains and represent the operating system aspects of a platform. Used on computation resources, they model the overhead required for event handling, task scheduling, and communication. Figure 4.13 shows the SystemClick platform representation that corresponds to the example used in Section 4.5 (cf. Fig. 4.5 on page 68). Each task chain of the partitioned graph is associated with a computation resource (shown in the upper part of the figure). Communication between tasks on different elements is associated with communication resources (in the lower part).

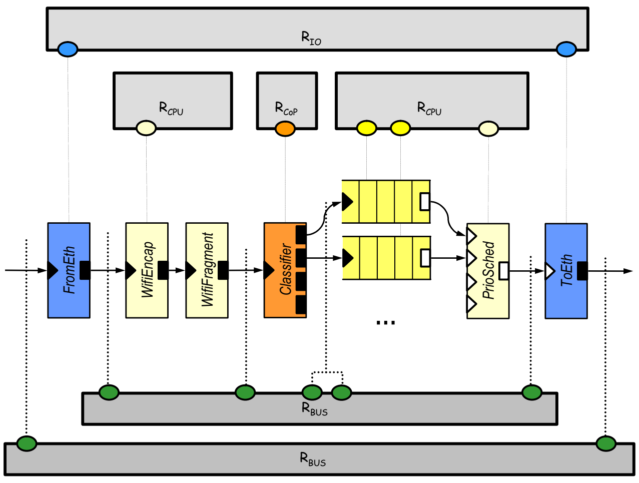


Figure 4.13: SystemClick representation of an application-architecture mapping.

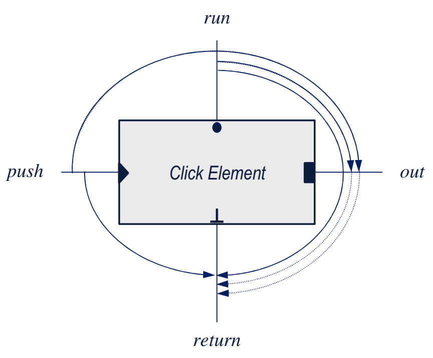
The platform representation is generated as a part of the SystemC performance model from the Click source, see Figure 4.3 on page 62. This source contains the application, architecture, and mapping descriptions. The model describes the architecture by instantiating resource managers, encapsulates partitioned Click/Cracc tasks with wrappers, and associates them with their resources. Furthermore, the model interfaces with a database for the performance annotation of tasks. The next subsections will address these aspects in more detail.

4.7.2 Performance annotation

For the simulation-based performance estimation of a design point, i.e. on an application-architecture mapping, each task of the application must be associated with its costs, e.g., in terms of execution and/or transfer time.

These costs depend on the particular resource a task is mapped onto. This means that every one of our Click/Cracc tasks must be characterized for the range of resources it can be mapped onto in order to enable design space exploration. The results of the characterization are gathered in a performance database, which becomes another input of the simulation as figure 4.12 shows. At runtime, the simulator accesses the database to lookup the costs of a {task, resource} pair.

Dynamic access is required since the particular costs can depend on the data and its flow through the task, i.e. Click/CRACC element. This is demonstrated in figure 4.14 (left). The figure shows the entry and exit points of a timed push element. A packet pushed into the element (▷) may be forwarded (□) or terminated locally (⊥). In addition, the element can be triggered by a timer (o). Usually, each arc from entry to exit point has different costs. In case of data- and state-dependencies these costs may vary dynamically, as the CRACC source code in the right part of the figure shows. Depending on the state ($\_forward$) the push function either takes the if or the else branch. There is an exit point for the element (out), which is taken only in the if-branch. Interestingly, the push function does not actually terminate at this point due to Click's function call communication semantics. Instead, the OUT function returns from downstream processing, the element continues its processing and eventually exits at $return$ (see also Figure 4.17 on page 85). Due to this behavior, the performance annotation must happen on the granularity of basic blocks. For the



```
PUSH_BEGIN( Element, this, port, p) {
    ...
    if (this->_forward) {
        ...
        OUT( this, port, p, "push-out")
    } else {
        this->_store = p;
        TIMER_SCHEDULE( this, this->_interval);
        UPDATE_META( this, "push-later");
    }
    ...
    RETURN(this, "push-ret");
} PUSH_END( Element, _this, port, p);

RUN_BEGIN( Element, this) {
    if (this->_store) {
        ...
         OUT( this, 0, this->_store, "run-out");
    }
    RETURN(this, "run-ret");
} RUN_END( Element, _this)
```

Figure 4.14: Entry and exit points of a Click element: left) on the Click level, right) in CRACC's source code.

lookup, relevant blocks are tagged (see the labels of the OUT, RETURN, and UPDATE_META macros in Figure 4.14, right). Tags and lookup enable the performance annotation at runtime.

Individual Click tasks can be characterized based on simulation, profiling, and estimations. For performance indicative results, we use Cracc and the processors instruction set simulators as described in Section 4.4 to characterize tasks running on embedded processors. The characterization of Click tasks mapped onto hardware depends on the RTL of the hardware building block. In any case, cycle or instruction precise data are derived. Section 4.7.4 following lateron in this chapter describes how the data is used and interpreted for performance simulation. In the next section, we will first focus on the Click/SystemC interface.

4.7.3 Embedding Click in SystemC

For performance simulation, we intend to execute click graphs within a SystemC environment. This way, the concurrent architecture platform can be modeled in SystemC as precisely as desired from building blocks (platform resources) with correct communication behavior (in terms of timing preciseness and bit trueness) while the function within a building block is specified by a Click application task graph. This means that an interface between SystemC and Click is required, which enables us to:

- slave a Click task graph into the SystemC simulation,
- execute Click graphs on different resources truly concurrently,
- dynamically annotate performance feedback, and
- map Click communication onto SystemC semantics and data types.

Existing approaches [132] that slave Click into discrete event simulation environments neither expose the concurrency of application nor the processing time to the simulator. Instead, they slave the Click engine into their simulation. Scheduling and arbitration of elements is left to the Click engine. The simulation time does not advance during the execution of elements. This means that the application is treated as if it were mapped onto a single processing node and as if the processing time were negligible. A tee element, for instance, which duplicates an input packet to all of its outputs, would output all copies in a burst just a delta time after it received its input.

In SystemClick, the concurrency of the Click application graph is fully exposed to the SystemC simulator and Click elements consume processing time. This is achieved naturally by executing concurrent task chains within different SystemC processes that synchronize their processing time with the simulation time at points of IO. We achieved this by encapsulating a (otherwise unchanged) Click task graph with FromSysC and ToSysC communication wrappers reusing the concept of IO wrappers as described for partitioning and mapping CRACC in Section 4.5. In addition, we changed the implementation of time and timers to reflect the SystemC environment.

Wrappers for communication and timers

SystemClick wrappers interface Click with SystemC and vice versa. This means that they first of all map Click's and SystemC's communication semantics and data types. In addition, they synchronize processing and simulation time, model operating system processing overhead, and associate task chains with resource managers.

Figure 4.15 shows how wrappers encapsulate the flow of *data* and *control* from and to Click task chains. A click chain can be triggered because there is data (push), data is required (pull), or a timer has expired (run). In all cases control is given to the Click task by one of the chain-starting wrappers (FromSysC-push, ToSysC-pull, Timer) as soon as the associated platform resource permits (more on this in Section 4.7.4). During execution, the task chain may request or produce data tokens (FromSysC-pull, ToSysC-push) or may schedule a timer for activation. When the task terminates, control returns to the activating wrapper and SystemC.

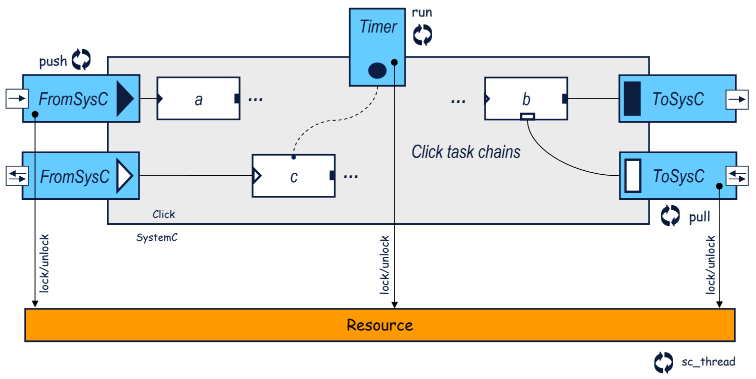


Figure 4.15: SystemClick wrappers (push/pull/timer) which integrate concurrent Click task chains into the SystemC environment.

On the SystemC-side, data communicating wrappers (From-/ToSysC) are sc_modules with ports for signals. In the case of communication between elements mapped onto different platform resources, a ToSysC and a FromSysC wrapper are inferred that are directly connected by a sc_buffer signal of the Cracc packet type, following the principles described in Section 4.5.3. In case the communication is terminated on the SystemC-side, specialized wrappers may convert data types. This is, for instance, required for pulls (pushes), which are intended for sampling (setting) the state of an environment signal. In this case, the signal state will be converted into content/color of a data packet and vice versa. Due to their special semantics, these wrappers must be instantiated explicitly by the application developer. The developer may also choose to refine pull communication in between task chains mapped onto concurrent resources. Strict and semantically correct mapping would require the puller to block until the data is generated by the pullee. More efficient implementations involve FIFOs, which effectively convert pull to push communication semantics (cf. Sec. 4.5.3). SystemClick supports the developer by providing a set of wrappers with sc_fifo interfaces that may be used for platforms with hardware support for FIFOs such as NOVA (see Sec. 4.6).

Processing and simulation time

Wrappers provide the infrastructure for accumulating the annotated processing time of Click elements and its synchronization with the simulation time.

Every time a wrapper triggers a Click task chain it creates a so-called Meta object which provides an interface to the processing time accumulator and the performance database lookup. This object travels through the Click graph with the flow of control. This is implemented by an element's entry, update, and exit point macros (see 4.14 on page 79) which pass a reference to Meta and update it with their processing time. If the flow of control reaches a wrapper again, i.e., there is a point of communication with the environment, the processing time is synchronized with the simulation time by a wait statement. Then, the communication takes place and the processing time accumulator is reset.

Figure 4.16 illustrates the relation between processing and simulation time using a simple Click graph which receives a packet (A) from the SystemC environment (FromSysC a), duplicates it by the Tee (b), and forwards both packets, the original (A) and the copy (B), to the SystemC environment again (ToSysC c). The arrival of Packet A at the wrapper starts the processing of the graph. Along with the flow of A, processing time is accumulated. By the time the packet reaches wrapper c, a processing time of a+b+c has accumulated but the simulation time has not advanced. The wrapper now calculates the simulation time from the Meta time value and waits until this time elapses. Then, it forwards packet A to the SystemC environment (OUT A), resets the processing time, and returns control to the Tee (b). The tee stores a copy of A, which is forwarded as B to wrapper c, too. This time, a simulation time of c'+b+c has to elapse before packet B can be pushed out. Wrapper c returns control to Tee b, which returns it to Wrapper a. The return chain, too, accumulates processing time (of c'+b'+a'). Thus wrapper a issues a wait as well.

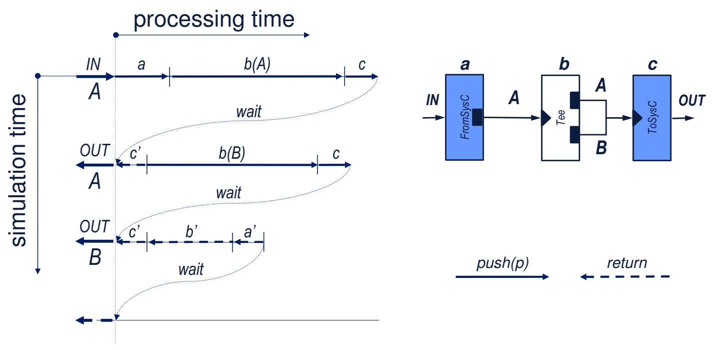


Figure 4.16: Synchronization of accumulated processing time and simulation time by wrappers (left) for the simple Tee graph (right).

In the example, both wrappers annotate processing time in the forwarding and the return path. This time represents the computational overhead of a platform for operating system aspects such as, e.g., communication, event handling, and context switching.

If the whole Click graph is mapped on a single CPU resource and performance annotation is enabled, as in our example, the original Click behavior can be observed: Packets are consumed and generated in intervals that depend on a task chain's execution time. Without performance annotation, i.e. with zero processing time, the tee element will output its packets at once as a burst at scheduling time. This resembles the behavior of ns-Click [132], which slaves Click into the ns-simulator.

4.7.4 Resource managers

Architecture resources are modeled by resource managers. Resource managers describe a resource in terms of their performance-relevant aspects, namely resource type, operating frequency, and arbitration policy.

Click communication and computation tasks are mapped onto resources by associating the task with a resource manager. Concurrent tasks may compete for a resource. The resource manager arbitrates these tasks (its clients) according to a particular arbitration policy. The winning task then locks the resource for a specific amount of time; the other tasks are blocked.

The arbitration policy can be selected explicitly per resource instance. A number of policies are relevant in our context:

- First-come-first-serve (FCFS) — Clients are served in the order they request the resource. The order of two request occurring at the same time is undefined. FCFS is the arbitration scheme used by Click running tasks in software on a CPU. In this case, requests are triggered by IO and timer events that start Click task chains.
- Round-robin (RR) — This arbitration scheme selects clients based on revolving priorities. The most recently served client becomes the lowest priority. RR is used, for instance, to arbitrate the access of a CPU cluster to a communication bus.
- Statically or dynamically prioritized (SP/DP) — The arbitration of clients is based on priorities that may be set once at association time (statically) or may change dynamically depending on a client's state and data.

The duration d of the resource lock for a task t depends on the task itself, the resource type r, and the operating frequency f, as shown in equation 4.1. The return value of the lookup $cost(t, r)$ in the performance database is interpreted as cycle count and scaled by the operating frequency.

$$d(t,r) = \frac{cost(t,r)}{f} \qquad \text{with} \qquad [\mu\text{s}] = \frac{[\text{cycle}]}{[\text{cycle/s}]} \tag{4.1}$$

Resource managers represent computation as well as communication resources. The exact meaning of a resource's parameters and lock time depends on the resource type.

Computation Resources

Computation resources are associated with Click tasks via the *From/ToSysC* or *timer* wrappers that start a Click task chain.

Computation resources may be characterized in the performance database using instruction counts rather than cycles. In this case, the frequency parameter f_c is specified in instructions/second, i.e. the frequency f must be scaled with the CPI of the resource (eq. 4.2).

$$d(t,r) \;=\; \frac{cost(t,r)}{f_c} \;=\; cost(t,r) * \frac{CPI(r)}{f} \tag{4.2}$$

In both cases reflects $d(t, r)$ the processing time of the Click task t on the resource r. In addition to the tasks of a chain T waiting times w_{io} for non-available IO resources contribute to the overall execution time D in case of blocking IO semantics:

$$D(T,r) = \sum_{t \in T} \frac{cost(t,r)}{f_c} \quad + \quad \sum_i w_{io}(i) \tag{4.3}$$

Whether an IO port has blocking semantics or not depends only on the connected SystemC link. A simple sc_buffer, for instance, does not block. SystemClick's sc_buffer_rm blocks in case the associated communication resource is not available.

Communication Resources

For a communication resource r specifies the performance database the transfer costs in cycles per unit. The frequency parameter $f' = N * f$, which is modified with the number of transferred bytes per cycle scales this value to a transfer time using the message length l as an extra parameter:

$$d(t,r,l) \quad = cost(t,r)\frac{l}{N * f} \quad \text{with} \quad [\mu s] = \left[\frac{\text{cycles}}{\text{unit}}\right] \frac{[\text{byte}]}{\left[\frac{\text{byte}}{\text{unit}}\right][\text{cycles/s}]} \tag{4.4}$$

Communication resources are associated with modified SystemC communication channels. Such channels lock/unlock the resource depending on transfer time and access semantics.

A write to the channel blocks the source until the resource can be locked. The resource is unlocked again as soon as the channel is read. The read is not affected by a sink's ability to lock its resource reflecting a single input register at the sink. Another write (to the same channel) may overwrite the previous data if it is not consumed in time.

4.7.5 Statistics

To support the evaluation of a design point's performance a range of statistics is provided by SystemClick. Depending on the required data, the designer may choose statistics functions that are associated with different SystemClick objects, namely resources, IO ports, or individual data packets. Over their lifetime these objects gather data that is reported by their *dumpstats()* function.

Resource statistics

Resource managers gather data on utilization and associated clients. In case of computation resources they are associated with Click elements, i.e. tasks. In case of communication they are directly associated with named point-to-point Click communication channels (links) and are thus indirectly associated with the IO sources and sinks of a named communication, too. For resources, the *dumpstats()* function reports:

- ID and type — Resources are identified by their ResourceID. The type corresponds with a column in the performance database.
- Number of clients — The number of clients that currently are and have been associated over the object's lifetime.
- Utilization — The processing/transfer time distribution per named client and the resource idle time are reported.

- Per client — Total processing/transfer time, number of invocations/transfers, minimum, average, and maximum processing/transfer time, and minimum, average, and maximum waiting time (per invocation).

IO element statistics

To refine the resource statistics, From- and ToSysC IO elements may gather statistics on their flow of tokens broken down by token size (i.e. the packet length in case of packets) or token color (i.e. the data's value in case of non-packets). *Dumpstats()* reports: name and type; number of passed packets/tokens; histogram of packet lengths / token color; and minimum, average, and maximum processing time per length/color.

Start points of tasks (FromSysC_push, ToSysC_pull) report on a per-task basis. They account for the full processing time of a task for the triggering token/packet. Other IO elements such as a push output actually report the individual processing time of a token/packet. Figure 4.17 shows the difference for a Tee element which duplicates packets received at its input.

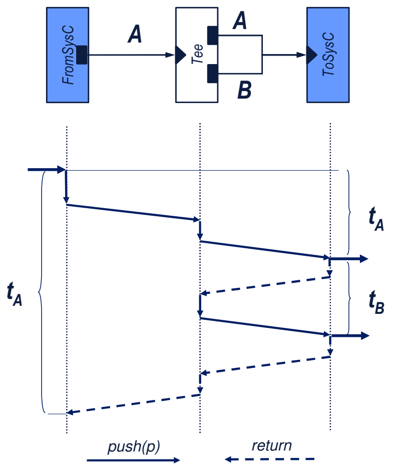


Figure 4.17: Difference in IO packet statistics of task starters (left) and other IO elements (right).

Per packet statistics

Per packet statistics are helpful to follow packets on their path through the system across different resources. For this purpose packets store the sequence of passed elements over their lifetime. For latency and delay analysis, timestamps for creation and expected deletion/processing time can be annotated. They may be used, for instance, to detect missed deadlines.

4.8 Quality of Results

The SystemClick framework in combination with the CRACC/NOVA platform promises a fast yet performance indicative way for the evaluation of application-architecture mappings. This section quantifies this claim in terms of the achieved simulation performance and quality-of-results.

4.8.1 Simulation performance

To assess the simulation performance of SystemClick, we are looking at two application benchmarks exhibiting different characteristics, a) the transmit processing chain of the wLAN model, and b) the full PCI Express model. The wLAN benchmark is a kernel which exposes the most compute-intense function of the protocol (tx frame processing) while the PCI Express model has much less computation per packet but is a complete application. Both applications will be exhaustively discussed with respect to their performance in Chapter 5.

Table 4.2 compares the simulation speed for the benchmarks. As a baseline, we use the CRACC code generator and execute the applications on a commercial instruction set simulator (MIPSSim[2]) representing a single-core architecture. The simulator is configured for fast execution, i.e., instruction instead of cycle resolution and IO intense tracing features disabled. As an upper bound for the speedup, we also execute the applications natively on a x86 host system.

Table 4.2: Relative simulation speed for two benchmarks

Type	*Description*	*wLAN*	*PEX*
MIPS ISS	performance	1	1
Standard SystemC[1]	behavior only	1280	572
SystemClick	performance annotations	200	80
SystemClick	performance + 2PE architecture	125	
SystemClick	add. sync + trace + prints	80	
Native X86	behavior only	3400	4118

[1]) SystemC 2.2 (Open Source Kernel)

Compared to the baseline, a speedup of two orders of magnitude (80-200) can be achieved by the simulation of a single-resource application-architecture mapping with performance annotations. The simulation speed varies with the benchmark due to different element granularities and the frequency of SystemC function calls.

Also, the simulation speedup decreases with model complexity. Adding another PE and a shared communication resource, for instance, reduces the speed by 38% (from 200 to 125) for the wLAN benchmark. For such a setup, however, two communicating instruction set simulators would be required for a fair comparison. Adding tracing and verbose outputs costs 30% of the speedup.

Our speedup results are in line with the ones reported, e.g., in [34]. Looking at multimedia function kernels, the authors in [34] measure speedups between 93 and 332, averaging at 200. However, the multimedia function kernels may be more favorable for speedup measurements since they most likely require more computations per token (i.e., image) than our applications leading to less frequent SystemC interaction and better performance. In addition, [34] uses performance annotations that are statically compiled-in while SystemClick accesses a performance data base at runtime. The use of static annotations or efficient caching therefore may further improve SystemClick's performance.

[2]www.mips.com, part of the MIPS software toolkit

4.8.2 Simulation accuracy

We are comparing the accuracy of performance-annotated SystemClick with the instruction set simulation using the wLAN benchmark.

For the simulation, the database of our framework must be populated for Click elements/subfunctions. The benchmark has both state and data dependencies (e.g., fragmentation) leading to several tagged exit points per element in the database. Applying the IMIX packet distribution, it is profiled on the ISS. The best match with the ISS performance is 1.54% when the profiled case is re-simulated with SystemClick. For longer runs, accuracy slightly drifts, see Table 4.3.

Table 4.3: SystemClick accuracy: Total number of instructions

IS Simulator [insn]	*Performance Sim. [insn]*	*Accuracy [%]*
1.948M	1.978M	1.54
3.580M	3.653M	2.04
8.162M	8.364M	2.47

The accuracy significantly depends on the quality of profiling data and the similarities of the traffic load at profile and simulation time. Changing the packet size distribution for the simulation, for instance, would lead to further inaccuracies ranging from 4% to 7% caused by the averaging effect and length-depending discrepancies of the cycle count. This is why multiple profiles for different packet length distributions (min, typ, max, and imix) are being maintained.

4.9 Chapter summary and conclusion

We have developed a comprehensive methodology for the application-driven development and deployment of programmable platforms, which we will use for the exploration of a flexible interface solution. In fulfilment of the requirements listed in Section 4.1, we are concluding on the key facets of our methodology:

- Our approach (published in [172]) is based on the Y-chart which enables the design space exploration by the separation of application, architecture, and mapping concerns.
- Applications (the packet-oriented IO protocols) are specified in a modular way using the Click model of computation. In order to ensure representative and performance-indicative models, the Click simulation framework is used to verify the protocol function independent of the implementation. The resulting application description is correct in terms of function and required protocol timing.
- From Click specifications, we derive efficient implementations for embedded platforms using our CRACC code generator (published in [173]). CRACC understands Click and generates C code for embedded multiprocessors, which is compiled individually per processing element using the element's native tool chains. A Click element is a natural encapsulation of data and processing required for a certain task. The number of

mapping and partitioning choices can thus be reduced considerably to a set of rational alternatives.

- As a modular and programmable platform for packet processing applications, we have introduced NOVA (published in [166]). NOVA 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 the CRACC framework. Our results show that the overhead of hardware and software modularity is reasonable for NOVA compared to state-of-the-art techniques and that NOVA is usable for systematic application-driven design space exploration.

- For early design space exploration, we have presented the SystemClick framework (published in [171]). SystemClick abstracts platform architectures as sets of shared resources, which arbitrate communication and computation tasks and consume time. From application, architecture, and mapping specifications, the framework generates SystemC simulation models that can be executed natively using a performance database. Indicative performance data is derived, e.g., from profiling partial CRACC/NOVA implementations. The combination of performance modeling and functional correctness enables the quantitative assessment of implementation alternatives for timing critical protocols. Sensitive design parameters such as the arbitration of shared resources and granularity of processing kernels are exposed to the designer. The integration with SystemC finally allows the reuse of simulation and analysis as well as refinement infrastructure.

In the next chapter, we will deploy our exploration tools and methodology to investigate the feasibility of a programmable interface platform.

Chapter 5

A Programmable Architecture for Packet-oriented Communication Interfaces

This chapter investigates the feasibility of a programmable platform for packet-oriented communication interfaces.

Programmable solutions are particularly interesting for the implementation of the discussed standards since they provide us with a platform for all protocols and allow us to adapt to late changes in the specifications. Remember that the processing of the investigated communication protocols is a peripheral service to the SoC's main processing core, i.e. this peripheral is separate from the micro-architecture of the multi-processor system-on-chip, as it is currently deployed. This section will elaborate on the question whether existing building blocks of a processing platform such as processing engines and communication infrastructure can be used again in order to implement the peripheral functionality.

Following our application-driven methodology of the last chapter, we will first look at a *fully programmable* solution based on an embedded standard core. Such a solution is most flexible as all functions are software, implemented in plain (and portable) ANSI-C, and there is no special purpose hardware. This solution is a corner case of an implementation and provides the necessary data for the exploration of the design space along several axis. Analyzing the performance, we quantitatively explore the design space with respect to number and type of cores, instruction set extensions, application-specific hardware accelerators, and communication topology. This results in an *optimized architecture*, which will be presented toward the end of this chapter. Furthermore, for the analysis, we quantify the overhead which is introduced by the modularity of the hardware/software platform and compare our solution with related work.

5.1 Fully programmable solution

For the profiling and analysis of the fully programmable solution, we will look at two protocols, PCI Express and wireless LAN. PCI express has the highest throughput requirements while wLAN requires the most complex protocol processing. Their joint requirements are the worst-case to be supported.

5.1.1 Architecture setup

Figure 5.1 shows the architecture of a fully programmable system based on a single embedded core. The module only comprises the processing core, its instruction and data memories, and memory-mapped transfer queues. All protocol functions are implemented in software on the embedded core.

Memories and transfer queues can be accessed with one cycle latency, i.e. they run at the same frequency as the processing core. The data memory has an extra interface to the SoC core which is used to transfer transactions between SoC core and IO module concurrently to the IO module operation.

In transmit direction data flows through the architecture as follows: an outbound transaction is stored in the data memory (1) and the associated message comprising the packet descriptor is sent to the module (2). The core reads the descriptor from its inbox (3) and processes the packet as specified by the Click task graph (4). If required, the packet is queued and scheduled (5) before its data and extra processing information can be sent to the PHY queue (6).

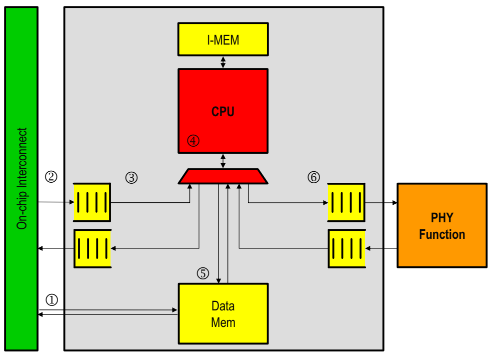


Figure 5.1: Fully programmable single-core architecture. All building blocks run at the same clock frequency. Fifos and the dual-port memory synchronize with the environment.

Not shown in Figure 5.1 is the additional PHY status register (read-only) which maps important state of the PHY function directly into the CPU memory space. It may be used to access (pull) the immediate link state, e.g., carrier sense in the case of Ethernet and wireless Lan. Its use is optional and depends on the protocol implementation. Alternatively, PHY processing state notifications can be generated (pushed) using the PHY queue as mentioned before.

Note that the protocol-specific PHY function consistently includes frameing/deframing for all protocols. While this is consistent with the standard for most protocols, GbE requires some adaptation logic to connect to standard compliant PHYs (GMII interface).

5.1.2 Profiling results

For the profiling and the analysis of the fully programmable architecture, we will examine two protocols, PCI Express and wireless LAN. PCI express has the highest throughput requirements. Wireless LAN requires the most complex protocol processing. To report the results, we follow the different packet-processing paths for receiving and transmitting packets as derived in Section 3.3. For each path the overall instruction counts are listed, which are based on the measurements of the individual functions. Those functions that contribute most are identified and listed separately. The overhead of this architecture for task handling and packet communication will be quantified at the end of this section.

Profiling procedure

For the characterization at the granularity of Click elements, we cut our applications into subgraphs. These subgraphs either contains multiple elements that are executed sequentially or they contain single elements that require careful simulation due to processing complexity and deep state such as the DCF function of the wLAN model. Each subgraph is mapped individually onto the processor target using the CRACC code generator as described in Section 4.4. CRACC generates modular and portable ANSI-C code suited for a range of processors. We chose the MIPS family of embedded processors since it provides a good performance/area foot print as will be shown later (cf. Fig. 5.6). The core interacts with its surroundings as described in the previous section (Sec. 5.1.1).

Profiling uses the core vendors tool chain which is based on GNU's compiler, gprof profiler, and comes with the cycle-precise MIPS-sim simulator. The tools are incorporated into our work flow. Compilation throughout the profiling process was carried out with *-O2* and *-O3* optimization. Simulation statistics are post-processed to extract results per Click element. As GProf relies on a function call graph produced by instrumented code, two separate simulation runs are used to achieve more accurate results without profiling overhead.

The measurement results reflect the computation per packet and function for a constant packet stream without bursts. The CPU is assumed idle with only one task graph running. This separates the effects of traffic characteristics and resource availability from the actual computation and communication. The measurements were executed with four different packet size distributions following the simple Internet protocol mix [2]: Minimum (64 B), Typical (550 B), Maximum (1498 B), Simple iMix (7:4:1).

PCI Express execution profile

The profile of executed instructions for the processing paths of PCI-Express is listed in Table 5.1. The table lists the common Cyclic Redundancy Check (CRC) function separately since its profile depends on the packet length. The remainder of the processing only accesses the packet header and is independent of the packet length. Thus, the number of executed instructions, e.g., for an outbound transaction of minimal size is 331 + 652 = 983 instructions. That makes path A the most instruction-consuming case. Link layer packets for acknowledges and flow control are only 6 bytes long.

Table 5.1: Single packet instruction counts for the PCI express model.

Path	Description	Instructions / packet size			
		min	typ	max	imix
A	Outbound transaction	331			
B	Inbound transaction	205			
C	Outbound Acknowledge*	77			
D	Inbound Acknowledge*	255			
E	Flow control (Tx)*	65			
F	Flow control (Rx)*	163			
	Common functions				
	+ CRC	652	6285	16712	3868

*) The frame length of control frames is 6 Bytes.

Wireless LAN execution profile

Table 5.2 summarizes the profiling results for the different processing paths of the wLAN model. Figure 5.2 shows the corresponding diagram. The instruction counts for encryption and CRC calculation are listed separately as before.

Table 5.2: Single packet instruction counts for the wireless LAN model.

Path	Description	Instructions / packet size			
		min	typ	max	imix
A	Outbound dataframe	2326	2618	2591	2445
B	Inbound dataframe	1151	1251	1171	1186
C	Ackn Tx [16B]**	510			
D	Ackn Rx [16B]**	739			
G	RTS tx, CTS rx**	1443			
H	RTS rx, CTS tx**	1363			
I	Management Tx*	3132			
J	Management Rx*	3250			
	Common functions				
	+ CRC	652	6285	16712	3868
	+ Encryption (WEP)	8515	31464	74129	32299
	+ Decryption (WEP)	8478	31427	74091	32262

*) Beacons. **) The frame length of control frames is 14-20 Bytes.

The results show the handling of management frames, i.e. beacon generation and reception, to be the most compute-intense processing paths. However, management frames are processed relatively infrequently compared to data and control frames. Beacons, for instance, are generated in tens to hundreds of milliseconds intervals, and are therefore not performance-critical. Of the more frequent data and control frames, the transmission of data frames in outbound

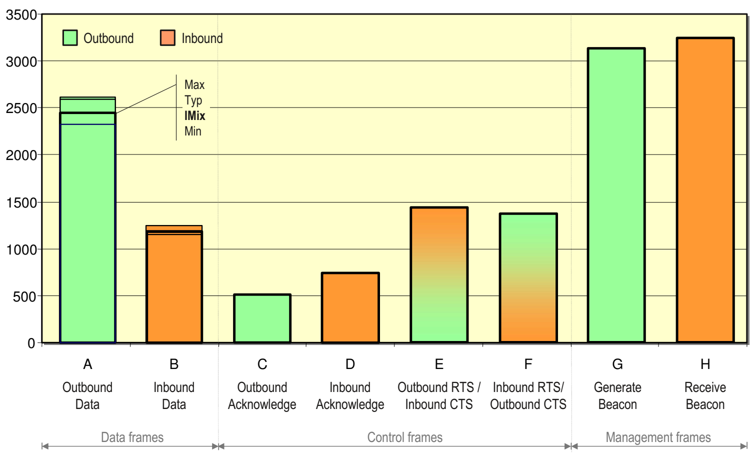


Figure 5.2: Single packet instruction counts per processing path for wLAN excluding CRC and WEP functions.

direction is the most instruction consuming case.

Since most of the payload-dependent processing can be attributed to the CRC and crypto (WEP) functions, the numbers do not depend on the packet length much. The little influence left is due to fragmentation and reassembly related rate changing effects.

Overhead for communication and task handling

In addition to the protocol function, the overhead for packet IO must be taken into account. This means that not only the memory-to-memory packet processing on a CPU system must be considered but also the processing overhead required to transfer the packet, i.e. the descriptor and/or the packet data from the inbound interface into the memory and from there back to the outbound interface. In case of the fully programmable solution, which relies on the CPU for the data movement, the overhead includes:

- SoC interface inbound — Packet descriptor messages from the SoC core are received in the inbox. They are read by the CPU and stored in data memory. Then, the associated task chain is identified and started.

- SoC interface outbound — A local packet descriptor is moved from the local data memory into the outbox.

- PHY interface outbound — Packet context and packet data are moved from the data memory into the outbound PHY fifo.

- PHY interface inbound — Packet context and packet data are read from the inbound PHY fifo and stored in the local memory. In the process, a packet descriptor is generated and the protocol task chain is started.

The number of required instructions depends on the amount of data that are transferred and the number of tasks that reside on the processor. Table 5.3

quantifies the overhead for packet descriptors of 64 Byte and the usual distribution of packet sizes. These overhead functions are embedded into the application without further penalties (such as interrupt handling and context switches) in the critical path.

Table 5.3: Instruction counts for task handling and packet IO.

Read descriptor from inbox (64B)	50[1]
Identify and start task chain	70[2]
...	
Write packet context to PHY fifo (64B)	50
Write packet data to PHY fifo (min, typ, max, imix)	50, 416, 1125, 262
Read packet context from PHY fifo (64B)	50
Read packet data from PHY fifo (min, typ, max, imix)	50, 416, 1125, 262
Start task chain ($N_{Tasks} = 1$)	20
...	
Write descriptor to outbox (64B)	50

[1]) 2 + S_{PD}/4*3 with $S_{PD} = 64$ [2]) approx. 12 + 7*N_{Tasks} with $N_{Tasks} = 8$

Many eco-systems of embedded cores integrate hardware support for timers and counters. Throughout our analysis, we therefore assumed that the handling of the global time structure (which usually is stored in multiple words in local memory and is periodically updated using a hardware counter), is avoided as another source of overhead by mapping the system time into a permanent hardware register (64b) that is always up-to-date.

5.1.3 Performance analysis

To analyze the performance of the single processor system, we will estimate the clock frequency that is required to meet the throughput requirements of the protocols. For this purpose, we will select the most cycle-consuming case from our processing paths. That is the transmission of data transactions (path A). For the analysis, we will further assume that such transactions are sent out back-to-back. This means, our results will be pessimistic since we do not consider less cycle-consuming cases and protocol-related gaps between frames.

Throughput requirements

PCI Express. For PEX, the required throughput at the MAC/PHY interface is 2.0 Gb/s per unidirectional link. There are no interframe gaps at the PHY layer, the bandwidth is fully utilizable. For the throughput estimation, we assume a cycle count per instruction (CPI) of 1.2 which represents the MIPS M4K processor for our application domain, as we will explain in Section 5.2.1. Considering the different packet sizes, clock frequencies between 3.6 and 5.8 GHz are required for packets ranging from 64 to 1512 Bytes, as Figure 5.3 shows.

Wireless LAN. The IEEE 802.11g standard has with 54Mb/s on the air the highest throughput requirements of the considered wLAN protocols. At the MAC/PHY interface the throughput depends on the packet size since the protocol defines interframe gaps for sequences of data frames of at least 28us

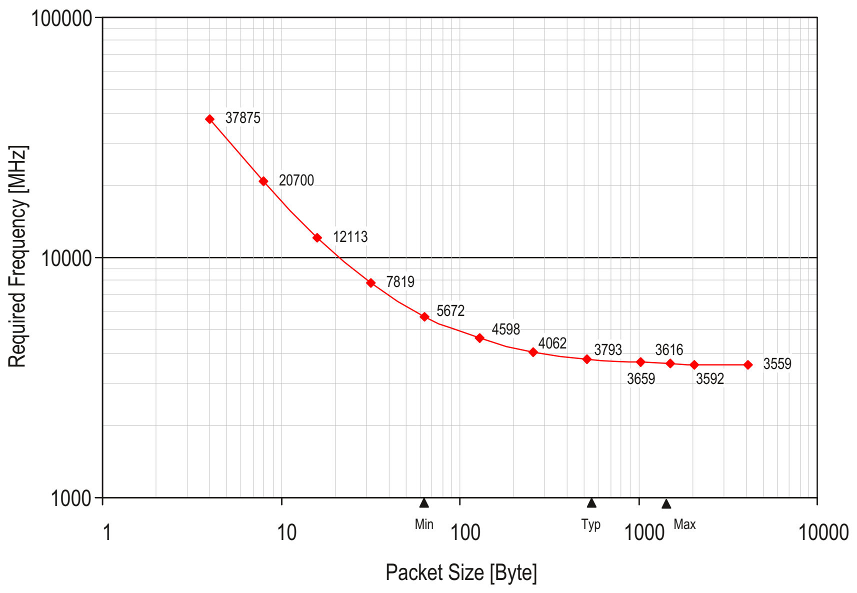


Figure 5.3: Required clock frequencies to meet PCI Express throughput requirements depending on the packet size.

(DIFS time) and a PHY layer overhead that is equivalent to another 26us per packet [49]. Due to this considerable overhead, clock frequencies between 223 and 415 MHz are required. Largest contributors are CRC and WEP encryption functions which require up to 96% of the cycles.

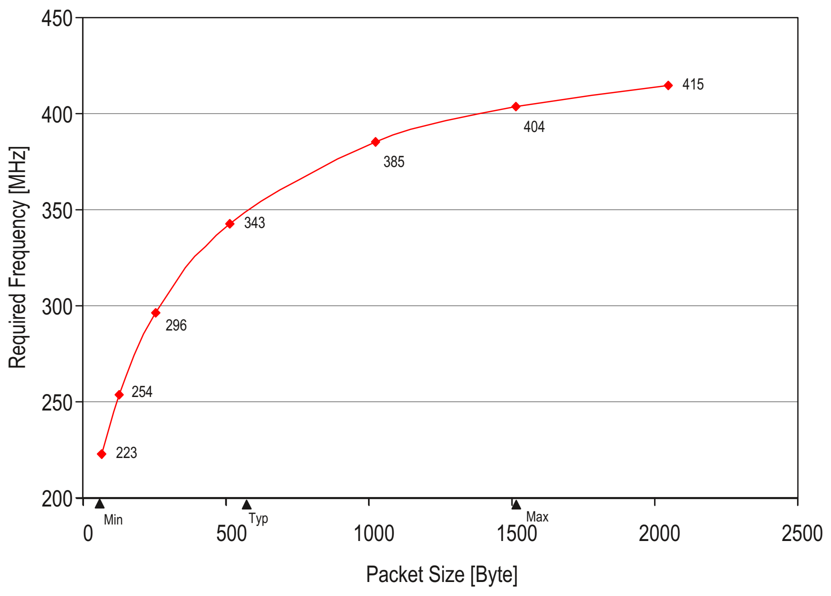


Figure 5.4: Required clock frequencies for the 11g wLAN throughput depending on the packet size.

Realtime requirements

PCI Express is latency-insensitive. Wireless Lan protocols, however, define firm deadlines for transactions of stations and access points on the air, see Section 3.3.5. For the MAC time budget most relevant is the timing for sending a response for a just received frame.

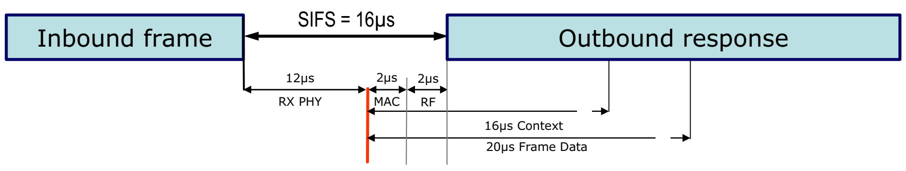


Figure 5.5: MAC time budget for response frames.

A typical response time is 16 μs as defined in IEEE 802.11 [36]. In case of 11g the time is defined as a 10 μs SIFS time plus an extra frame extension time of 6μs. Since the interframe gap is defined on the air, the PHY receive and transmit processing delays are included in the interval and must be subtracted. Figure 5.5 shows this in detail. In the Figure, the last byte of the inbound frame is given to the MAC by a Rx PHY only after 12 μs. The CRC can be calculated and the MAC processing starts. For the response frame transmission, the Tx PHY requires the frame data and context after different setup times. After only 2 μs, the MAC must signal whether a transmission will follow. The frame context is required after 16 μs. The first word of the frame itself finally must be available after 20 μs. For the purpose of the analysis in this chapter, we will focus on the frame context deadline. In order to be met, this deadline requires the completion of the entire protocol processing.

Table 5.4: Estimate of the required clock frequencies for the realtime response of 802.11 frames.

	min	max
Read data frame from PHY (PD+data)	100	1175
CRC processing	652	16712
Receive path processing	1151	1171
WEP decryption (resource not available)	8478	74091
Scheduling Overhead	70	70
Ack generation	510	510
CRC processing	326	326
Write context	50	50
Sum [Inst]	11337	94105
Sum [Cycles, CPI=1.2]	13605	112926
Required frequency [MHz]	850	7058

Table 5.4 shows the required clock frequencies for meeting this deadline. Frequencies up to 7 GHz are necessary to support the required cycles. The table assumes an idle resource which is available without delay for the processing of

the received frame. In this case, the decryption function shown in the table would not be part of the critical path since a packet can be decrypted after its acknowledgement. On the other hand, the CPU may just have started the processing of a large tx frame (dominated by the encryption function) so that the WEP cycles represent the non-availability of the resource for a ballpark frequency estimate. This will be discussed further in Section 5.2.4. Figure 5.15 (p. 113) shows histograms of the processing jitter for a busy core.

5.1.4 Observations

The fully flexible single-core architecture cannot fulfill throughput and realtime requirements in all cases considered. Especially the high throughput rates of PCI Express are difficult to achieve. In the case of the 11g wireless LAN protocol, the throughput requirement is moderate and would be achievable. However, the required deadline for the response generation cannot be met for larger frames (not even on an idle resource). To enable a processor based solution, the cycle counts must be reduced and the realtime critical path must be shortened significantly.
The performance analysis in particular revealed:

- Per-packet protocol processing — Most of the protocol processing happens per-packet and is based on the packet context only, which is correlated with the packet header. The context is either extracted from the header or forms the header. The packet payload is rarely touched.

- Cycle-consuming per-byte payload processing — Those functions that actually touch the whole packet such as CRC calculation encryption, framing/deframing, are the most cycle-consuming functions.

- Non-deterministic processing times — The dependency of the payload processing functions on packet length leads to irregular and monolithic processing which makes performance prediction and task scheduling difficult if not impossible in some cases.

- Unexploited parallelism — The protocols exhibit data- and task-level parallelism which would enable partitioning and mapping onto concurrent resources without data coherency problems.

Further design space exploration is required to find a feasible platform proposal. In the next section, we will therefore explore several axes of the platform design space and discuss their impact on the system's performance compared to the fully programmable single-core solution.

5.2 Design trade-offs and consequences

The required cycle counts for the fully flexible single-core architecture are too high for an implementation. This problem can be addressed by different design approaches which will be individually explored and quantitatively analyzed in the following subsections. The fully flexible single-core solution serves as baseline for this exploration of the platform's design space.

5.2.1 Core type and configuration

To compare the performance of different RISC cores and their compilers, we profiled packet processing functions implemented in CRACC on a representative set of synthesizable embedded 32bit processors using cycle-accurate simulators.

For the analysis we assumed a 90nm Infineon technology under worst-case operating conditions. The cores were configured to there smallest area footprint. Wherever possible, we disabled unused options such as caches, and cache controllers. Most of the cores (PowerPC 405D, ARM926EJS [63], MIPS M4K [14], ARC600 [63]) have five pipeline stages and run at clock frequencies of 290 to 440 MHz (Worst case). N-core [106] and ARM7 [5] have 3 pipeline stages, the processing engine [133] has 4. These cores run at frequencies of 160 to 270 MHz.

The results published in [173] reveal that among the modern general purpose 32bit RISC cores (ARM, MIPS, and PowerPC) the MIPS 4K has the best performance/area trade-off for the packet processing application domain (cf. Figure 5.6). For this reason, the MIPS 4k is used as the primary core for our analysis.

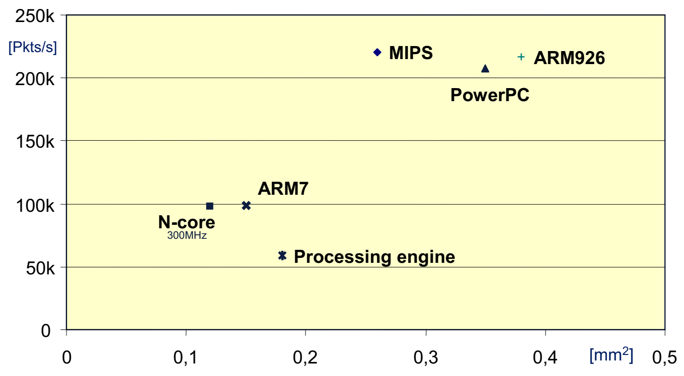


Figure 5.6: Performance of selected processor cores over core area for a packet processing benchmark [173].

Among the smaller cores which trade off speed against area (ARM7, N-core) or deploy a much reduced instruction set (packet processing engine), the N-core would be a reasonable second choice. The N-core was developed to enable efficient modifications on micro-architecture and compiler simultaneously – as they are necessary, e. g., for the exploration of ISA extensions. The *Processing engine* with its highly optimized but limited instruction set could not perform well. Performance improvements that are achieved with the specialized instructions are annulled by insufficient general purpose instructions as matched by a compiler. Improvements could be achieved by coding critical parts manually in assembly. We will come back to these points later, in Section 5.2.3, where we analyze the performance of an application-specific instruction set.

Core configurations

The impact of corner case configurations on the MIPS 4k's performance is analyzed using the cycle-per-instruction (CPI) measure. To derive the CPI we profiled the most complex application, wireless LAN, and measured elements individually as well as the application as a whole. For the M4k, two configuration options proved to be relevant:

- Memory interface — Configured with a unified memory interface (as it was used in the previous section) the M4k has a CPI of around 1.45. This is a significant performance penalty since the configuration with dedicated interfaces for instruction and data (Harvard architecture) requires only 1.08 cycles per instruction.
- Fast multiply unit — The impact of removing the fast multiply unit is rather small (CPI 1.13 vs. 1.08) and is limited to the few elements relying on multiplications (e.g. those, that calculate transfer times in wLAN such as SetDuration).

From the cycle-accurate profiling results for the M4K, the CPI ratio per processing path inside every Click element can be computed. These CPIs range from 1.0 to 1.2 with an average of 1.06 over the CRACC application library. This is slightly better than the ratio for the complete application setup from above, which included more branches in code for scheduling and calls to push/pull functions. We therefore use 1.2 as the upper bound for our instruction-to-cycle calculations in this chapter.

5.2.2 Application-specific hardware accelerators

Customized hardware accelerators can help to reduce the cycle-count requirements for the communication protocols by offloading compute-intense tasks. Their impact on the system's performance, however, depends on the way of their integration into a system. Before we discuss the performance impact, let us therefore first look at different integration patterns.

Accelerator integration

With respect to the flow of control and data hardware accelerators, can be deployed following different strategies:

- Flow-trough (FT) processing — Flow-through coprocessors usually process data while they pass through the unit. The work in a pipelined fashion as pre- or post processors for the main processing unit.
- Lookaside (LA) processing — Lookaside accelerators are slaved to the main processing unit. They are operated similarly to function calls. The main processing unit passes commands (function's name) and data (parameters of the function) to the accelerator and receives the results of the processing.

Flow-through engines can be designed to run fully autonomous and parallel to the processor. This means that they can offload all processing cycles from the processor without introducing extra overhead for communication and synchronization. LA engines, on the other hand, require processing cycles for the transfer of information. Depending on the complexity of the operation they may still run concurrently to the processor. But in this case extra overhead for synchronization will be required in addition to the communication.

Performance impact

The execution profiles for the fully programmable solution are dominated by the payload processing functions (cf. Tables 5.1 and 5.2). Offloading them to special purpose accelerators can relieve the cycle budget significantly.

This insight is not new. **CRC units**, for instance, are usually implemented in hardware and efficient flow-through solutions exist [70] and are being used in network processors. We therefore assumed full acceleration for this task.

Another cycle-consuming aspect is the processor's involvement in the movement of data between transfer fifos and local memory. A **DMA hardware accelerator**, which moves the data from the transfer fifos into the local memory and vice versa, could run concurrently to the processor and relieve it from this burden. Such a DMA engine must be managed by the processor in a lookaside fashion. To enable decoupled operation, the coprocessor has inboxes (command queues) and outboxes (result queues). In a steady state, the processor initiates a transfer (write) every time the result of the previous one (read) is consumed. Including status check, ten instructions are required for such a transfer.

Assuming hardware acceleration for CRC and a DMA engine for moving the packet data, the overall system performance for PCI Express could be improved as shown in Figure 5.7. With the accelerators, only the protocol processing is left which is constant per packet so that the required clock frequency decreases with packet size. If the DMA engine is used for the packet descriptor as well, the IO overhead can be reduced further and the required cycles would decrease by another 9%.

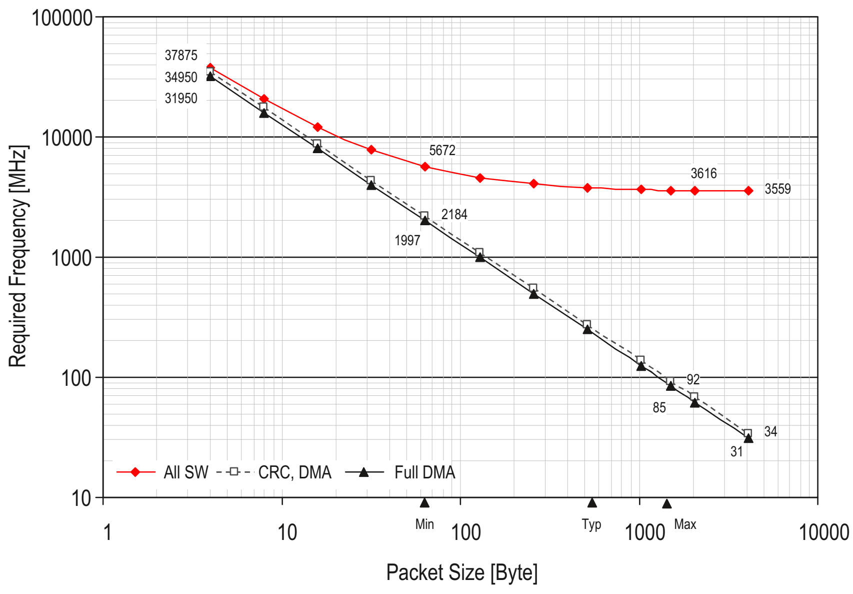


Figure 5.7: Impact of hardware accelerators for DMA and CRC on the required clock frequencies for PCI Express.

The same hardware accelerators for CRC and packet data DMA applied to wireless LAN save up to 19% of the cycles as Figure 5.8 shows. In this case clock frequencies up to 338 MHz are required. If the **WEP encryption/decryption** elements are accelerated as well, the required frequency shrinks to less than 50 MHz. However, encryption is only used by wireless LAN, and the required cycles are moderate. Multi-standard security coprocessors usually require considerable resources, see, for instance, the implementation in [187] which describes a (micro-) programmable coprocessor solution. Leaving encryption executed in software would, therefore, be an option in our case.

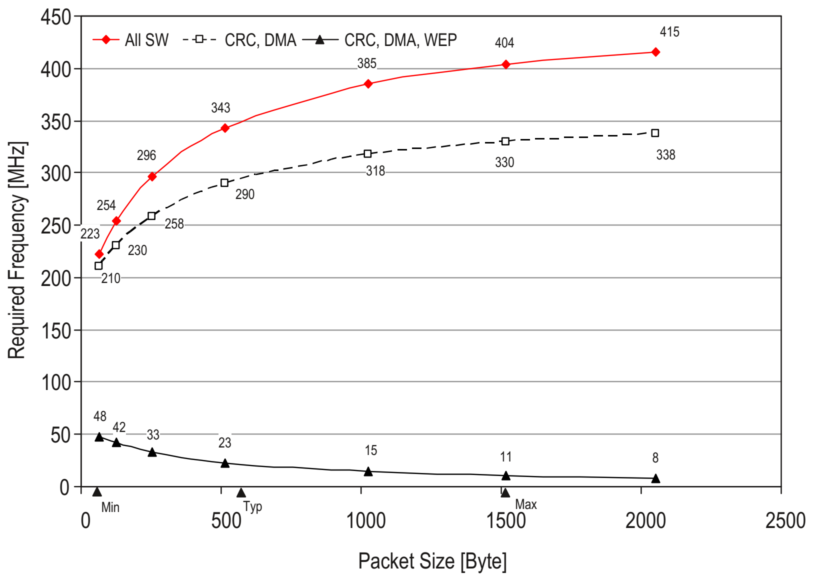


Figure 5.8: Impact of hardware accelerators for DMA, CRC, and WEP on the required clock frequencies for wLAN.

5.2.3 Application-specific vs. general purpose ISA

This subsection explores the trade-offs in programmability and speed between a general purpose and an application-specific instruction set (ISA). In order to derive computational requirements on an application-specific instruction set we define a processing engine with an micro architecture which is optimized for packet processing. We then derive static profiling results for PCI-Express as the specification with the highest throughput requirements, and compare the results to our fully programmable solution.

Micro-architecture Model

As an application-specific instruction set processor (ASIP) targeted at packet processing tasks, it is clear that we need specialized instructions for bit-level masking and logical operations [133]. We therefore base our simplified micro-architecture model on the processing engine used for the comparison of cores in Section 5.2.1 which is comparable to the one published in [133]. We also assume support for a large number of general-purpose registers (GPRs) per thread. Intel's processing engine [31], for instance, provides 64 GPRs per thread. We do not believe that the size of the register file is a limiting factor of our application.

Indeed, our analysis showed less than 30 registers used concurrently in the worst case. We, therefore, do not discuss the influence of register spills in this context. The data path is assumed to be 32 bit as in all major network processors. Since we consider reliable protocols, the support for timers is mandatory. Hardware timers are implemented using special registers and can thus be accessed quickly (see also Section 5.1.2).

Simplified Instruction Set with Timing

The following instruction classes are used in order to statically derive the execution profile of our Click elements. An application-specific, register-to-register instruction set is assumed to support bit-level masks together with logical, arithmetic, load, and store operations to quickly access and manipulate header fields.

- *Arithmetic operations (A):* additions and subtractions take one cycle.
- *Logical operations (L)*: such as *and*, *or*, *xor*, *shift*, and compare (*cmp*), take one cycle.
- *Data transfer operations:*
  - Load word (*ldr*) from memory: two cycles latency from embedded RAM.
  - Load immediate (*ldi*), move between registers (*mvr*): take one cycle.
  - Store word (*str*) to memory: one cycle latency on embedded RAM.
- *Branch instructions (B):* two cycles latency (no consideration of delay slots).

Our Click elements are annotated with a corresponding sequence of assembler instructions that are needed to perform the task of the element. These assembler programs of course depend on particular implementation decisions, which will be described next.

Optimized Implementation

For the implementation, we assume an optimized implementation of Click task chains (cf. Sec. 4.4.3). Click elements that belong to one chain are mapped to the same thread of the processing engine and share temporary registers, e.g. a pointer to the current packet descriptor does not need to be explicitly transferred to the next Click element. Furthermore, the code from Click elements within push and pull chains without branches on the same thread can be concatenated so that jumps are avoided.

We assume the IO overhead for the ASIP to be the same as before. The movement of the packet descriptor from and to the registers of the processing engine takes 18 cycles. Another 64 cycles are required for management and scheduling of a small set of software tasks.

Profiling Procedure

The static profiling of our Click elements is executed as follows: Given our models in Click, we annotate each of the Click elements with the number of assembler instructions that the described packet-processing engine would need to execute the element. An example is given in Figure 5.9. The code excerpt is part of the transaction layer flow control element. The method *push_ta*() handles the transmission of transactions to the data link layer. The method *hasroom*() is a helper method that checks the available space at the receiver by using the provided credits. The helper method is rather small, and we thus assume that *push_ta*() can use *hasroom*() in-line. Each basic block is commented with the number of required assembler instructions, following the implementation assumptions described in the preceding subsection. For each element, we choose a worst-case execution path and use the resulting number of instructions as the annotation of this element[1]. Annotations may depend on the packet length and are parameterized accordingly.

```
bool CLASS::hasroom(unsigned int ta, unsigned int header_size,
                    unsigned int data_size) {
  if ((header_size + _credit[ta][0][0] > _credit[ta][0][1])    // _credit contains only 12 values,
       && _credit[ta][0][1])                                   // i.e. offset calc. is considered with one add
          return false;                                        // 3 add (2 offsets), 1 cmp, 1 and, 2 ldr (from _credit), 1 branch
  if ((data_size   + _credit[ta][1][0] > _credit[ta][1][1])
       && _credit[ta][1][1])
          return false;                                        // 3 add, 1 cmp, 1 and, 2 ldr, 1 branch
  _credit[ta][0][0] += header_size;                            // 2 add (one for offset), 1 str (for _credit)
  _credit[ta][1][0] += data_size;                              // 2 add, 1 str

  return true;                                                 // Overall hasroom(): worst case: check both if's and update credits
                                                               // 10 add, 2 cmp, 2 and, 4 ldr, 2 str, 2 branch
                                                               // Less than 10 registers
}                                                              // (ta, header_size, data_size, _credit, 4 _credit values)

bool CLASS::push_ta( Packet *packet) {
  // extract packet type and size information
   ...                                                         // 4 add, 1 shift, 1 ldi, 3 ldr, 1 branch

  // posted transactions
  if (type == 0x40 || type == 0x60 || (type & 0x38) == 0x30)
    h = hasroom(0, header_size, data_size);                    // 3 cmp, 2 or, 1 branch, hasroom()
  else // non posted transactions
  if ((type & 0x5e) == 0x00 || (type & 0x3e) == 0x04)
    h = hasroom(1, header_size, data_size);                    // 2 cmp, 1 or, 1 branch, hasroom()
  else // completion
  if ((type & 0x3e) == 0x0a)
    h = hasroom(2, header_size, data_size);                    // 1 cmp, 1 branch, hasroom()
  else {
    h = false; packet-> kill(); return (true);
  }                                                            // Overall push_ta():
  if (h)                                                       // Worst-case: completion transaction
    output(OUT_TA).push(packet);                               // 4 add, 1 shift, 6 cmp, 3 or, 1 ldi, 3 ldr, 4 branch, hasroom()
  return(h);                                                   // Less than 10 registers (packet, type, header_size, data_size, h)
}                                                              // Shareable with hasroom():  header_size, data_size
```

Figure 5.9: Derivation of the instruction histogram for the flow control Click element.

We then follow the different packet-processing paths in our Click models for receiving and transmitting packets, as derived in Section 3.3.1. The result is a histogram of executed instructions together with the number of registers used for each of these cases. The profile also represents the execution time on one processing engine if weighted with the instruction latencies introduced earlier. The execution time under a certain load could also be derived by assuming defined backlog levels in each of the participating queues.

As a result, our method is an estimation of the workload based on a static analysis of the control and data flow graph, extracted from the source code of the Click models.

[1]The example in Fig. 5.9 is listed as *flow ctrl* in Table 5.5.

Execution profile

The profile of executed instructions using the instruction classes introduced earlier for the major Click elements for PCI-Express are listed in Table 5.5.

Table 5.5: Profile for PCI-Express Click elements. The marked elements (*) are realized in hardware.

Instruction class	A	L	ldr	ldi	str	B
Click element/subfunction, 64 Byte data packet						
flow ctrl	14	14	7	1	2	6
Ack/Nack Tx	5	5	1	0	6	0
prio sched	0	1	1	0	0	0
check paint	1	1	1	0	0	0
Ack/Nack Rx	3	2	4	0	1	1
classify	2	1	2	0	0	1
flow ctrl rx	8	8	6	3	5	0
Ack/Nack packet-specific						
AckNackGen	0	0	0	4	1	0
Ack/Nack Tx ack	7	9	2	0	1	1
Ack/Nack Tx nack	4	9	22	1	3	1
Flow control packet-specific (in TaFl)						
flow ctrl update (Rx)	1	4	1	4	0	0
ctrl hasRoom (Tx)	10	4	4	0	2	2
ctrl newCredit (Tx)	8	2	3	0	2	3
common						
Calc CRC*	64	320	144	64	0	0
descr enqueue	2	2	0	0	4	0
descr dequeue	2	2	4	0	0	0
payload enqueue*	3	3	1	1	16	0
payload dequeue*	3	2	16	1	1	0

The profiles listed in the table reflect the computation requirement for one packet. We assume that one packet must be retransmitted if a Nack packet is received and that the priority scheduler only has to check one queue for its size. The requirement for the scheduler listed in the table is low since enqueue and dequeue operations are counted separately. Classifications are simple since they only rely on small bitfields for packet and transaction types so that they can be implemented using table lookup.

Performance Impact

Given the profiling results in the form of instruction histograms for the different processing paths, we now derive execution times by weighting the histograms with the corresponding execution latencies per instruction class. Compared to the cycle counts for the general purpose core, reductions between 39% and 68% are observable, as Figure 5.10 reveals. The figure shows the absolute and relative cycle counts per processing path. For the comparison, identical sets of hw accelerators (CRC, DMA) and the same IO overhead were assumed.

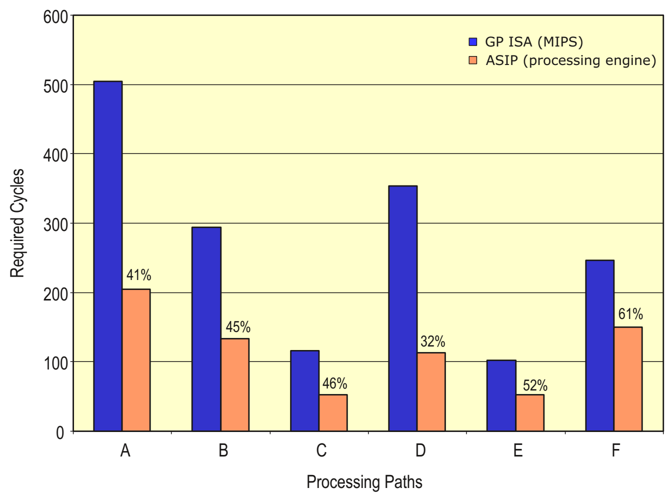


Figure 5.10: Comparison of the required cycles for general purpose and application-specific ISA for PCI Express.

For the ASIP, Case A - the transmission of transactions - remains the most cycle-consuming case. The relative cycle reduction of about 60% directly translates into a reduction of the required clock frequency as Figure 5.11 shows.

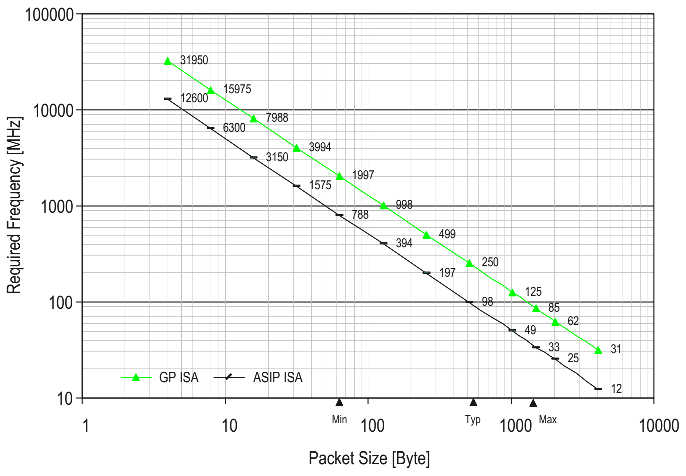


Figure 5.11: Impact of an application-specific ISA on the required clock frequencies for PCI Express.

5.2.4 Multiple processors

Multiple processing elements can help to meet performance requirements by exploiting task- and data-level concurrency of an application. This is especially observable in the field of network processor architectures which is known for the deployment of parallel cores in varying topologies for highest throughput requirements [57]. This subsection applies the general principles observed in network processors to the particular domain of communication interfaces by investigating the impact of multiple processors on the performance.

Multiprocessor setup

For the exploration, we assume an allocation of homogeneous processing elements. Similar to the baseline architecture (cf. Figure 5.1), each processing element has its own instruction and data memory, and a set of transfer queues for the inter PE communication which supports direct communication between two processing elements. Thus, we do not depend on SRAM or scratchpad memory, since the message passing communication between tasks is either mapped to direct connections between two engines or processed internally if the corresponding tasks are mapped to the same engine.

The surrounding SoC and PHY system remains the same as in our baseline architecture but incorporates a DMA coprocessor for the transfer from packets between PHY and packet memory. The coprocessor has a similar set of transfer queues to participate in the message passing communication. Access to the packet memory will be possible for the PEs but is not required since the packet descriptor contains all of the relevant data (i.e., processing state and packet header).

The topology of multiprocessor setups can be characterized using two axes, namely the number of processing elements per stage in a pipeline and the number of pipeline stages. In network processing, which is tailored to packet forwarding, we find all varieties of topologies ranging from fully pipelined to fully pooled solutions. We, therefore, assume an interconnection network which can support arbitrary topologies and enables the communication between any of the processing elements including the DMA coprocessor.

To separate the effects of computation and communication, the interconnect network is assumed to be ideal. Transfers do not consume time and the network is always available. In a separate exploration step (Section 5.2.5), the effects of transport delay and resource sharing are considered.

Expected performance speedup

Multiple processors can help to reduce the required clock frequency for the deployed cores by distributing the load (in settings with given throughput and processing requirements). For the following analysis, we, therefore, define the performance gain as the ratio of the required clock frequencies for single (f_s) and multi-core systems (f_i):

$$Speedup = \frac{f_s}{\max_{i=1..N} f_i} \tag{5.1}$$

The single-core frequency f_s is the product of the required bandwidth (BW_{IO}) and the number of cycles to be executed (C_{IO}). If an application can be partitioned evenly and mapped without overhead for communication and synchronization among N cores, the speedup is ideal and equals the number of processing cores:

$$Speedup = N \quad \text{with} \;\; f_i = \frac{BW_{IO} \, * \, C_{IO}}{N} \;\; \text{and} \;\; f_s \; = BW_{IO} \, * \, C_{IO} \tag{5.2}$$

Such an ideal speedup would be achievable in pooled settings, in which every core executes the complete and statically assigned task graph for a stream of data tokens independently of other cores without sharing of processing state. However, such settings are limited to cases where the computation requirement of the task graph is below the inter-arrival time of packets in the data stream. In the more general case, some degree of pipelining is necessary, e.g., to classify and distribute data (streams) among the processors or to match the inter-arrival time of packets within the stream.

Taking into account the overhead for communication of packet descriptors C_{PD} in a pipeline of processing elements with N stages and the overhead for scheduling and arbitration on each processing element C_S, the achievable speedup becomes limited and saturates:

$$Speedup = N * \frac{C_{IO}}{C_{IO} + (N-1)(C_{PD} + C_S)} \tag{5.3}$$

$$\lim_{N \to \infty} Speedup = \frac{C_{IO}}{(C_{PD} + C_S)} \tag{5.4}$$

This behavior is shown in Figure 5.12 which plots the calculated speedup for PCI Express, transmission of data packets, considering communication and scheduling overhead. The non-linear scheduling overhead depends on the number of tasks mapped onto a core and therefore ranges from 68 instructions (8 tasks, 1 core) down to 19 instructions (1 tasks, > 7 cores). The communication overhead is 20 instructions.

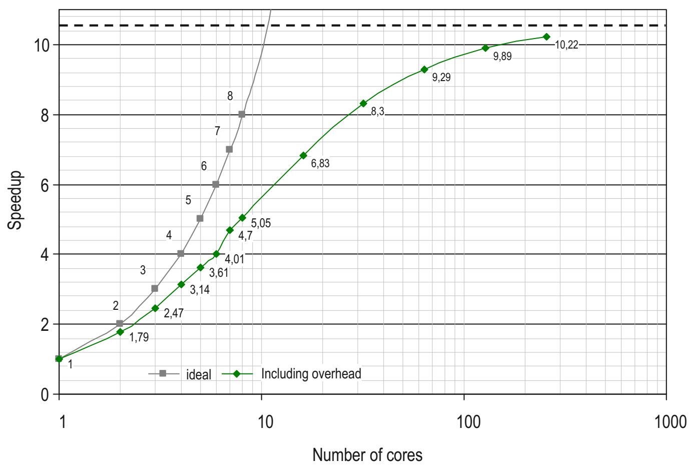


Figure 5.12: Theoretical speedup for PCI Express (Path A, data tx on MIPS cores) considering a) an ideal pooled setup (Eq. 5.2), and b) communication and scheduling overhead for a pipelined setup (Eq. 5.3).

The performance gain in the figure is calculated under the ideal assumption that the single task graph can be well-balanced even for high core numbers. Now, the challenge is to extract sufficient concurrency from the application and to take the effects of multiple task graphs into account.

Application concurrency

The packet-oriented communication interfaces, as modeled and discussed in Chapter 3.3, exhibit only a limited degree of data- and task-level concurrency compared to typical multi-ported packet-forwarding applications (such as the ones described, e.g., in [57, 174]). In particular, we observed:

- A limited number of concurrent packet flows — Multiple physical IO ports would be a natural source for data-level parallelism since each of them handles independent packet flows. Communication interfaces, however, have exactly one physical IO port with, at best (full duplex), two truly concurrent packet streams, receive and transmit. Even though a small number of logically independent streams exists at the transaction layer (such as the three quality-of-service classes in PCI Express), multiplexing and joint stateful processing of these streams is required at the link layer.

- Fair task-level concurrency — The different processing paths of the communication interfaces as identified in Chapter 3.3 represent independent task chains which may execute concurrently. Each of the processing chains comprises several Click elements thus enabling pipelining. In case of PCI Express, for instance, between 3 to 6 elements are instantiated.

These observations indicate that only settings with small numbers of processing cores are meaningful for the interfaces. In the following, we will, therefore, focus on setups between one and seven cores. We focus on the PCI Express protocol, which has the highest throughput requirements and is latency insensitive. Only at the end of this section shall we come back to wireless Lan and report the results of its exploration.

Application setup and partitioning

For the performance analysis of the PCI Express model, we configure the simulation for the transmission of a continuous stream of packets between to interfaces. For proper operation, two PCI Express lanes are required. The first, from if1 to if2 transmits the data packets while the second, from if2 to if1, is used for returning acknowledge and flow control credits.

Following our CRACC mapping approach as described in Section 4.5, we partition the application at element boundaries as shown in Figure 5.13 and map the subgraphs to different MP configurations. The longest processing path is partitioned into two subgraphs at the border between TA and DL layer. The inter-PE communication links are configured to be ideal, representing, for instance, dedicated point-to-point connections.

Performance impact

To study the impact of the concurrent receive and transmit processing tasks for a duplex PCI Express interface, we start with a single processor mapping (named 1CPU). In this case, a CPU frequency for if1 is required that is 18% higher than for the data transmission, only. Running at the same clock frequency as if1, the core of if2, which receives the data and generates the feedback is utilized by 92%, i.e., the required frequency is 6.7% higher than the pure data tx case.

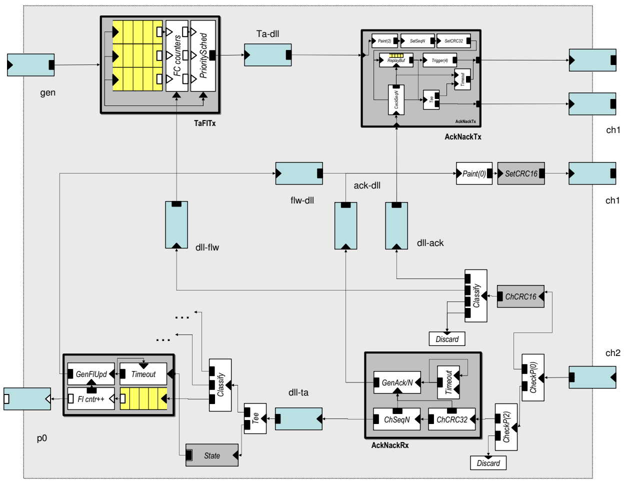


Figure 5.13: Partitioning of the PCI Express model for the MP simulation.

This data is derived from Figure 5.14 which shows the utilization of the cores labeled "1CPU.tx" for if1 and "1CPU.rx" for if2 to be reasonably well balanced.

In the next step, we map the transaction and the link layer on two different processors (2CPU). In this case, an overall speedup of 1.44 is achieved for the combined processing on if1, which is less than expected. Figure 5.14 reveals the DLL layer CPU (CPU.tx.2) to be the bottleneck in the system. The CPU is fully utilized but only 80% of the load are caused by the transmit task. Almost 20% of the load are due to the reception of link layer feedback packets, i.e., acknowledgements and flow control credits.

To relieve the DLL layer CPU, we now consider a four core mapping (4CPU) for the interface by separating both receive and transmit parts of both layers. We do not expect much utilization for the two receive cores of if1, since our traffic pattern only transfers data from if1 to if2. Vice versa, if2 should not utilize the transmit cores much. This expectations are confirmed by Figure 5.14, cases 4CPU.tx.1-4 for if1, and 4CPU.rx.1-4 for if2. Interestingly, we achieve only a small reduction of the rx load on core #2 which leads to a slightly improved speedup of 1.51. The remaining rx load is caused by the processing of received acknowledges within the AckNackTx element, which is also part of the Tx processing task chain.

To consider a fully loaded 4-core system, we finally use a symmetric traffic setup in which if1 simultaneously sends and receives packets. The overall speedup in this case is 1.6. CPU #2, again, is fully utilized and remains the computational bottleneck (cf. Fig. 5.14). The other cores are between 70% and 90% utilized.

In all mapping variants, the overall speedup is dominated by the partitioning of the most compute intense task chain, the transmission of data packets. Table 5.6 shows this in detail. The monolithic AckNackTx element consumes 205 instructions and basically is the only element on CPU 2. Further performance improvement therefore requires optimization of this bottleneck by one of the following options:

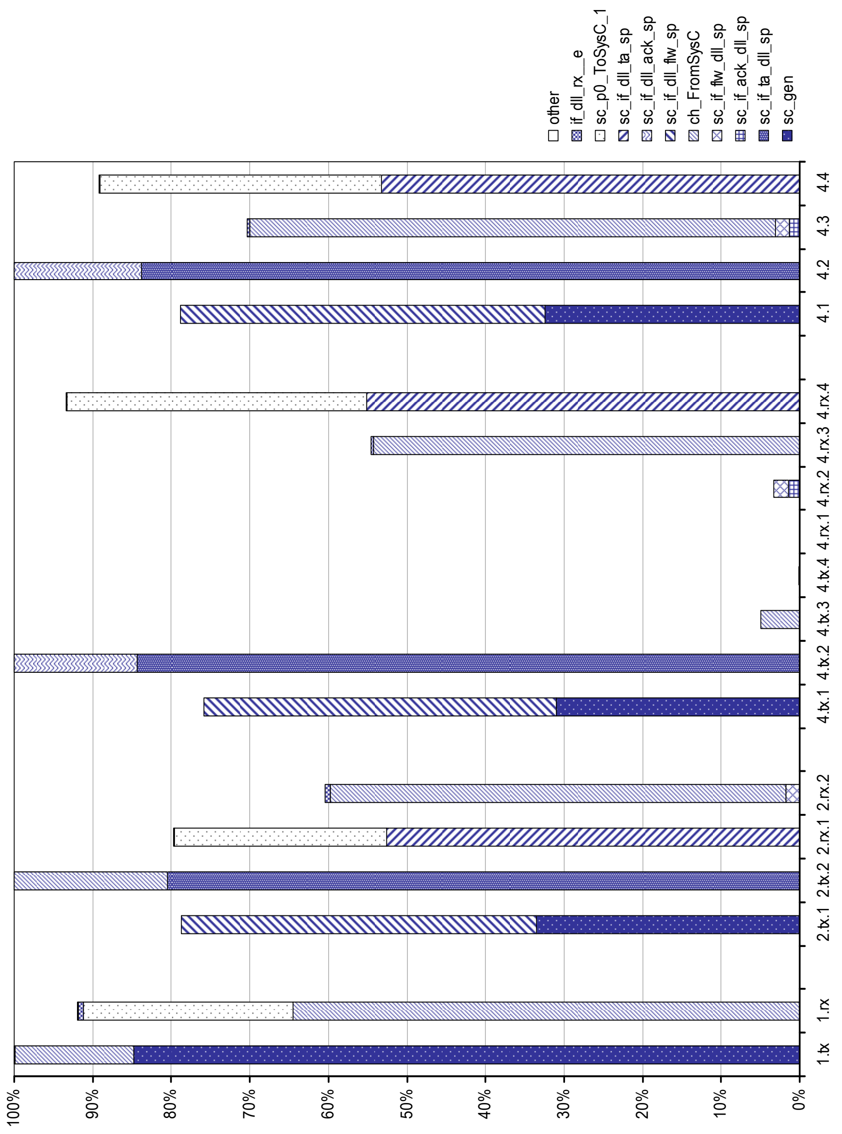


Figure 5.14: Utilization for different PCI Express mappings.

- Optimize the execution performance — The current element implementation is functionally correct but is not optimized for processing speed. Although not in the focus of this section: Code optimizations, e.g., by coding critical parts in assembly, should be considered after careful performance analysis. The detailed performance analysis of the AckNackTx element, for instance, revealed potential for a less expensive timer function in settings with only a small number of timers on a core. This actually small change would improve the speedup for the four core mapping to 1.85, as calculated in column 4.2' of Table 5.6.

Table 5.6: Partitioning of the most critical PCI Express task chain.

Mapping/core No. of chains	1 7	2.1 5	2.2 5	4.1 3	4.2 3	4.2' 3	4.1" 3	4.2" 3
Read PD	10	10	10	10	10	10	10	10
schedule chain	61	47	47	33	33	33	33	33
TaFlTx.push	51	51		51			51	
Paint	29	29		29			29	
AckNackTx.push	135		135		135	135	35	100
Write PD	10	10	10	10	10	10	10	10
AckNackTx.timer	70		70		70	35		70
TaFlTx.pop	30	30		30			30	
Dequeue	17	17		17			17	
Total [inst]	413	194	272	180	258	223	215	223
Speedup	1	2.13	1.52	2.29	1.60	1.85	1.92	1.85

- Partition critical element, optimize mapping — Partitioning of large elements eases the balancing of pipeline stages. In case of the AckNackTx element and its transmit path three only loosely associated parts can be identified[2]: set sequence number (35), enqueue & push (100), and reschedule timer (70/35). A separate SetSeqN element, for instance, which is mapped onto tx.1 would lead a speedup of 1.85 as well. If combined with the timer optimization above, the other CPU becomes the bottleneck resulting in a speedup of 1.92 for the transmit path. Instantiating an additional CPU in this path would increase the speedup to 2.19 (see the 5-core case in Table 5.7).

- Replicate element, distribute load — Additional instances of compute-intense elements can be mapped onto different processors to distribute the load. This solution, however, may depend on issues such as sharing of processing state or the sequence of token processing. The AckNackTx element could be duplicated and mapped onto two processors. An upstream element would distribute packets to both instances in alternating order so that each instance handles every other packet, a downstream multiplexer gathers them and enforces the correct sequence. Received acknowledges would need to be sent to both instances so that each one can update its state individually. Table 5.7 shows this mapping in more detail for a 5-core (7cpu) and a 4-core (6cpu) setting.

The speedup calculations in Table 5.7 assume that the initially critical path remains most critical in all cases. Even with this assumption: the refined multi-core mappings with up to seven cores do not provide much additional speedup compared to the four-core mapping.

We therefore continue with the four-core setting, which provides – for the given application model and optimization level – a speedup of 1.85. In symmetric traffic scenarios, the system utilizes its cores almost fully (86%, 100%, 94%, 99,9%). Considering the packet size, clock frequencies between 1270 MHz (64B) and 55 MHz (1512B) are required in order to meet the throughput requirements of a full-duplex PCI Express link (2x2 Gb/s). The iMix packet size distribution requires a clock frequency of 232 MHz.

[2]Setting the CRC would be a fourth part but this function is handled by hardware anyway.

Table 5.7: Refined partitioning of the most critical PCI Express task chain using 3, 5, and 4 cores in the transmit path.

Core	5.1	5.2	5.3	7.1	7.2	7.3,4	7.5	6.1	6.2	6.3,4
Read PD	10	10	10	10	10	10	10	10	20	10
schedule chain	33	33	33	33	33	33	33	33	66	33
TaFlTx.push	51			51				51		
Paint		29				29				29
Distribute*					35				35	
AckNackTx.seq		35				35				35
AckNackTx.push			100			100				100
Multiplex*							35		35	
Write PD	10	10	10	10	10	10	10	10	20	10
AckNackTx.timer			35			35				35
TaFlTx.pop	30			30				30		
Dequeue	17			17				17		
Total [inst]	151	117	188	151	88	126	88	151	176	126
Speedup	2.73	3.53	**2.19**	**2.73**	4.69	3.28	4.69	2.73	**2.34**	3.28

*) estimated

Wireless Lan on concurrent processors

A single-core solution with hardware accelerators would be sufficient to support our 802.11 model in terms of its throughput requirements. This section thus focuses on meeting the realtime requirements.

In general, the problem of missed transmission deadlines can be addressed by different approaches. First, the granularity of Click tasks and their scheduling could be modified as discussed in the section above. Large Click elements can be partitioned to reduce worst-case resource blocking time. In addition, the run-to-completion semantics of Click could be changed to preemptive/prioritized scheduling at the expense of extra overhead[3]. This is a serious change in the underlying model of computation and will thus not be followed here.

Second, resource allocation and mapping can be modified so that interference is reduced. The results of the preceding subsection indicate that it will be beneficial to either allocate different processing resources to receive and transmit paths or mapping time-critical elements to a separate resource.

Figure 5.15 shows the results for the latter option in terms of a response-time histogram. Mapping time critical rx/tx functions (basically the link layer of the model including the tx queues) onto an extra processor leads to reduced response times. We even applied slack to reduce the processing speed of the two cores. For the same throughput, the non-realtime CPU runs at 150 MHz, while the real-time CPU requires slightly more than 200 MHz.

5.2.5 Communication Topology

In the previous section we studied the impact of multiple processors on the system's performance taking into account the *computational* overhead of the

[3]Comparing different OS Samadi et al. report at least 100 cycles [165].

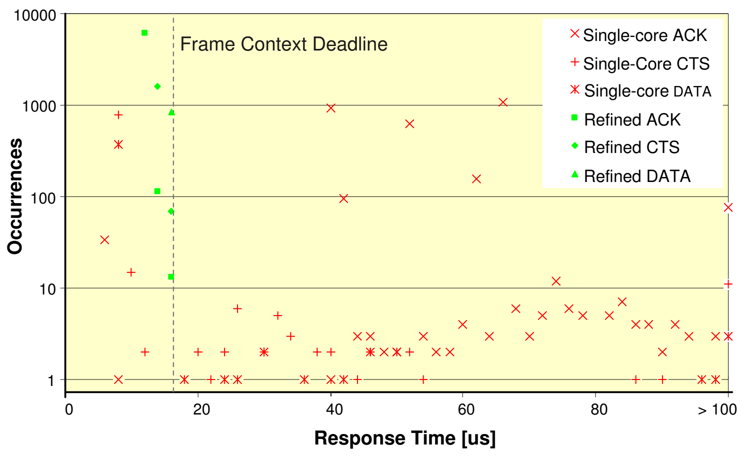


Figure 5.15: Response time distribution for the single CPU (400MHz) and refined dual CPU (150/200MHz) mappings.

inter-PE communication but assuming an ideal communication network otherwise. In this section, the effects of transport delay and shared communication resources will be considered. In addition, we will study the queueing requirements for the exchange of messages between PEs.

Communication Setup

For the analysis, we use the PCI Express 4PE mapping as discussed in the previous section. The packet descriptor communication between Click elements that are mapped onto different processors is implemented using an interconnection network, which enables the lossless exchange of messages between any of the processing nodes. For each communication link, the PEs implement sets of out- and inboxes, which effectively decouple the processors from the communication. Lossless operation is guaranteed by back pressure which stalls the sending processor until space becomes available in its outbox (blocking write). An outbox only requests transfers from the communication network if the next inbox has sufficient space available.

Given the small number of participating nodes, all of them can be connected by a single write-only bus. A complex communication network is not required. This means that each processing node is both, a master that sends messages and a slave that receives them. Only one message can be in transfer at any given time. A transfer is assumed to consume 16 bus cycles which is sufficient for packet descriptors up to 64 Byte. The bus speed as well as the sizes of the in- and outboxes varied in the analysis.

Performance Impact

To study the sensitivity on these parameters, we vary the bus speed in reference to the frequency of the cores from 10 to 100% (10 steps), and the size of the in- and outboxes entries from 1 to 8 (6 steps each:1-4,6,8) per communication link. As performance metric, we examine the relative change of the packet throughput for each design point.

The relative throughput for the initial 4PE mapping (4CPU) is shown in Figure 5.16 which lists each of the 360 design points along the x-axis. The throughput is normalized to the one of the best mapping (4CPU'). This means that a maximum of about 89% is achievable. Most of the design points with a bus frequency of 30% or higher reach this performance (cases 72 to 320). The bus utilization decreases with speed starting at 64%. The 10% bus frequency group limited by the fully utilized bus only reaches 46,5%. Only the 20% bus frequency group shows significant differences depending on the fifo sizes. The

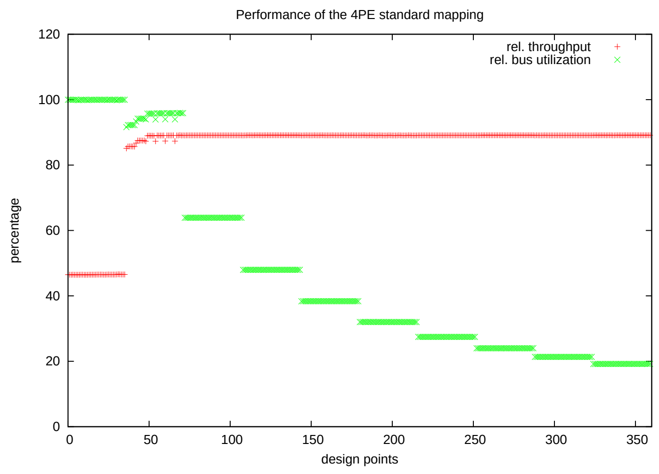


Figure 5.16: Relative throughput and bus utilization for all design points of the 4PE standard mapping. The design points are ordered by {bus freq, inbox size, outbox size}.

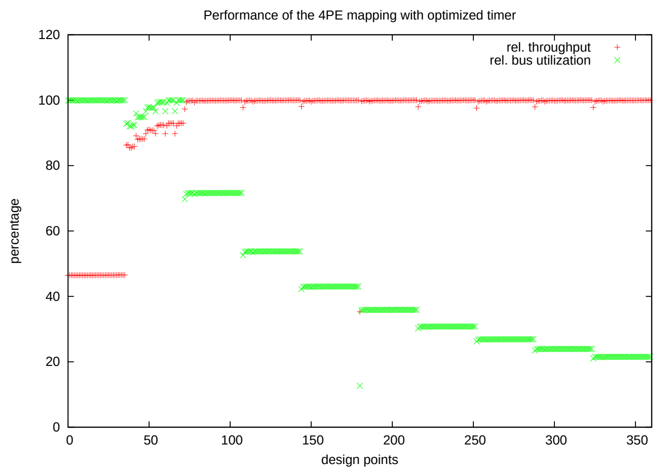


Figure 5.17: Relative throughput and bus utilization for all design points of the optimized 4PE mapping. The design points are ordered by {bus freq, inbox size, outbox size}.

optimized mapping 4CPU' is shown in Figure 5.17. It achieves an 11% better performance for those cases that are not limited by the bus speed, i.e. run at 30% or more of the core frequency. The price is a higher utilization as the comparison of the two figures reveals. At 10% bus speed, the setting is limited the same way as before, reaching only 46,5% throughput. The 20% group's performance increases with the amount of storage in the system.

The dependency of the performance on the size of the in- and outboxes is studied in more detail in Figure 5.18. Even for the highly utilized 20% case (Fig. 5.18, left), the standard mapping does not depend on the queue size much. Overall a outbox size of 3 and inbox sizes of 2 would be sufficient. The optimized mapping would benefit from sizes that are slightly increased (4,3 – cf. Fig. 5.18, right). For bus speeds of 30% or better, combined sizes of 3 for in- and outbox are sufficient to avoid the performance degradation of the (1,1)-size case as can be observed in Fig. 5.17.

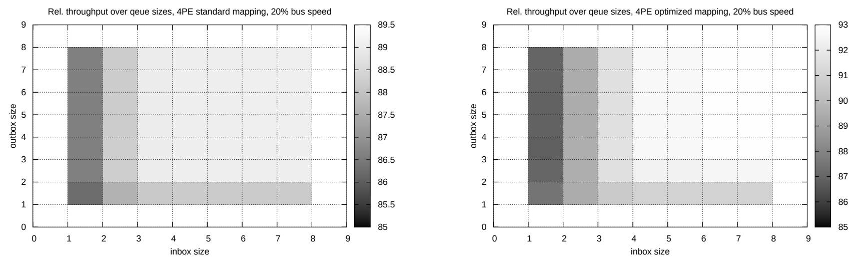


Figure 5.18: Relative performance over the size of in- and out boxes for a bus speed of 20%, shown for standard (left) and optimized (right) mappings.

To further detail the communication requirements of the application-architecture mapping, we conducted an in-depth analysis of the backlog in the communication queues. For this purpose, the in- and outboxes are bound to 128 entries, the application's transmit and receive queues are set to 256 entries. Due to the throughput-oriented simulation setup, the transmit queue (app-tx) is kept full, while the receive queue is emptied as fast as possible (app-rx).

The results are shown in Figure 5.19. Interestingly, the backlog in the 4CPU system reveals, that only one communication queue actually accumulates backlog: the inbox on core#2 (queue ta-dll-i, top-left diagram of Fig. 5.19). In the mapping considered, this core is fully utilized and becomes the bottleneck of the communication pipeline. A design point, which is limited by the bus speed, is shown in the middle row of the figure for the optimized mapping (4CPU', 20% bus speed). In this case, the outbox to channel one (ch1-o) acquires some backlog but queue levels are otherwise steady. The last design point, shown in the bottom of the figure, removes the highly utilized bus as bottleneck (bus speed 30%, 4CPU'). In this case, backlog accumulates in the inbox of core#4, which is the bottleneck of the optimized mapping, while the initial backlog in front of core#2 decreases.

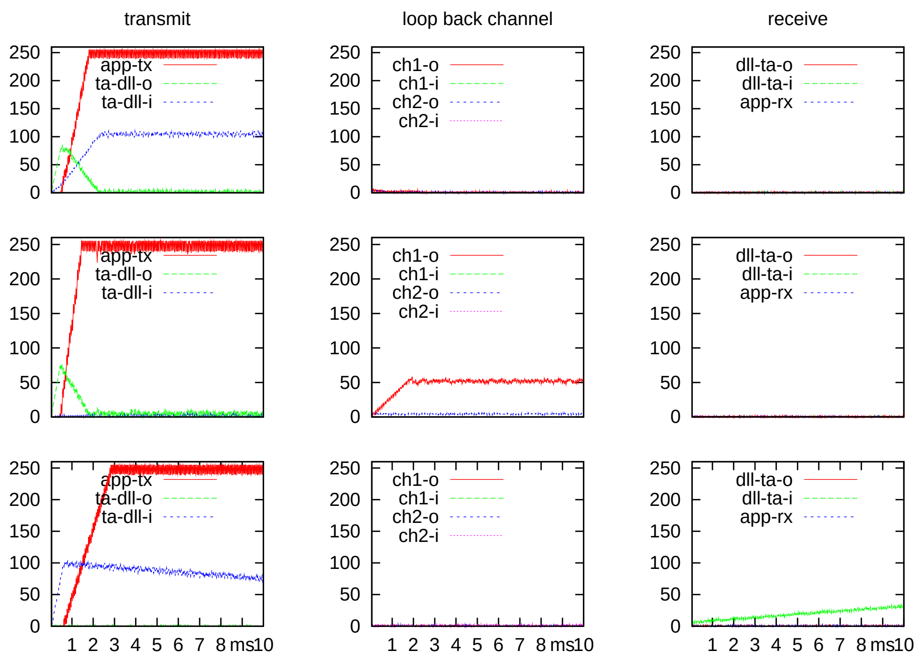


Figure 5.19: Backlog in in-and out-boxes along the communication path for different design points: Top) Standard 4core mapping, 20% bus speed, Middle) optimized mapping, 20% speed, Bottom) optimized mapping, 30% speed.

5.3 Proposal for an optimized architecture

The previous section explored several dimensions of the architectural design space and studied the impact on a systems performance. These pieces will be put together in this section which proposes an optimized architecture for the implementation of the packet-oriented communication interfaces.

The emphasis of the architecture is still on flexibility and programmability. This means that wherever possible, we choose software over hardware, and general purpose over protocol specific functions.

5.3.1 Block diagram

Figure 5.20 shows the architecture of the optimized system which is based on our NOVA hardware platform (introduced in Section 4.6). Accounting for the results of the exploration, the system deploys four general-purpose processing elements and a specialized co-processing element comprising the identified hardware accelerators for data transfer and CRC calculation. All processing elements are encapsulated by NOVA sockets, which provide interfaces to the message passing network and to the memory hierarchy ('memory bus'). The system exploits the concurrency in protocol and payload processing by separating both facets. The protocol processing functions are mapped onto the general-purpose processing elements, while data streaming and CRC calculation are provided by the specialized co-processor module.

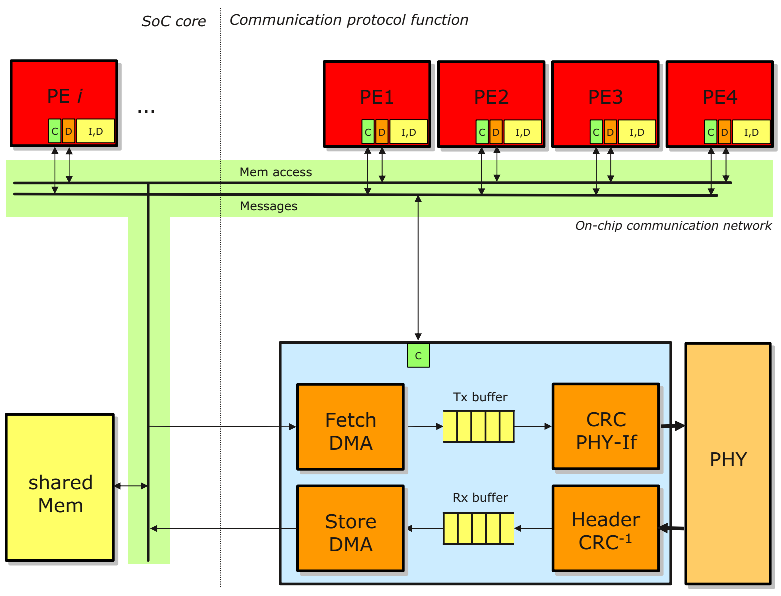


Figure 5.20: Block diagram of the optimized 4-core architecture

In transmit direction, control and data flow through the architecture as follows. An outbound transaction is stored in the shared memory. The SoC core sends the packet descriptor to one of the interface PEs. The core reads the descriptor and processes the packet according to its program. For pipelined operation, the packet descriptor is passed to the next processing node. Eventually, the protocol processing is completed, and a descriptor is sent to the coprocessor node, which – configured by the PD's sideband information – reads the packet data from the memory and streams it towards the PHY interface, calculating and inserting the CRC on the fly. In receive direction, the coprocessor node receives a stream of packets from the PHY and stores them in the shared memory using its DMA function. In addition, the module calculates the CRC and generates a descriptor for each packet, which is forwarded to one of the PEs, e.g., #3. On the PE cluster, the receive protocol processing takes place and received transactions are forwarded accordingly.

For smooth operation, all the communication between the processing nodes is based on message passing. The in- and outboxes on each node effectively decouple them and may compensate temporary rate differences. Within the coprocessor, the same principle is applied to decouple the packet streams from and to the PHY from memory accesses. The function of the hardware building blocks is explained in more detail next.

5.3.2 Hardware building blocks

On the SoC level, the optimized 4-core architecture actually breaks into two building blocks, a general purpose processing element and a specialized processing element. As other SoC modules, these building blocks are integrated and connected using the NOVA socket, which provides interfaces to access (shared) memories and the message passing network.

General-purpose Processing Element

The general purpose PE comprises a 32bit MIPS M4k processing core with local instruction and data memories. In addition, a memory interface, a tiny DMA function, and the message passing interface are integrated into the module.

- M4k core — the core is configured with a dual SRAM interface in order to execute instruction fetches and load/store accesses in parallel. Its register set comprises hardware counter and timer registers.
- Local memories — Two single-cycle, single-port SRAMs are used for instruction and data memory. A small boot rom is mapped into the imem range. Subsystem functions are mapped into the dmem map and can be accessed via load/store instructions. Using the redirection feature of the core, the imem can be accessed via load/stores as well.
- Memory Interface — The memory interface translates the core's blocking load/store accesses into pipelined split-transaction memory transfers. This means that writes to distant memories can be completed within one cycle while reads stall the core until the data becomes available.
- Message Passing Interface — The interface provides multiple (2-4) sets of hardware in- and outboxes for the transfer of messages with different priorities. Queue status and data ports can be accessed by the core in addition to the DMA.
- Tiny DMA — The DMA engine relieves the core from cycle-consuming data transfers. It may be used a) to transfer bulk data from distant to local memory (memcopy), and b) to move messages between local mem and message passing interface. To avoid stalls, the DMA has lower priority than the core.

Specialized Processing Element

The specialized processing element encapsulates data movers and stream processing units. In particular, it comprises:

- Fetch DMA — The Fetch DMA receives a packet descriptor, extract packet descriptor and processing flags, fetches the packet from its storage location, and streams it towards the transfer fifo. In the process, it inserts a stream header, which contains processing flags and packet length, for downstream processing units. In addition, it may insert extra data (the packet header) from the packet descriptor in front of the packet. The Fetch DMA may be configured send a free message after completion of the transfer.
- Store DMA — The Store DMA takes a packet from the receive fifo, removes the stream header, and stores it the packet memory. For this purpose, it uses pointers to free packet memory that are received at its inbox. It may be configured to send a message after completion.

- Receive and transmit buffers — In transmit direction, the buffer should store sufficient amounts of data to avoid underruns during the transfer of individual packets. In receive direction, it is used to compensate the processing latency for the on-the-fly allocation of packet storage.
- Tx-CRC unit — The CRC unit calculates the CRC for the data stream on the fly. Its operation can be configured per-packet using the stream header. Options include: type of CRC (polynom, width), bypass/append/overwrite, ignore N initial bytes of the packet. In addition, the module handles the frame alignment (off, pad if lower than N byte, pad to multiple of N byte). If required, precise PHY timing is enforced (send at timestamp).
- Rx-CRC and descriptor generator unit — In receive direction the unit calculates the CRC while the data is written into the receive buffer. Its operation can be selected per-packet using a simple classification of the first bytes of the packet. In addition, the unit creates a packet descriptor, stores processing flags (including a timestamp), and copies a configurable number of initial bytes (the packet header) into it. To avoid storage overhead, e.g., in case of the PEX link layer packets, the unit can be configured not to forward small packets of certain classes which fully fit into the packet descriptor.

5.3.3 Synthesis results and cost estimation

We implemented a 4PE prototype of the NOVA platform. Its block diagram is shown in Figure 4.11. The system implements MIPS based processing elements, dedicated hardware Ethernet IO modules that are connected to external PHYs, a statistics coprocessor, and a memory interface for shared off-chip SRAM memory. The on-chip communication is based on three OCP buses for system messages, packet descriptors, and memories. In this way, high priority delivery of system messages is assured. Based on the prototype, we evaluate the resource requirements of the optimized solution. For this purpose, we will report the synthesis results for a 90nm ASIC design technology in the following[4].

ASIC implementation

The processing element of the prototype shown in Figure 4.10 comprises most of the intended PE functions. It provides interfaces for memory bus and message passing network, implements dedicated in- and outboxes for two message priorities using single-cycle dual port SRAMs, provides single-port instruction and data memories of 8KB each, boot rom, interrupt controller, and so on.

In the 90nm technology chosen, the processing element achieves a clock frequency of about 360 MHz. The frequency is limited by the MIPS core itself. The most critical path runs within this IP core. There is almost no variance with operating conditions (357 MHz, WorstWorst - 361 MHz BestBest). The area requirements for the general-purpose PE are listed in Table 5.8. For the configured memory sizes, the prototype PE requires .8114 mm^2. Most of the area (.556 mm^2, 68%) is occupied by memories. Adding the tiny-DMA function

[4]To proof the concept, the prototype was also mapped onto a FPGA device (Xilinx XC2V6000-4) and successfully demonstrated in a real-world Ethernet LAN setting.

requires an extra address decoder and data multiplexer, a 2nd port on each client (data memory, MP and mem interfaces), and the DMA logic itself. These modules increase the PE area by about 8%. On the other hand are the transfer queues currently implemented using dedicated dual port memories for each of them. Mutually exclusive access in combination with the DMA would enable the optimization of these queues into one instead of four dual-port memories, which reduces the PE size to .7161 mm^2.

Table 5.8: Area requirements of the General Purpose PE.

General Purpose PE sub-module	area [mm^2]	memory [%]	remarks
MIPS core	0.2079		w/o fast mply
data mem	0.1677	99.90	8kB, 1x32x2048 SP
inst mem/boot rom	0.1710	97.90	8kB, 1x32x2048 SP
memory interface	0.0047		
MP Interface 0	0.1356	88.80	512B, 2x32x128 DP
MP Interface 1	0.1138	88.60	128B, 2x32x32 DP
other logic	0.0107		e.g., intc, data mux
tiny-DMA function*	0.0657		4 channels
optimized MP if**	-.1610	100	only 1x DP mem of 512B
GP PE total	0.7161	55.18	

*) estimated based on IO DMA function, **) area of removed memories

The estimation of the special-purpose PE is based on the encapsulated hardware Ethernet IO module of the prototype. This is why we report its footprint first. The IO module achieves operating frequencies of 357 (WorstWorst) to 401 MHz (BestBest) for the system clock. Table 5.9 lists its area requirements. It, too, is heavily memory-dominated and has roughly the same area footprint as one of the prototype PEs.

Table 5.9: Area footprint of the NOVA Ethernet IO module.

Ethernet IO module sub-module	area [mm^2]	memory [%]	remarks
FE/GbE Mac core	0.5813	93.9	
Memory Interface	0.0050		
DMA function	0.0055		
DMA control/cfg	0.0047		
Parser	0.0724	83.10	512B, 1x32x128 DP
Unparser	0.0662	90.90	512B, 1x32x128 DP
Message Controller	0.1175	85.80	128B, 2x32x32 DP
Ethernet IO total	0.8480	90.90	

The specialized PE reuses most of the IO modules socket functions. For truly concurrent operation, its memory interface is duplicated in receive and transmit direction. Thus, the DMA function is required twice, and the memory interface is partially duplicated (x1.5). Parser (packet descriptor generator, rx path), unparser (packet descriptor interpreter, tx path), message controller, and DMA configuration modules remain the same. The pure CRC calculation logic in hardware is almost negligible, the modules, however, require some pipeline stages similar to a unidirectional memory interface and configuration and state registers. The streaming buffers are fifos with dedicated control logic. Due to the moderate clock frequencies at the PHY layer they are implemented using single port memories. These estimations all together lead to an area requirement of 0.442 mm^2 for the specialized processing element, as Table 5.10 summarizes.

Table 5.10: Area requirements of the specialized PE module.

Specialized PE sub-module	area $[mm^2]$	memory [%]	remarks
Memory Interface*	0.0075		
Fetch DMA	0.0055		IO DMA function
Store DMA	0.0055		IO DMA function
DMA control/cfg	0.0047		
Parser	0.0724	83.10	512B, 1x32x128 DP
Unparser	0.0662	90.90	512B, 1x32x128 DP
Message Controller	0.1175	85.80	128B, 2x32x32 DP
Tx Stream Buffer	0.0580	93.16	2KB, 33x512 SP
Rx Stream Buffer	0.0900	95.59	4KB, 33x1024 SP
Tx CRC*	0.0024		0.5x MemIf
Rx CRC*	0.0097		0.5x MemIf + cfg
Specialized PE total	0.4418	82.69	

*) estimated based on prototype modules

In total, the optimized 4-PE architecture requires an area footprint 0.4418 + 2.8644 = 3.3062 mm^2. The extra area required for the on-chip communication network was less than 1% (or .0496 mm^2) in our prototype implementation with 8 participating nodes. Considering the smaller number of nodes and the O(N^2) complexity, the interconnect area would be approx. 0.0279 mm^2 for the architecture setup considered.

The overall footprint is dominated by the different instruction, data, and transfer memories, which require 58% of the area in the given configuration. An optimization of the SPE interface queues similar to the GPE could save about 5% of the overall area. Sizes of 8kB for code and 8kB for data memory per PE are sufficient to support all protocols except wireless Lan. Increasing the dedicated code and data memories by a factor of 2 (4) would increase the overall area by 26% (76%), see Table 5.11. Section 5.3.4 will discuss other options in more detail.

Table 5.11: Area requirements for different sizes of code and data memory, absolute in mm^2 (and normalized).

data	instruction memory		
memory	8 kB	16 kB	32kB
8 kB	3.3341 (1.00)	3.7621 (1.13)	4,5941 (1.38)
16 kB	3.7621 (1.13)	4.1901 (1.26)	5.0221 (1.51)
32 kB	4.5941 (1.38)	5.0221 (1.51)	5.8541 (1.76)

5.3.4 Performance & further optimization potential

.

The current optimized 4-PE architecture follows the principles of our design methodology in terms of strict hard and software modularity and high-level programmability. The architecture is flexible in large parts due to the general-purpose PEs and does not comprise any protocol specifics. Even the specialized PE (which is not flexible) is protocol agnostic.

Given the 90nm ASIC system clock frequency of 360 MHz and a message bus frequency of more than 120 MHz, the 4-PE solution is able handle PCI Express at full duplex line speed, i.e. 4Gb/s throughput, for packets of 206B or larger which includes the iMix packet size distribution. Smaller packets can be handled at lower throughput rates. Minimum size packets of 64 Bytes, for instance, can be handled up to line rates of 1134 Mb/s. This performance is sufficient to support application setups such as a wireless LAN access point which requires aggregated throughput rates in the range of 300 - 600 Mb/s.

However, to support small packets at full speed, further improvement of the performance is required without increasing the area footprint. Leaving obvious options aside such as faster technologies and a full custom PE design (see, e.g., Intel's micro-engines, which run at frequencies up to 1.4 GHz) and focusing on improvements of the platform architecture, several options can be identified:

- Code and data memory footprint — Code and data memory requirements depend on application, mapping, and optimization effort. Sizes of 8kB for code and 8kB for data memory per PE are a reasonable lower bound for individually assigned memories since they are sufficient to support all protocols except wireless Lan.

  The PCI Express system function, for instance, as implemented by our model requires 11.5kB runtime code for protocol (7.4 kB) and os functions (4.1kB). This means that the 4PE mapping utilizes the code memories by 78% since most of the OS function is required per PE. The PEs data memory of 8kB each is approx. 33% utilized by stack, element configuration, and program data. The remaining space serves as heap. The size is sufficient for the protocol operation and allows, e.g., to queue more than 80 packet descriptors of 64B locally per PE.

  The wireless LAN system function exceeds the 8 kB code memory space. In its current implementation, the 802.11a/b/g+e station requires code sizes of up to 22.5 kB (single PE mapping, incl. 6kB os functions, excl. management functions). The dual core mapping which separates TA and DL layer can be squeezed in 2x 16kB of code memory (89% and 96% utilization). The data memory is utilized up to 48%, which leaves space for more than 64 packet descriptors of 64B per PE.

  The PEs have access to the shared packet memory, which can be used as overflow storage for code and data. Access to this memory is slow but the tiny-DMA function enables explicit data object and code-page fetching. But even in cases where the fetching time can be hidden by the processing of the core, some software overhead is required due to the DMA handling.

  Another option would be a sharing of code memory between PEs to more flexibly assign and better utilize the memory. Pairwise organized memory banks, for instance, enable a continuous 16kB code memory for the dual PE wLAN mapping without increasing the overall area footprint. In this case (no shared code), there is no performance penalty. However, concurrent access to truly shared code will have performance penalties due to arbitration effects.

- Reduction of scheduling overhead — The analysis in previous sections revealed the OS overhead to be significant, see Sec. 5.1.2, 5.2.4. The handling of a timed event, for instance, required 70 cycles (cf. Tab. 5.6) to set the timer, and another 33 cycles to start the associated task chain once the timer expired. The overall processing budget of a distributed application usually is only a few hundred cycles per core. An optimization of such particular overhead, e.g., by implementation in assembly, could therefore improve the performance significantly.
- ASIP based solution — Another option would be to switch to a processing element with an application-specific instruction set, as discussed in Section 5.2.3. Due to the speedup of 2.5X, a clock frequency in the range of (only) 500 MHz would be required to support 64B PCI Express packets at line speed.[5] Using an ASIP, however, means giving up high-level programmability and a loss of platform flexibility for two reasons. First, ASIPs still lack sufficient compiler support. Manual and low-level implementation of the whole application is required to leverage the specialized instruction set, which is cumbersome and time-consuming, as our own experience shows [104]. And second, ASIP PEs increase heterogeneity, so that task-mapping and resource allocation would be restricted to the particular PEs. A (potentially SoC-wide) mapping to other PEs would require extra implementations of the system function due to the gap between the high-level model and the assembler implementation.

However, switching to an ASIP based solution is a drastic step which would specialize the whole platform towards our application. Before considering this option, other optimization potential in application and mapping implementation should be considered. In the next section, we will therefore investigate the overhead of modularity in hard- and software, and conduct an experiment which tailors the SPE logic further.

5.4 Costs of modularity and programmability

Using the NOVA prototype, we quantify the overhead of modularity in hardware and in software. To derive a lower bound on the requirements of a general-purpose core based solution, we will then look at an implementation which is embedded in a more protocol-specific hardware environment (a 'tailored' SPE) and does not use the strict modularity of the NOVA platform.

5.4.1 Hardware modularity

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

Looking at NOVA's Ethernet IO module in Figure 5.21, 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

[5]In combination with the orthogonal optimization of the scheduling overhead, which becomes more significant for the smaller instruction count on an ASIP, even smaller packets could be processed at line speed (550MHz for 32B packets).

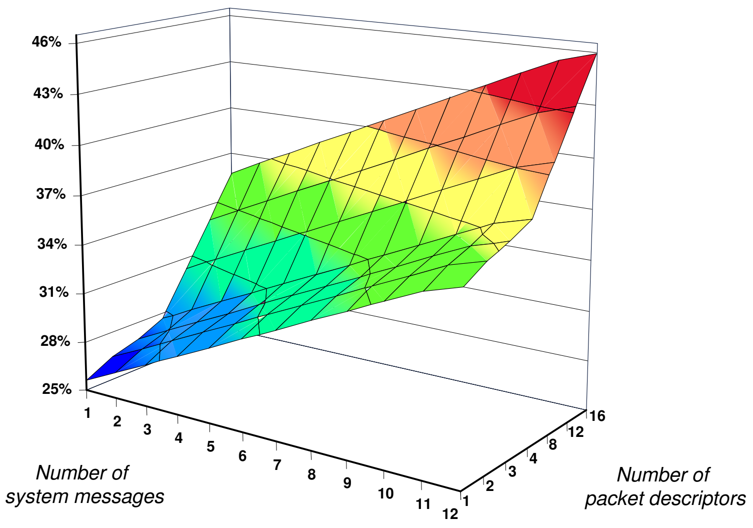


Figure 5.21: Relative area overhead of the NOVA Ethernet IO socket.

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.

The current socket implementation realizes all transfer queues using dedicated dual-port SRAM modules. This ensures maximum concurrency and implementation flexibility in terms of the different clock domains. But dual-port SRAMs are extremely area expensive, and multiple DPRAMs are more expensive than a single one of the aggregated size. Thus, if restrictions in concurrency or clock frequency ratios can be exploited, much smaller area footprints will be achievable. We applied this consideration to the PE's area estimation, see the calculation in Table 5.8.

5.4.2 Software modularity

To determine the runtime overhead of the modular programming environment, we use the CRACC code generator described in Section 4.4 and run the packet processing benchmark [174]. 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 5.22, 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).

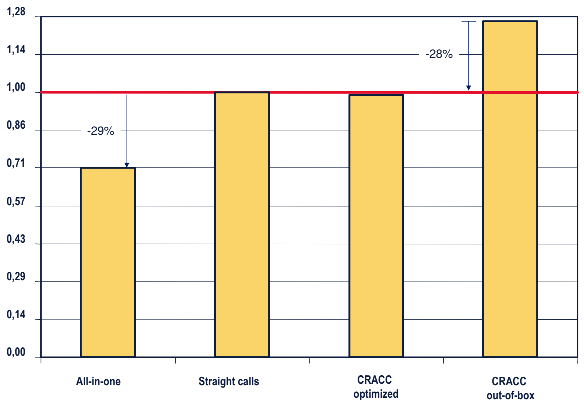


Figure 5.22: Overhead of modular software.

On the other hand, a fine-granular partitioning of the application into small elements is helpful for multi-processor mapping, as we observed in Section 5.2.4. Thus, a post-mapping optimization step, which groups task chains into 'super'-elements, would potentially be beneficial for the execution performance.

5.4.3 A less modular architecture experiment

To derive a lower bound on the requirements of an implementation approach based on general-purpose cores, we now consider an architecture which does not follow the strict modularity and generality paradigms of the NOVA platform. The system implements the GbE protocol in full duplex mode, which is the least complex protocol version in our collection.

Hardware Architecture

The architecture comprises a processor which is directly connected to the functions of the specialized PE via buffered communication links. The boundaries between GPE and SPE blur, see Figure 5.23.

The former SPE functions, sender and receiver, are here tailored to the processing specifics of the GbE protocol. Like the functions described in Sec. 5.3.2, these modules handle the CRC processing. The protocol timing function of the sender is limited to a counter for the interframe and pause intervals. Unlike the modules of the SPE, the (protocol-specific) framer and deframer functions are included here to enable a standard compliant GMII interface to the PHY. Sender, receiver, and transfer buffer interface with the interrupt controller (ITC) of the core, which is used to notify the core about state changes. For this purpose, a queue is provided that holds cause and context data of events. For the experiment, we are not interested in the SoC interface to the remaining system. This is why we keep it separated. An interface comparable to the one of the NOVA system encapsulates the packet fetch and store functions which autonomously transfer the packet data.

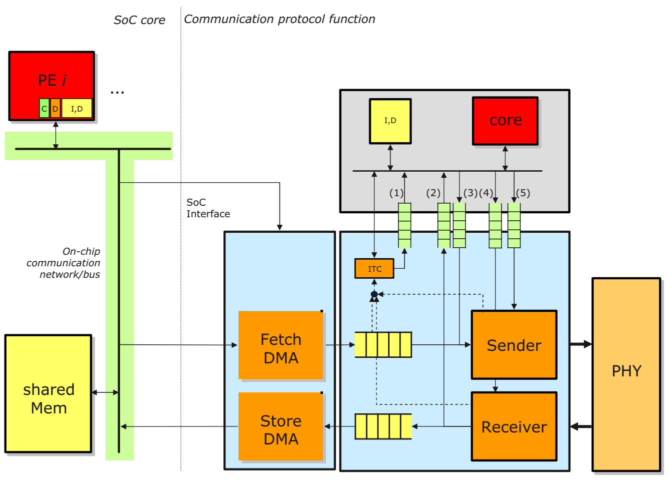


Figure 5.23: Specialized GbE SoftMac architecture.

As soon as a packet is ready for transmission, the core is notified by an interrupt via the ITC module. The software reads cause and packet length from communication link #1 and starts the transmit processing, which gathers the information required for the operation of the sender module. As soon as its processing is complete, this data is written into communication link #5. The sender reads the information into its register context and starts the transmission of the frame as configured.

Incoming packets are handled by the receiver, which stores them in words into the receive buffer. In the process, the packet header together with the processing status is copied into communication link #2. As soon as the packet reception is completed, an interrupt is generated, and the processor core is notified. The software uses the provided information to process the packet. It also calculates the true packet len, which is used by the sender to update the (alternating) write pointers of the receive buffer (link #4).

If the buffer runs full, pause frames are generated by the software and written into link #3, from where they are transmitted as other tx frames. Received pause frames are forwarded to the core (#2), which deletes them from the receive buffer and configures the sender to pause for the specified interval.

Protocol Software

The software for the GbE protocol is implemented as an interrupt routine of the processor core. Following the insights of the previous section, the program implements the full protocol function as single loop body without function calls and sub programs. Communication accesses are included directly as inline assembler instructions. The rest of the program is C code.

The performance critical execution path of the software is the reception of a data frame, its validation, and receive buffer update, which is followed by the generation of a pause frame.

Implementation results

As before, the architecture is synthesized for an Xilinx Virtex-II FPGA and a 90nm Infineon semi-custom technology. For efficiency reasons, the FPGA implementation deploys the Xilinx Microblaze CPU as processing core and uses the fast-simplex links for the communication. The Microblaze is optimized for the FPGA implementation which results, compared to the MIPS core, in much higher operating frequencies (100 vs. 25MHz) and fewer slices.

Without consideration of the required context switch, the execution of the critical path requires 97 instructions which translate into 129 cycles on the Microblaze. Taking into account the overhead for routine call (6+6 cycles) and context saves/restores (14x4 cycles) a total of 197 cycles is required. This means that full line speed for minimum size packets could be achieved with clock frequencies between 210 (no context save required) and 294 MHz (full context save required). The full program requires less than 2kB code and data memory (1132 Byte code memory and about 200B data memory).

Table 5.12 reports the ASIC area requirements for the GbE SoftMac. The area depends on the memory configuration. The smallest area has the memory configuration with a single 2kB memory for both, code and data. Compared to a NOVA implementation with a single general-purpose PE and similar memory configuration, shown in the right column, the GbE SoftMac has an area advantage of 16%.

Table 5.12: Area requirements in mm^2 for the processor-based GbE variant with different memory configurations in comparison to the NOVA implementation.

sub-module	GbE SoftMac			NOVA SoftMac	
	Neumann	Havard	Havard		Havard
MIPS core	0.2079	0.2079	0.2079	GPPE	0.7161
data mem	0.0982	0.0630	0.1710		-.1190
inst mem/boot rom		0.0487	0.1677		-.1080
Rx buffer fifo	0.0900	0.0900	0.0900	sPE	0.4418
Tx buffer fifo	0.0900	0.0900	0.0900		
Rx len fifo	0.0193	0.0193	0.0193		
Tx len fifo	0.0237	0.0237	0.0237		
Communication links	0.0215	0.0215	0.0215		
Hardware logic*	0.0595	0.0595	0.0595		
SoC interface**	0.1561	0.1561	0.1561		
total (minimal mem)	0.7626	0.7762			0.9309
total (regular mem)			1.0032		1.1579

*) includes FIFO control logic. **) 3xMemIf, 2xDMA, SysMsgIf (Tab. 5.10).

The software implementation of the tailored GbE MAC is substantially better in both, code size and runtime performance, than the CRACC-generated version. Including operating system functions, the model generated by CRACC has a runtime code size of approx. 7 kB and would require 572 cycles for the execution of the critical path (on the MIPS PE). These differences have several causes:

- Overhead of modularity — The GbE implementation essentially is a single function which comprises all necessary operations. No subroutines whatsoever are required. The Click model on the other hand has at least ten elements in the critical path. Considering the insights on the overhead of modularity from Section 5.4.2, this alone would account for approx. 152 - 170 cycles.

- Comprehensive Click elements vs. essential functions — Most Click/Cracc elements comprise code that makes them more robust, better interchangeable, or independent. Elements usually can be configured for several use cases or are implemented for the general case. The Classify element, for instance, is only used to filter pause frames (particular ethertype value, i.e., a comparison of two bytes) but can easily cope with large sets of classification rules. The GbE SoftMac implements the essential function in a few lines.

- Shared state and packet descriptor handling — The Click/Cracc implementation holds processing state in structures (either in the element or in the packet descriptor) that are stored in main memory. To share state, elements have to exchange it explicitly, i.e., it must be copied in memory. This requires extra code and processing cycles compared to the GbE implementation, which holds all processing state in local variables that are mapped to registers.

- Dynamic scheduling, different IO functions, other platform overhead — The NOVA scheduler enables dynamic task scheduling and handling while the GbE Mac relies on a statically scheduled implementation. NOVA packet descriptors furthermore are 64 Bytes and must be moved between memory and transfer queues (10+10 cycles). The GbE Mac reads only those 20 Bytes that are actually required by the protocol (5 cycles). The transfer of the pause frame costs only seven cycles. Furthermore, the NOVA platform provides APIs for timers, memory access and management, message passing, and packet handling, which are all avoided by the tuned GbE implementation.

Overall, the optimization potential of a software protocol implementation that is generated by CRACC is large. For the Gigabit Ethernet protocol, we observe a potential runtime improvement by a factor of 2.9X for the manually tuned (but still in C implemented) version of the protocol, which avoids the Click/CRACC programming model and platform abstractions. However, since the GbE protocol is the least complex function of our protocols and, therefore, exposes the platform overhead the most, we expect the potential for optimizations to be smaller for the other protocols.

The tailored hardware implementation of the protocol can not improve the required area much. For the GbE implementation we expect an area advantage of only 16% compared to a single-core NOVA platform. This is not surprising since, in any case, the implementation is dominated by memories, communication interfaces, and the processor. The actual protocol-specific logic is only a fraction of the area. The NOVA general-purpose implementation, however, scales better with the number and the type of the supported protocol functions.

5.5 Related work

Work related to our implementation of packet-oriented communication interfaces based on a programmable platform can be found in in three domains: I) modular and programmable platform architectures, II) existing Flexibility in communication interfaces, III) other flexible IO interface implementations.

5.5.1 Modular platform architectures

Programmable platforms, according to [56] p.15, are the next design discontinuity. They promise higher design productivity, easier verification, and predictable implementation results. Rather than the design through assembly of individual IP blocks, programmable platforms provide hard- and software templates based on a family of processors that can be tailored quickly to optimize costs, efficiency, energy consumption, and flexibility. Key features of a programmable platform thusly are modularity, simplicity, and scalability.

- Modularity in hard- and software aspects enables the system-level designer to quickly customize the platform to the specifics of an application without breaking the programming model. It also leads to clearer design choices when building complex systems [92]. In addition, it eases the extension of the platform's modules by providing a set of well defined interfaces.
- Simplicity improves usability and eases the use of a platform. Instead of, e.g., centering a platform around a particular (feature-rich) processor architecture and bus protocol, a platform should keep concerns separated. The platform should be core- and communication-protocol agnostic. Consequently, an existing bus interface should be replaced with another one rather than with a wrapper for the existing interface. In addition, this means that the complexity of modules such as a standard processor core should be carefully reduced as needed, e.g., by removing MMU, caches, and certain functional units.
- A scalable platform is able to accommodate a wide range of I/O interfaces and processing elements for handling different processing and communication requirements without breaking the programming model or forcing the applications to be re-written for each platform instance.

A survey of the broad variety of packet processor architectures and platforms can, e.g., be found in [177, 205]. Most of these architectures have been optimized for high-performance. But other platform aspects such as the programmability and modularity have often been neglected. Further examples of commercial platform solutions for wireless and multimedia domains are Nomadik from ST Microelectronics [12], and Philips Nexperia [44]. As mentioned before, these architectures are centered around particular cores and bus protocols.

There are a few platforms [145, 116] which follow similar approaches as the NOVA platform in terms of programmability and message-passing communication:

- StepNP [145] is probably the processor platform which is most closely related to NOVA and CRACC (see the discussion in Section 4.2.2 on its programming model and exploration method). Both platforms support the range from general-purpose over application-specific to hardwired (co-) processing nodes which can be connected following the network-on-chip paradigm.

Similar to NOVA, the StepNP platform provides support for message-passing in hardware. In addition (and unlike NOVA), it implements centralized hardware accelerators for dynamic task allocation, scheduling, and synchronization, as it is beneficial for SMP and SMT programming models [146], which are not in the focus of the NOVA hardware implementation. However, StepNP seems to always map processor-memory communication onto the same latency-intense on-chip communication network and therefore requires hardware multi-threading to hide memory access latencies in any case [146].

NOVA supports both, simple pipelined memory buses (split-transaction, ld/st access) as well as memories which can be accessed via message-passing primitives (rd/wr, rd response). Contrary to StepNP, NOVA does not depend on hardware multi-threading and therefore does not limit the application concurrency. The run-to-completion and encapsulated-state semantics of the Click model-of-computation naturally support software tasks-switches with low overhead.

Judging from the case studies in [146], StepNP's platform overhead is somewhat comparable to NOVA. StepNP implements only small MPI transfer buffers of 512b (16 words) while NOVA implements at least 512B (128 words). Yet, for a fair comparison, StepNP's extra hardware contexts must be taken into account as well (between 112 and 224 words), since they are used to store the processing context for the individual transaction. The reported software overhead for the handling of messages as well as the round trip latency is similar to NOVA's system messages. The extra overhead of 10-12 instructions on the receiver ('server-side') to invoke the requested method seems quite efficient but may be too optimistic in peer-to-peer communication settings. NOVA's current implementation depends on the number of (dynamically registered) 'listeners' and requires between 19 and 47 instructions for our application mappings. Code and data memory footprints for OS-like functions are not reported.

- Tensilica's Xtensa [116, 121] is a configurable processor which (apart from the usual bus interface) provides direct support for communication with data-flow semantics. The processor template [64] allows the addition of arbitrary-width IO ports, which can be used, e.g., for queued and direct register-to-register communication between two processors. This feature allows embedding the processor directly in the data flow as data can be consumed and generated using in- and outports.

  However, use cases such as the packet descriptor passing of Click can not leverage this feature well. Packet descriptors must be stored in local memory but not necessarily require processing of all data words. A flow-through approach would therefore impose unnecessary load on the core, see Section 5.2.2. Extra hardware, similar to NOVA's tiny-DMA function would be the consequence.

  We kept the NOVA platform core-agnostic by using a memory-mapped subsystem which provides access to local peripherals via load/store instructions. This way, arbitrary cores (including the Xtensa) can be supported. The exploitation of core specific-features such as specialized low-latency IO ports would be an interesting optimization of a particular NOVA-PE.

We explored the benefits of a flow-through setting using the Xilinx Microblaze with its Fast-Simplex-Links in Section 5.4.3. The Click/Cracc programming model would be portable to a Xtensa PE. Tensilica itself does not provide a multiprocessor programming model.

NOVA postulates strict modularity in hard- and software as a key concept. This is in tune with the goals and efforts of initiatives such as the VSI alliance which has a broad scope and aims at virtual IP sockets to enable reuse and integration [117]. The NOVA socket follows the widely used OCP specification[6], which is more focused and better suited to our needs. The definition of meaningful software interfaces and programming abstractions is still an area of lively research, see, e.g., [115, 208, 179] and the references therein. The Multicore Association is aimed at the definition of a MP communication API [118]. Our implementation of the NOVA OS functions was driven by the requirements of the Click model-of-computation. Wherever possible, well known and effective techniques were used, e.g., for memory management or message-passing communication.

5.5.2 Flexible interface implementations

Already the analysis of contemporary SoCs in Chapter 2 revealed different implementation techniques to provide flexibility in communication interfaces. Apart from the physical multiplexing of individual IP modules [3] and the embedding of reconfigurable logic into the SoC [61, 120], we found few (rare) programmable approaches for IO interfaces:

- Use of specialized co-processors — At the high-end, some systems deploy specialized co-processing elements that are micro-programmable for a well defined set of network interfaces (ATM, Ethernet). The Motorola C-5 [51], for instance, uses two Serial Data Processors (one for send, one for receive) in each of its 16 channel processors. They provide a number of individual low level tasks for the network interface such as bit field extraction and CRC calculation.

- Fully in software — For low performance interfaces, we find implementations solely provided in software running on processors. Such a solution provides full flexibility with only minimal area consumption for the interface. In case of Ubicom [47], the multi-threaded processor with specific instruction set requires a specialized operating system to fulfill the real-time requirements of the interfaces. Per software IO interface, a dedicated hardware real-time thread is required (up to six are concurrently possible), which is scheduled deterministically (TDMA). The remaining threads are used by the application software and are scheduled round-robin. Although a promising approach in terms of full programmability, the solution can

[6]www.ocpip.org

only support limited bandwidth [47, 62]. The implementation of a standard PCI interface, which is supposed to run at 33 or 66 MHz, achieves a bus frequency of 10 MHz.

Certainly, one reason for this serious limitation is the flow-through approach of the architecture. The processor must move every data word from and to the interfaces, which leads to extremely tight budgets of a few cycles per word. The Ubicom approach is therefore not feasible for our high-throughput communication interfaces.

To the best of our knowledge, there have been no successful attempts to implement PCI Express, Hypertransport, RapidIO, or Gigabit Ethernet similar to us on a programmable architecture. Only the work in [80] describes the idea of a PEX device in which the hardware is assisted by a microcontroller. The controller runs a realtime OS and apparently handles link layer packets including the replay buffer. However, neither protocol performance nor implementation results are reported.

Mature work related to software implementations of our protocols can be found only for wireless Lan, probably fostered by the relatively low bandwidth requirements, the proximity to software-defined-radio platforms (which require MAC functionality for their already flexible PHY radios), and the availability of a 802.11a Mac reference implemented in SDL. In the following, we will therefore examine a selection of these wLAN Mac implementations and discuss them with respect to their flexibility in three broad categories in comparison with our approach:

- The usual hardware/software function split — Most of the published protocol implementations follow a function split, which implements time critical and compute intense aspects such as channel access timing (DCF), control frame processing, CRC calculation, and address validation in hardware close to the PHY interface - the 'low Mac'. Other protocol functions such as management and queueing, which are neither compute-intense nor performance critical, are kept in software – the 'high Mac' [65, 165, 216, 140]. Across the implementations, this function split is relatively fixed.
  - In [65], Hannikainen et al., describe the implementation of a wLAN MAC protocol starting from SDL. The resulting system maps the most critical functions into hardware (channel timing, encryption, CRC), the rest is software on a DSP processor. The reported memory sizes of 490kB code, 61kB data, and 100kB dynamic memory include functions that are usually considered for running on a host processor as part of the OS' device drivers such as user interfaces, network management, rate selection, and statistics.
  - In [165], some data is provided on the performance of different operating systems in terms of context switching time and code and data memory footprints. Due to the considerable overhead, the authors avoid the use of any OS. Instead, they implement the high Mac function in two routines (rx, tx) as part of the interrupt handler on the DSP processor of their prototyping system. After some code optimizations, a throughput rate of 54Mb/s is reported for larger frames that are fragmented into 256Byte segments.

  The best OS in the comparison is ThreadX, which is reported to require 15kB code and 2kB RAM. The NOVA runtime configuration for wLAN requires approx. 6kB code and a few hundred bytes RAM for the OS functions. Context switching, i.e., swapping the register context is reported to require 100 cycles for ThreadX. The NOVA system does not require this overhead due to its run-to-completion model-of-computation.

  – Panic et al. [140] present synthesis results for a MIPS4k based system which follows the usual function split. The implementation suffers from deficiencies such as the lack of memories for the target standard-cell technology. Without the memories[7], the hardware accelerator is 5/6th of the area of the MIPS processor. This seems quite large. Without its memories, the NOVA SPE would be less than 50% of a MIPS core.

• Modifications of the MAC function — Although for diverse reasons, a considerable fraction of the related work tries to change the MAC function, see [43, 75, 209, 210] and the references therein. All this work would benefit from a platform which enables easy changes on the MAC function.

  – Doerr et al. [43] describe the SoftMac approach, which tries to disable most of the wireless MAC function found in commodity hardware. This enables a framework (MultiMac) that supports coexisting software Macs with different functions. Interestingly, an interface to Click is provided to ease the protocol development. So far, the solution would not achieve wLAN timing in software.

  – The work by Wu et al. [209, 210] circumvents changes of the MAC itself by describing a layer on top of regular 802.11 a/b/g MACs. The solution provides quality-of-service features to support Voice-over-IP in wireless multi-hop networks. For this purpose, the low-level device driver on the host processor is changed and extended. According to the authors and their discussion of related work do others depend on changes of the DCF function or require 11e capable MAC functions for the same purpose.

  – Addressing the need for changeable MAC functions from a different angle, Hunter et al. [75] present a FPGA based prototyping platform, which (to some extend) allows modifications of the low Mac function.

• Fully flexible implementations — Few papers actually map the full MAC function onto a processor core and analyze the achievable performance. Similar to our fully flexible approach, the authors use their findings as starting points for further optimizations either of the processor or its hardware environment.

  – To motivate their later implementation of the usual much less flexible function split, Panic et al. ([140], see above), start with a SDL simulation model of the MAC. They use the model to generate C-code which is profiled on a MIPS M4k instruction set simulator. Since

[7]The memory for the low Mac is synthesized from standard cells and reported separately.

the execution requires some sort of SDL runtime environment causing heavy overhead, the generated code is manually optimized. After optimizations, the authors report a clock frequency requirement of 1000 MHz for a single-core 11a MAC (12Mb/s) fully implemented in software. This seems somewhat in the range of our own initial profiling results (4 GHz for 54Mb/s, cf. Sec. 5.1.3). However, a detailed analysis (which may have revealed the critical and only dependency on the monolithic encryption function) is not performed. Further quantitative insights that would support the usual function split of their resulting system are not provided.

– Shone et al. [184] implement a wLAN MAC fully in software on a PowerPC running at 400MHz. The processor runs VxWorks as operating system and is part of a software-defined-radio prototype, which comprises several physical modules that are connected by a VME bus. The implemented Mac function is 802.11a. Features such as security (WEP) and RTS/CTS control frame handling are not implemented.

  For this system setup, three severe difficulties are reported in meeting the realtime requirements of the wLAN protocol (i.e., the SIFS deadline of 16μs). First, the inter-module bus connection is too slow. It requires already 40μs to transmit a maximum Ethernet frame between PHY and MAC module. Second, interrupt response (3μs) and context switching (200μs) times of the OS are too large. And third, the clock of the CPU board had a jitter of 399 μs. As a consequence, the protocol deadline had to be extended by at least a factor of ten to make the system operational.

  Due to a single-chip implementation, our NOVA solution avoids the issues of clock jitter and extra transfer times. In addition, NOVA executes the Mac protocol processing concurrently to the packet transfer which shortens the critical path. NOVA's event handling and task activation latencies are in the order of less than .5μs.

– Other work [74, 84] maps a Mac onto an ARM processor to study the impact of micro-architectural changes. In [74] the benefits of multithreading, extra functional units, and a reduced instruction set, are reported in comparison to the original configuration. Overall a performance improvement by a factor of two is reported using a 3-thread two-issue configuration. Indications of the absolute performance are not provided. The work in [84], which provides a profile of the executed instructions, argues in favor of a dynamically configurable instruction set to increase the utilization of the micro-architecture.

  We explored the performance potential of an application-specific processor with reduced instruction set in Section 5.2.3 and observed relative improvements in the same order of magnitude (factor of 2.5) for the given implementation.

5.5.3 Complexity of individual protocol implementations

This section seeks to gather *publically* available data on the implementation costs of individual communication interfaces *in hardware*, and (if possible) compares their function to our implementations.

In general, it is difficult to find published data on the costs of particular implementations and its breakdown to individual functions. The most comprehensive set of data could be collected for Xilinx IP cores, which we synthesized on a Virtex-II FPGA. Each of the Xlinix cores basically implements the least functionality to comply with the standard, i.e. optional features were disabled. SoC interface functions such as DMA engines and bus interfaces are not included. Table 5.13 summarizes the results. Unfortunately, a detailed functional breakdown of the required resources was not possible for PEX, HT, and RapidIO due to the lack of analyzable source code. A wireless Lan Mac hardware module is not available. The related work on wLAN MACs implementations discussed in the previous section implementations did not provide conclusive data on hardware resource requirements either.

Table 5.13: Minimal resource requirements of individual communication interfaces based on Xilinx IP cores on a Virtex-II FPGA.

Module	Included Function	Slices	BlockRAM
Gigabit Ethernet (GbE)	PHY, DLL, TA	1400	4x2= 8kB
Hypertransport (HT)	PHY, DLL, TA	3990	20x2=40kB
RapidIO (RIO)	PHY, DLL, TA	5277	9x2=18kB
PCI Express (PEX)	PHY, DLL, TA	6719	16x2=32kB
Microblaze (Mbz)	CPU, OPB-If, 2xLMB	1020	>2x2= 4kB

References: HT [211], RIO [215, 212], PEX [214], GbE [213]

As a point of reference, the resource requirements of an Microblaze processor core with a minimal subsystem comprising an OPB bus interface, and code and data memory interfaces (local memory bus - LMB), are provided.

- PCI Express and RapidIO — In addition to the functionality modeled in Click, the Xilinx cores [214, 215, 212] include initialization and power management (PEX) functions as well as maintenance and status tracking. Also, configuration and management modules are included, which handle communication with the other modules and provide an interface to the user-level.

  Other implementations of PCI Express endpoints:

  - Based on designs by Infineon Technologies, an area of 1.48mm^2 for a rudimentary device without PHY functions can be estimated. This includes an AHB slave as SoC bus interface and support for one virtual channel. The transaction size is limited to the burst size of the bus (max. 64B) to avoid segmentation and reassembly. DMA and reordering features are not included.

    If three channels are included (as in our model), the area increases to 2.09mm^2 due to larger memory. Without the SoC interface and using the ratio published in [30] (see below) to estimate the required slices, this would roughly translate into 9000 slices.

- Hypertransport — Similar to our HT Click model, the Xlinix IP core [211] used in Table 5.13 implements an revision of the standard which does not include the packet-based mode with retransmissions (see the discussion of [30] below). For the same revision, another commercial HT endpoint is available, which requires 2.7x as many slices (10750) [50] but has a wider data path (128b) and allows higher throughput. In both cases, a SoC socket interface is not included.

Other work on HT implementations includes [68, 30]:

- Hasan et al. [68] implement a network-on-chip based on the HT protocol which may homogenize the overall communication structure of systems. Their 'HT light' tunnel implementation is discussed in reference to HT. However, in terms of implementation complexity for an HT node, no comprehensive picture is provided. Only two aspects are mentioned: 1) a buffer memory requirement of approx. 9.4kB for such a node, and 2) a complexity estimate of 9.25k gates only for the CRC calculation logic.
- Castonguay and Savaria [30] implement a protocol compliant HT tunnel (device with forwarding function). The system requires approx. 132k gates or 18.1k LUTs. In addition, a total of 5.4kB memories for data (3kB), command (400B), retry buffer (1kB), and user data buffer (1kB) is used. According to earlier work by the authors [29], this memory size is sufficient to store one packet per flow. The per-module breakdown identifies flow control and retry functions to contribute most to the system complexity (52% of the area). Since a tunnel requires two interfaces, and all modules except user interface and configuration and status are instantiated twice, a ballpark figure for the complexity of a single interface can be estimated as 73k gates without memories. Given the reported size on a Virtex-II Pro FPGA, this would translate into approximately 73/132 x 18.1 x 1/2 = 5000 slices and is in the range of the other implementations.

• Gigabit Ethernet — The area breakdown for a complete Gigabit Ethernet Mac controller including an exemplarily interface to a 64bit PLB is specified in Table 5.14. In addition to the data buffers (4kB each), extra queues are implemented, which store the length of each packet. The data is based on our own implementation which was verified against the Xilinx IP core.

Table 5.14: Resource breakdown of a GbE hardware controller.

Module/Function	Slices	BlockRAM
PHY layer	~480	
MAC (DLL/TA)	575	
Tx Buffer (data+len)	201	3x2=6kB
Rx Buffer (data+len)	207	3x2=6kB
PLB bus interface (64b)	~500	
	1960	6x2=12kB

5.6 Chapter summary and discussion

This chapter investigated the feasibility of a programmable solution for packet-oriented communication interfaces, which is:

- composable from a modular platform that can be scaled to the interface specifics of a System-on-chip without breaking the overall construction paradigms and is
- easily programmable, following a high-level programming model that is natural and productive for the application domain.

NOVA platform

For this purpose, we have developed NOVA, the Network-Optimized-Versatile Architecture platform. NOVA supports the exploration of a programmable and modular solution by applying four design principals:

- Modularity — Generalized hardware sockets are used 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.
- Simplicity — To tackle the complexity of today's SoC's, NOVA is build from basic hard- and software modules, which: keep concerns separated, do not make assumptions on each other's function, and are designed for a particular purpose. Instead of extending a module to serve multiple purposes, we prefer to provide another module wherever possible. In combination with the modularity principal this helps to avoid the usual performance-killing fallacy of complex hard- and software IPs wrapped in multiple layers of abstraction circuitry for configuration and interfacing.
- Scalability — NOVA supports a wide range of I/O interfaces and processing elements to handle different application requirements without breaking the programming model. The NOVA socket and message format allow the integration of traditional shared buses [164], as well as better scalable Network-on-Chip (NoC) techniques [18] and enable arbitrary numbers of processing nodes.
- Customizability — NOVA provides design points for customization if the most flexible solution does not meet design criteria. Coprocessors, for instance, either tightly or loosely coupled, can be used without breaking the representation of the application.

Guided by these design principles, the NOVA platform implements the state-of-art in concurrent SoC platforms while avoiding some of the fallacies of other approaches as discussed in Section 5.5. Our analysis of the overhead introduced by modularity in hard- and software in Section 5.4 further revealed costs in terms of on-chip area and execution performance that are also comparable to existing techniques. We have published our findings in [166].

Design space exploration

Using NOVA and our code generators, we explored the performance / area tradeoffs involved into the implementation of a programmable interface solution. Following our application-driven methodology, we started with the profiling of the communication interfaces mapped on a single-core architecture and observed:

- Per-packet protocol processing — Most protocol processing happens per-packet and is only based on the packet's context/header.

- Cycle-consuming payload processing — Those rare functions that actually touch the whole packet, such as CRC calculation and encryption, are the most cycle consuming functions.

- Non-deterministic processing times — Large monolithic elements in combination with Click's run-to-completion model-of-computation make it impossible to meet hard real-time deadlines as required,e.g., for wireless Lan.

The performance of the single-core architecture is not sufficient as the analysis revealed. Especially for small packets, multi-gigahertz clock frequencies would be required to meet either the throughput requirements (PCI Express) or the tight deadlines for packet reception and response generation (wireless Lan).
Next, we explored several axes of the platform design space and studied their impact on the system's performance using our SystemClick framework and the collected profiling data:

- Core type — To compare different embedded RSIC cores and their compilers, we profiled a packet processing benchmark implemented in CRACC on a representative set of synthesizable embedded 32bit processors. Although the general purpose cores achieved a somewhat comparable performance in terms of packet throughput for our benchmark, we found the MIPS M4k to have the best performance-area trade-off. The application-specific core included in the comparison were not able to perform well, due to insufficiencies in its high-level programmability.

- Hardware accelerators — By deploying hardware accelerators for the payload processing functions CRC and data transfer (DMA), the processing performance can be improved by factors between 2.8 (min) and 114 (max) for PCI Express, the most complex high-throughput protocol. In case of wLAN, the slowest of our protocols, up to a factor of 1.2 was observed.

  Another candidate for hardware acceleration would be the WEP function which is used by wireless Lan and consumes the vast majority of its cycles. However, the overall performance requirement of wLAN is moderate so that WEP could remain in software, saving protocol-specific area.

- Application-specific instruction set processor — By deploying an ASIP instead of a general purpose processing core, performance improvements by a factor of 2.53 can be expected looking at the critical path of PCI Express.

  Using an ASIP, however, means giving up high-level programmability and a loss of platform flexibility. Our experience from network processing [104] and our ASIP design experiments with the Tipi framework published in [128, 168] show that ASIPs still lack mature compiler technology. See also the performance of the ASIP and its compiler in our core type exploration. Programming them in assembler is the consequence as we did for the static analysis in Section 5.2.3.

ASIP PEs increase the platform's heterogeneity and limit mapping and resource allocation since the whole application must be re-implemented in assembly to leverage the specialized instruction set. Although supported by the library based approach of Click/Cracc this would lead to multiple sources for an element due to the gap between high-level and assembly implementation.

- Multiple processing elements — Exploiting concurrency in the application, we investigated mappings of up to seven cores.

  The analysis revealed a four-core mapping to be a reasonable choice between speedup and number of cores for the high-throughput interfaces. A four-core mapping of a duplex PCI Express link achieves a speedup of 1.85. The considered higher core mappings would only marginally improve the speedup (5:2.19, 6:2.34, 7:2.73) due to the given application granularity and implementation.

  The realtime requirements of the wireless Lan protocols can be met using a dual-core mapping, which separates real-time critical and compute-intense long running functions from each other.

- Communication topology — To study the communication requirements, we assumed all processing nodes to be connected by a single write-only bus which is used to exchange packet descriptor messages. Based on SystemClick several hundred design points were evaluated that vary in bus speed and size of the in- and outboxes.

  We found a single bus to be sufficient in any case if it runs at 30% or more of the PE's clock frequency. Only those cases in which in- and outboxes have only a single entry per link show slight performance degradation (-1%).

  With increasing bus utilization, i.e. for lower bus speeds, the size of the transfer queues becomes more important for compensating the non-availability of the bus. In the case of a 20% speed, sizes of 3 (out) to 4 (in) entries per link are necessary to achieve the full performance. If the bus speed becomes too low (10%), the available communication bandwidth is no longer sufficient. In this case, queues do not matter. They are always full and stall the processing pipeline.

4PE+SPE instance of the NOVA platform

Based on these insights, we propose a NOVA platform instance with four general-purpose PEs for the protocol processing software and a specialized hardware processing element which comprises flow-through accelerators for the payload processing and handles the PHY interface. The architecture is protocol agnostic. Only the CRC function, which is part of the SPE, must implement the superset of all used polynomials. Leveraging the implementation of the NOVA prototype, we conclude on the platform instance:

- For the chosen 90nm ASIC technology, the architecture runs at a system clock frequency of 360 MHz and requires an area of 3.3 mm^2. 58% of the area are memories. The modules are decoupled and run asynchronously.

- The system can handle PCI Express at full duplex line speeds (4 Gb/s goodput) for packets of 206 Bytes or larger. Smaller packets can be handled at lower speeds, e.g., 64 Byte packets at 1134 Mb/s. A similar performance can be expected for RapidIO and Hypertransport.
- Wireless Lan can be handled at line speed using only two of the four processing elements. Due to the complex protocol processing, the wLAN has higher code memory requirements so that the code memory of the two other PEs is used as well (2x 2x 8kB).
- The Gigabit Ethernet protocol can be handled at line speed with a single processing element after software optimizations.

The detailed analysis of the performance revealed potential for improvements:

- Application implementation — First, the concurrency of the application model can be improved to ease partitioning and pipeline balancing. See the discussion on the PCI Express model in Section 5.2.4. Second, elements and their parameters should be tuned to the platform's capabilities based on profiling feedback.
- Mapping and platform overhead — Elements belonging to the same task chain should be concatenated to avoid function call overheads. Also, the OS function for event handling and task scheduling (currently implemented for flexibility) can be optimized to reflect the more confined setting of the protocols.
- Architecture modifications — For the high-throughput interfaces more processing elements can be deployed if the granularity of the application permits.

Main results

Looking at the *performance* we found an implementation based on a modular and protocol-agnostic platform to be feasible:

- without performance restrictions for Gigabit Ethernet and wireless Lan 802.11 a/b/g. These protocols can be fully handled in software.
- with performance restrictions for PCI Express, Hypertransport, and RapidIO. Full line speed cannot be achieved for small packets.

Even with the performance restriction PEX, HT, and RIO are fully functional since the protocols' flow control schemes will effectively throttle the throughput in compliance with the standard. This is different to a known software-based implementation of PCI, which violates the physical wire speed ([62], cf. p. 131).

The *area footprint* of such a solution depends on the use case. If we compare our flexible solution with individual IP modules, we find:

- Compared to a single GbE interface (cf. Tab. 5.9), the programmable solution would be larger by factors between 1.3 (ASIC, dual port memories) and 1.64 (ASIC, single port memories).

- The flexible implementation of a PCI Express endpoint (4PEs) is larger than a dedicated ASIC implementation by a factor of 1.59 (cf. p. 135).

Integrating a flexible implementation as a replacement for a mature and single protocol interface is not beneficial in terms of the required area. A flexible solution is an alternative to physical multiplexing in cases where mutually exclusive protocols must be supported:

- The combination of PCI Express with either HT or RIO would already save some area (7% for HT+PEX, 12% for RIO+PEX)[8].

- Support of all three interfaces has an area advantage of 37%.

This indicates an advantage of a flexible solution in terms of performance and area. In settings with moderate throughput requirements already the current implementation is feasible as we will conclude in the next chapter. Assuming the discussed software optimizations are applied, further significant improvements of the performance/area ratio can be expected by different means: First) by deployment of faster cores (either by semiconductor technology, full custom design, or micro architecture). The 2nd generation of the Intel Microengine, for instance, run at clock speeds of up to 1.4 GHz [53]. And, second) by deviation from the general-purpose instruction set.

[8]Calculated on the basis of the PCI Express IP module size reported on page 135. The size of HT, RIO is estimated based on the ratio of logic slices in table 5.13 assuming similar memory requirements.

Chapter 6

Conclusion

Today's heterogeneous systems-on-chip combine customized blocks for communication interfaces with processing cores (see Chapters 1 and 2). Such *programmable platforms* promise higher design productivity since they are built more abstractly from coarse-grain building blocks rather than being designed on the RT level. By their software programmability, they also make late or in-field adaptations of a system's function possible, thus reducing the first-time-right risk and increasing the longevity of systems in evolving markets. For application domains such as network and protocol processing, dozens of different *multiprocessor* platforms were created which deploy concurrent processing elements in increasing numbers. Their IO interfaces remain dedicated non-programmable hardware modules integrated separately and relatively late during the design process.

But communication interfaces must become a first class citizen for the design of embedded systems-on-chip. They contribute significantly to the complexity and the heterogeneity of an embedded system as contemporary evidence shows. This is caused firstly by their own growing complexity, which is driven by increasing bandwidth requirements and more sophisticated protocol processing (cf. Sec. 1.1), and, secondly, by the diversity of the protocol landscape requiring the integration of many different IO functions on a single die (cf. Sec. 2.6).

This work tackles the diversity of communication interfaces and their contribution to a system's heterogeneity by exploring the feasibility of programmable IO architectures following two key ideas. A flexible module, which can be targeted at multiple IO protocols (1st key idea), reduces the number of heterogeneous building blocks on the die. It also enables reusing the system in changing and evolving communication environments since IO functions can be updated or replaced. And, it eases a system's integration by supporting variable function splits between the core and the IO module. A programmable solution amplifies these advantages. By leveraging a platform's processing cores for the IO function (2nd key idea), a platform becomes even more homogeneous. System integration and deployment are eased further since both, IO and core functions, are specified using the same integral programming model and can be distributed more flexibly among the processing cores. This means that the interface function becomes a part of a now fully programmable platform.

The central hypothesis of our work is that there is a set of communication interfaces that can be implemented by a flexible solution so that a) it is programmable using a platform's programming model, and b) is applicable at reasonable costs in terms of performance and area. Focusing on the fast-growing class of packet-oriented IO protocols (as motivated in Sec. 2.6) this hypothesis was proven in two steps:

1. We developed a domain-specific methodology for the application-driven development and deployment of programmable communication interfaces and provide a set of code generation tools for its application.

2. We applied our methodology to a set of communication protocols and quantitatively explored their performance on a modular architecture platform.

The results indicate the feasibility of a programmable interface architecture for embedded settings with moderate performance requirements and demonstrate the absorption of interface functions into a fully programmable platform as will be discussed in the next sections.

6.1 Application-driven methodology

Due to the lack of established methods for the systematic exploration of a platform's design space, we had to develop a comprehensive methodology for the application-driven development and deployment of programmable platforms to be used for the exploration of flexible interfaces. The Y-chart-based approach focuses on the characterization of the application domain as early as possible. With this input, the design space can be narrowed to one major design trajectory starting with the most flexible (i.e., fully programmable) solution and iteratively refining the platform to meet performance/cost constraints. The methodology specifically targets the whole system to ensure the integral consideration of the communication interfaces. Further contributions to the disciplined development and deployment of programmable SoC communication interfaces are:

- Modeling technique for packet-oriented interfaces — By modifying and extending the well-established Click framework, we applied best-practise in implementing network applications to the domain of communication interfaces. Contrary to known approaches our framework focuses on exploration and implementation efficiency.

- CRACC implementation framework — Based on architecture-independent protocol specifications in Click, we derived implementations for concurrent platforms using our CRACC code generator. CRACC closes the gap between Click and an efficient software running on a wide range of embedded processor cores while related work focuses on hardware generation or a specific ASIP target. It enables Click as integral programming model.

- SystemClick exploration framework — For a fast exploration of the design space, SystemClick is provided. From Click specifications of application-architecture mappings, SystemClick generates functional SystemC models which can be executed natively using a performance database. To enable quick alterations and, in contrast to other work, SystemClick's performance information is derived by profiling and is dynamically annotated on a coarse granularity, leveraging the well-defined element boundaries of Click.

A speedup of two orders of magnitude can easily be achieved using SystemClick generated models with performance annotations compared to instruction set simulation. Contrary to related work, which often focuses on benchmark kernels, the figures include the full system function. The quality of the simulation results is within a 2.5% margin and depends on the match of traffic characteristics between profiling and simulation.

Our methodology and tools are not bound to packet-oriented communication interfaces. They can be utilized for the design of network processing systems as shown and discussed elsewhere [172, 22].

6.2 Architecture exploration

Following our application-driven methodology, we first analyzed and characterized the communication interface domain:

- Usage-based classification of communication interfaces — A set of more than 30 contemporary SoC architectures from three communication-intense application domains was analyzed and classified in its interface specifics. Being the first of its kind, the deployment survey discusses interface diversity and existing approaches towards flexibility.

- Characterization and comparison of packet-oriented interfaces — Using a representative selection of protocols, a common structure and a set of elementary tasks were derived, and similarities in occurrence and deployment were discussed. While putting its emphasis on the transaction and the data link layers of the protocols, the comparison is the first that takes real-time critical interface protocols into account.

Applying our findings together with method and tools summarized in the previous section, we implemented a set of interfaces in Click and conducted an architecture exploration for a programmable communication interface using building blocks of the NOVA hard- and software platform. Our NOVA platform implements the state-of-the-art in concurrent platforms while putting more emphasis on modularity and programmability (cf. Sec. 5.5.1).

We quantified the performance of our interfaces along five primary design axes: type of the embedded standard cores, hardware accelerators, application-specific processing cores, number of processing elements, and communication topology (cf. Sec. 5.6). Based on these insights, we proposed a NOVA instance with up to four PEs for the protocol processing software and a specialized element (SPE), that comprises protocol-agnostic hardware accelerators (cf. Fig. 5.20). Our main findings were:

- Performance of a programmable interface — An implementation on a modular and protocol-agnostic platform meets the performance requirements:

  a) *without restrictions* for Gigabit Ethernet and wireless Lan 802.11 a/b/g. These protocols can be handled fully in software using 1-2 NOVA processing elements running at clock speeds between 150 and 360 MHz. Considerably improving related work protocol deadlines of wireless LAN can be met by a software implementation of the control frame processing.

b) *with restrictions* for PCI Express, Hypertransport, andRapidIO. Full throughput of up to 5Gb/s can be achieved for packets of 206 Bytes or larger on a 4PE NOVA system running at only 360 MHz.

But even with the performance restriction for small packets, PEX, HT, and RIO are fully functional since their flow control mechanism will effectively throttle the throughput in compliance with the standard. This is different from a known software-based implementation of PCI that violates the physical speed on the wire.

- Area footprint of a programmable interface — Compared to the area of individual IP modules and depending on the use case we observed:

  a) Compared to *a single IP interface*, the programmable solution is larger by factors between 1.59 (PCI Express, 4PEs) and 1.64 (GbE, 1PE).

  b) *Physically multiplexed IP modules.* The combination of PCI Express with HT or RIO has an area advantage of 7% to 12%. A combined solution for all three interfaces has an area advantage of 37%.

6.3 Is a programmable solution feasible?

The implementation of communication interfaces on a protocol-agnostic programmable platform is feasible as our results show.

Mapping link and transaction layers of packet-oriented communication interfaces onto a multiprocessor platform, we found that all of our protocols can be handled in software. The performance and real-time requirements for medium-throughput protocols can be met running on processing elements with moderate clock frequencies (150-360 MHz). Performance restrictions apply for high-throughput protocols since these protocols throttle the throughput for small packets. In such cases, the full line speed of up to 5Gb/s is not achieved.

This means that a programmable solution as proposed can be deployed in settings that do not require highest packet throughput. In contrast to such embedded settings, high-performance computing and graphics application would call for even higher throughput rates, i.e., support for multiple parallel lanes would be required. Most of the SoCs, which were discussed in Chapter 2, have aggregated throughput requirements, which are far lower than those 5Gb/s of a single IO interface. The whole IXP1200 network processor system, for instance, targets line rates of 2.5 Gb/s. Potential deployment scenarios are:

- Residential gateways — A wireless access point, which, apart from wireless Lan and Ethernet interfaces, also integrates a PCI express interface, would not require more than a few hundred Mb/s throughput (150-600 Mb/s).

- Network processors — PEX, RIO, or HT may be deployed as control-plane and switch-fabric interfaces in network processors. At the control plane, short messages would not appear very frequently compared with the forwarding throughput. For the switch fabric interface, on the other hand, fixed packet sizes of 64B or larger would be a reasonable assumption.

The *area footprint* of such a solution depends on the use case. If we compare our flexible solution to individual IP modules, we find the programmable solution to be larger by about 60%. Integrating a flexible implementation as a replacement for a mature and single protocol interface, therefore, is not beneficial in terms of the required area. The flexible implementation if considered 'standalone' is more expensive.

However, if multiple mutually exclusive protocols must be supported or a protocol is immature and still evolving, the flexible solution would be an alternative to the physical multiplexing of several dedicated IP modules. Furthermore, if an application under-utilizes a processing element of the SoC, its spare cycles may be used to execute flexible interface functions. This lowers the area footprint. Another Ethernet interface, for instance, would require only the extra SPE while the GPE functions are mapped onto the core's PEs, which is there, anyway.

In summary, we are convinced that this strongly indicates the feasibility of a flexible solution in terms of performance and area. In embedded settings with moderate throughput requirements even the current implementation is advantageous.

6.4 Directions for further research

While identifying the interface problem and laying a foundation for the systematic development of flexible packet-oriented communication interfaces, several starting points for further research remain, including the following:

- A better performance/area trade-off may be achieved by the optimization of the application description and the platform mapping. We found, e.g., the applications's granularity to be performance-limiting. Also, platform optimizations depend on profiling feedback. A tight integration and the (potentially) automated generation of profiling data could shorten the feedback cycle for hard- and software optimizations.

- Results can also be improved by deviation from the general-purpose instruction set. Maturing the programmability of ASIPs (e.g, their compiler support) therefore would enable their integration as building blocks of a fully programmable platform.

- Focusing on performance and area, our exploration did not consider power consumption. Including power consumption as an additional exploration objective would complement our analysis and might provide insights for application domains such as handheld devices.

- Adopting a depth-first approach, we could demonstrate value for packet-oriented interfaces. Currently, our methodology is applied to further communication protocols in this domain such as the emerging 802.11n protocol family. Further potential may exist in other interface classes, e.g., the memory interfaces.

Bibliography

[1] Advanced Switching Special Interest Group. PCI Express Advanced Switching Specification, Draft 1.0. www.asi-sig.org, Sept. 2003.

[2] Agilent Technologies. JTC 003: Mixed packet size throughput. *The Journal of Internet Test Methodologies*, pages 16–18, Sept. 2004.

[3] J. R. Allen, B. M. Bass, C. Basso, R. H. Boivie, J. L. Calvignac, G. T. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, A. Kind, J. F. Logan, M. Peyravian, M. A. Rinaldi, R. K. Sabhikhi, M. S. Siegel, and M. Waldvogel. IBM PowerNP Network Processor: Hardware, software, and applications. *IBM Journal of Research and Development*, 47(2-3):177–194, Mar. 2003.

[4] Apple Developer Connection. Firewire overview. www.apple.com/firewire.

[5] ARM Holdings plc. ARM7TDMI 32-bit RISC core performance characteristics. http://www.arm.com/products/CPUs/ARM7TDMI.html, March 2009.

[6] ATM forum. UTOPIA Specification Level 1, Version 2.01, af-phy-0017.000. www.atmforum.com, www.ipmplsforum.org, March 1995.

[7] ATM forum. Utopia Specification Level 2, Version 1.0, af-phy-0039.000. www.atmforum.com, www.ipmplsforum.org, June 1995.

[8] ATM forum. Utopia Level 3 Physical Layer Interface Specification, af-phy-0136.000. www.atmforum.com, www.ipmplsforum.org, Nov 1999.

[9] ATM forum. Utopia Level 4 Specification, AF-PHY-0144.001. www.atmforum.com, www.ipmplsforum.org, March 2000.

[10] F. Balarin, M. Chiodo, H. Hsieh, B. Tabbara, A. Sangiovanii-Vincentelli, A. Jurecska, K. Lavagno, C. Passerone, and K. Suzuki. *Hardware-Software Co-design of embedded systems - The Polis approach.* Kluwer Academic Publishers, 1997.

[11] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[12] M. Barron. Nomadik roaming in 2003: STMicroelectronics introduces its family of cell-phone chips. *Microprocessor Report*, 2/24/03-01, 2003.

[13] M. Barron. STs reconfigurable greenfield - infrastructure chip combines ARM926, eDRAM, and M2000 eFPGA. *Microprocessor Report*, 4/4/05-02, April 2005.

[14] M. Barron. A MIPS32 core for your MCU. *Microprocessor Report*, 12/17/07-01, December 2007.

[15] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the Association for Computing Machinery*, 22(2):248–260, April 1975.

[16] D. Bees and B. Holden. Making interconnects more flexible. *EE Times*, Sept. 2003.

[17] F. Belina, D. Hogrefe, and A. Sarma. *SDL with applications from protocol specification.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[18] L. Benini and D. Bertozzi. Network-on-chip architectures and design methods. *Computers and Digital Techniques, IEE Proceedings -*, 152(2):261–272, 2005.

[19] M. Bhardwaj, C. Briggs, A. Chandrakasan, C. Eldridge, J. Goodman, T. Nightingale, S. Sharma, G. Shih, D. Shoemaker, A. Sinha, R. Venkatesan, J. Winston, and W. Zhou. A 180MS/s, 162Mb/s wideband three-channel baseband and MAC processor for 802.11 a/b/g. In C. Briggs, editor, *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 454–609 Vol. 1, 2005.

[20] A. Bianco, R. Birke, D. Bolognesi, J. Finochietto, G. Galante, M. Mellia, M. Prashant, and F. Neri. Click vs. Linux: two efficient open-source IP network stacks for software routers. In R. Birke, editor, *High Performance Switching and Routing, 2005. HPSR. 2005 Workshop on*, pages 18–23, 2005.

[21] O. Blaurock. A SystemC-based modular design and verification framework for C-model reuse in a HW/SW-codesign design flow. In *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*, pages 838–843, 2004.

[22] H.-M. Blüthgen, C. Sauer, D. Langen, M. Gries, and W. Raab. Application-driven design of cost-efficient communications platforms. In A. B. Cremers, R. Manthey, P. Martini, and V. Steinhage, editors, *INFORMATIK 2005 - Informatik LIVE! Beiträge der 35. Jahrestagung der Gesellschaft für Informatik*, volume 67 of *LNI*, pages 314–318. GI, September 2005.

[23] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *Computer*, 33(5):59–67, 2000.

[24] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/ERL M05/21, EECS Department, University of California, Berkeley, 2005.

[25] R. Budruk, D. Anderson, and T. Shanley. *PCI Express System Architecture.* Addision-Wesley, Sept. 2003.

[26] L. Cai, A. Gerstlauer, and D. Gajski. Retargetable profiling for rapid, early system level design space exploration. In *DAC*, 2004.

[27] J. Calvez, D. Heller, and O. Pasquier. Uninterpreted co-simulation for performance evaluation of hw/sw systems. In *Hardware/Software Co-Design, 1996. (Codes/CASHE '96), Proceedings., Fourth International Workshop on*, pages 132–139, 1996.

[28] R. Camposano. EDA Challenges in the Road Ahead. *2nd Annual IEEE Northeast Workshop on Circuits and Systems (NewCAS)*, 2004.

[29] A. Castonguay and Y. Savaria. A HyperTransport chip-to-chip interconnect tunnel developed using SystemC. In Y. Savaria, editor, *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pages 264–266, 2005.

[30] A. Castonguay and Y. Savaria. Architecture of a HyperTransport tunnel. In Y. Savaria, editor, *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4 pp.–, 2006.

[31] P. Chandra, S. Lakshmanamurthy, and R. Yavatkar. Intel IXP2400 network processor: A 2nd generation Intel NPU. In *Network Processor Design: Issues and Practices*, volume 1, chapter 13. Morgan Kaufmann, 2002.

[32] X. Chang. Network simulations with OPNET. In *Simulation Conference Proceedings, 1999. Winter*, volume 1, pages 307–314 vol.1, 1999.

[33] B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor PC router. Proceedings of the 2001 USENIX Annual Technical Conference (USENIX'01), Boston, MA, June 2001.

[34] E. Cheung, H. Hsieh, and F. Balarin. Framework for fast and accurate performance simulation of multiprocessor systems. In H. Hsieh, editor, *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 21–28, 2007.

[35] CoFluent Design. CoFluent Studio. www.cofluentdesign.com.

[36] T. Cooklev. *Wireless Communication Standards: A study of IEEE 802.11, 802.15, and 802.16.* IEEE Standards Wireless Networks Series. Standards Information Network, IEEE Press, 2004.

[37] Coware. ConvergenSC. www.coware.com.

[38] N. Cravotta. RapidIO vs. Hypertransport. *EDN*, June 2002.

[39] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu. A next-generation design framework for platform-based design. In *Conference on Using Hardware Design and Verification Languages (DVCon)*, February 2007.

[40] N. de Wet and P. Kritzinger. Using UML Models for the Performance Analysis of Network Systems. *Computer Networks*, 49(5):627–642, 2005.

[41] A. DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, 2000.

[42] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli. A Platform-Based Taxonomy for ESL Design. *IEEE Design and Test of Computers*, 23(5):359– 374, September 2006.

[43] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D. Sicker, and D. Grunwald. MultiMAC - an adaptive MAC framework for dynamic radio networking. In M. Neufeld, editor, *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 548–555, 2005.

[44] S. Dutta, R. Jensen, and A. Riekmann. Viper: A Multiprocessor SoC for Advanced Set-Top Box and Digital TV Systems. IEEE Design & Test of Computers, Sept. 2001.

[45] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[46] I. Elhanany, K. Busch, and D. Chiou. Switch fabric interfaces. *Computer*, 36(9):106–108, 2003.

[47] D. Foland. Ubicom MASI - wireless network processor. *15th Hotchips Conference, Palo Alto, CA*, Aug. 2003.

[48] S. Fuller. *RapidIO - The embedded system interconnect.* Wiley & Sons, 2005.

[49] M. S. Gast. *802.11 Wireless Networks.* O'REILLY, 2nd edition, April 2005.

[50] GDA Technologies Inc. HyperTransport Cave Core. Product Specification, Spetember 2003.

[51] G. Giacalone, T. Brightman, A. Brown, J. Brown, J. Farrelland, R. Fortino, T. Franco, A. Funk, K. Gillespie, E. Gould, D. Husak, E. McLellan, B. Peregoy, D. Priore, M. Sankey, P. Stropparo, and J. Wise. A 200 MHz digital communications processor. In *IEEE International Solid-State Circuits Conference (ISSCC)*, Feb. 2000.

[52] A. Girault, B. Lee, and E. Lee. Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(6):742–760, 1999.

[53] P. N. Glaskowsky. Intel Beefs up Networking Line: New Chips Help IXP Family Reach New Markets. *Microprocessor Report*, March 2002.

[54] S. Graf. Expression of time and duration constraints in SDL. In *3rd SAM Workshop on SDL and MSC, University of Wales Aberystwyth*, number 2599 in LNCS, June 2002.

[55] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal, Elsevier*, 38(2):131–183, December 2004.

[56] M. Gries and K. Keutzer. *Building ASIPs: The Mescal Methodology.* Springer, 2005.

[57] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer. Exploring trade-offs in performance and programmability of processing element topologies for network processors. In *Network Processor Design: Issues and Practices*, volume 2. Morgan Kaufmann, 2003.

[58] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[59] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):24, 2007.

[60] T. Halfhill. Intel network processor targets routers. *Microprocessor Report*, 13(12), Sept. 1999.

[61] T. Halfhill. Triscend revs up for motors. *Microprocessor Report*, (9/15/03-02), September 2003.

[62] T. Halfhill. Ubicoms new NPU stays small,. *Microprocessor Report*, Apr. 2003.

[63] T. Halfhill. ARC 700 secrets revealed. *Microprocessor Report*, (6/21/04-01), June 2004.

[64] T. Halfhill. Tensilica upgrades Xtensa Cores. *Microprocessor Report*, (12/4/06-02), April 2006.

[65] M. Hannikainen, J. Knuutila, T. Hamalainen, and J. Saarinen. Using SDL for implementing a wireless medium access control protocol. In *Multimedia Software Engineering, 2000. Proceedings. International Symposium on*, pages 229–236, 2000.

[66] D. Harel. StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

[67] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 642–649, 2001.

[68] S. Hasan, A. Landry, Y. Savaria, and M. Nekili. Design constraints of a hypertransport-compatible network-on-chip. In A. Landry, editor, *Circuits and Systems, 2004. NEWCAS 2004. The 2nd Annual IEEE Northeast Workshop on*, pages 269–272, 2004.

[69] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. chan Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan. Effects of detail in wireless network simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 3–11, Phoenix, Arizona, USA, January 2001. USC/Information Sciences Institute, Society for Computer Simulation.

[70] T. Henriksson, H. Eriksson, U. Nordqvist, P. Larsson-Edefors, and D. Liu. VLSI implementation of CRC-32 for 10 Gigabit Ethernet. In *8th IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS)*, Sept. 2001.

[71] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[72] A. I. Holub. *C+ C++: programming with objects in C and C++*. McGraw-Hill, Inc., New York, NY, USA, 1992.

[73] HomePNA. The Home Phone line Networking Alliance (HPNA) webseite. www.homepna.org.

[74] I.-P. Hong, Y.-J. Lee, S.-J. Chun, Y.-S. Lee, and J. Joung. Multi-threading processor architecture for wireless lan mac controller. In Y.-J. Lee, editor, *Consumer Electronics, 2005. ICCE. 2005 Digest of Technical Papers. International Conference on*, pages 379–380, 2005.

[75] C. Hunter, J. Camp, P. Murphy, A. Sabharwal, and C. Dick. A flexible framework for wireless medium access protocols. In J. Camp, editor, *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, pages 2046–2050, 2006.

[76] HyperTransport Consortium. The HyperTransport Consortium website. www.hypertransport.org.

[77] Hypertransport Consortium. HyperTransport I/O link specification, rev. 1.10. www.hypertransport.org, Aug. 2003.

[78] Hypertransport Consortium. HyperTransport I/O technology comparison with traditional and emerging I/O technologies. white paper, www.hypertransport.org, June 2004.

[79] Hypertransport Consortium. HyperTransport I/O link specification, rev. 3.0a. www.hypertransport.org, Nov. 2006.

[80] E. Hyun and K.-S. Seong. Design and verification for pci express controller. In K.-S. Seong, editor, *Information Technology and Applications, 2005. ICITA 2005. Third International Conference on*, volume 1, pages 581–586 vol.1, 2005.

[81] IEEE 1394. The IEEE1394 Trade Association website. www.ieee1394ta.org.

[82] IEEE Standards. standards.ieee.org.

[83] IEEE802.3. Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. standards.ieee.org/getieee802/802.3.html, 2002.

[84] M. Iliopoulos and T. Antonakopoulos. Optimised reconfigurable mac processor architecture. In T. Antonakopoulos, editor, *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*, volume 1, pages 253–258 vol.1, 2001.

[85] IrDA. The Infrared Data Association website. www.irda.org.

[86] ITU-T. Specification and Description Language (SDL). Recommendation Z-100, www.itu.ch - Electronic Bookshop, Geneva, 1999.

[87] ITU-T. SDL combined with UML. Recommendation Z-100, www.itu.ch - Electronic Bookshop, Geneva, 2001.

[88] J. W. Janneck and R. Esser. Higher-order petri net modelling: techniques and applications. In *CRPIT '02: Proceedings of the conference on Application and theory of petri nets*, pages 17–25, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[89] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *Computers and Digital Techniques, IEE Proceedings-*, 152(2):114–129, 2005.

[90] JEDEC - The Joint Electron Devices Engineering Council. Double Data Rate (DDR) SDRAM Specification. www.jedec.org /download/search/JESD79F.pdf (DDR1), JESD79-2E.pdf (DDR2), JESD2008.pdf (DDR2-1066), JESD79-3B.pdf (DDR3).

[91] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts of *EATCS Monographs in Computer Science.* Springer-Verlag, 1992.

[92] A. Jerraya, A. Jerraya, H. Tenhunen, and W. Wolf. Guest editors' introduction: Multiprocessor systems-on-chips. *Computer*, 38(7):36–40, 2005.

[93] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77.* North-Holland Publishing Co., 1977.

[94] H. Kalte, M. Porrmann, and U. Rückert. A prototyping platform for dynamically reconfigurable system-on-chip designs. In *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC), Hamburg, Germany*, 2002.

[95] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 876–881 Vol. 2, 2005.

[96] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW performance estimation framework for early System-Level-Design using fine-grained instrumentation. In *Design, Automation & Test in Europe (DATE)*, Munich, Germany, March 2006.

[97] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1523–1543, 2000.

[98] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. 11th Int. Conf. on Application-specific Systems, Architectures and Processors, Zurich, Switzerland, 1997.

[99] L. Kleinrock. *Queueing Systems, Volume I: Theory.* John Wiley & Sons, Inc., 1975.

[100] E. Kohler. *The Click modular router.* PhD thesis, MIT, Nov. 2000.

[101] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable tcp in the prolac protocol language. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 3–13, New York, NY, USA, 1999. ACM Press.

[102] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[103] C. Kulkarni, G. Brebner, and G. Schelle. Mapping a domain specific language to a platform fpga. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 924–927, 2004.

[104] C. Kulkarni, M. Gries, C. Sauer, and K. Keutzer. Programming challenges in network processor deployment. In *Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, October 2003.

[105] K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(6):768–783, 2001.

[106] D. Langen, J.-C. Niemann, M. Porrmann, H. Kalte, and U. Rückert. Implementation of a RISC Processor Core for SoC Designs FPGA Prototype vs. ASIC Implementation. 2002.

[107] M. LaPedus. AMCC tops IBM, Intel, Moto in net-processor share. *EE Times*, Jan. 2003.

[108] P. Latkoski, T. Janevski, and B. Popovski. Modelling and simulation of ieee 802.11a wireless local area networks using sdl. In *Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean*, pages 680–683, 2006.

[109] L. Lavagno, G. Martin, and B. Selic, editors. *UML for real: design of embedded real-time systems.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[110] J.-Y. Le Boudec and P. Thiran. *Network Calculus.* Springer Verlag LNCS, 2001.

[111] R. Le Moigne, O. Pasquier, and J.-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *Design, Automation and Test in Europe (DATE)*, volume 3, pages 82–87 Vol.3, 2004.

[112] P. Leblanc and I. Ober. Comparative case study in sdl and uml. In *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*, pages 120–131, 2000.

[113] E. Lee and S. Neuendorffer. Concurrent models of computation for embedded software. *Computers and Digital Techniques, IEE Proceedings -*, 152(2):239–250, 2005.

[114] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[115] K. Lee, G. Coulson, G. Blair, A. Joolia, and J. Ueyama. Towards a generic programming model for network processors. In G. Coulson, editor, *Networks, 2004. (ICON 2004). Proceedings. 12th IEEE International Conference on*, volume 2, pages 504–510 vol.2, 2004.

[116] S. Leibson and J. Kim. Configurable processors: a new era in chip design. *Computer*, 38(7):51–59, 2005.

[117] C. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee. Standards for system-level design: practical reality or solution in search of a question? In P. Schaumont, editor, *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pages 576–583, 2000.

[118] M. Levy. Multicore multiplies the challenges: New consortium defines specs for multicore communication api. *Microprocessor Report*, 9/5/06-01, 2006.

[119] Y. Liu and B. Liu. Mac implementation with embedded system. In *ASIC, 2003. Proceedings. 5th International Conference on*, volume 2, pages 757–760 Vol.2, 2003.

[120] A. Lodi, A. Cappelli, M. Bocchi, C. Mucci, M. Innocenti, C. De Bartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma, and R. Guerrieri. XiSystem: a XiRisc-based SoC with reconfigurable IO module. *Solid-State Circuits, IEEE Journal of*, 41(1):85–96, 2006.

[121] G. Martin and S. Leibson. *Commercial configurable processors and the MESCAL approach*, chapter 8, pages 281–310. Springer, 2005.

[122] P. Marwedel. Embedded software: how to make it efficient? In *Digital System Design, 2002. Proceedings. Euromicro Symposium on*, pages 201–207, 2002.

[123] Mathlab and Simulink homepage. www.mathworks.com.

[124] Mentor Graphics. Seamless and System Architect tools. www.mentor.com.

[125] R. Merritt. Dueling I/O efforts gear up to revamp comms. *EE Times*, Oct. 2003.

[126] A. Mihal. *Deploying Concurrent Applications on Heterogeneous Multiprocessors.* PhD thesis, University of California, Berkeley, January 2006.

[127] A. Mihal, C. Kulkarni, C. Sauer, K. Vissers, M. W. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, and S. Malik. *Developing Architectural Platforms: A Disciplined Approach*, volume 19, pages 6–16. IEEE Design and Test of Computers, 2002.

[128] A. Mihal, C. Sauer, and K. Keutzer. Designing a sub-risc multi-gigabit regular expression processor. Technical Report UCB/EECS-2006-119, University of California at Berkeley, September 2006.

[129] B. Moller-Pedersen. SDL combined with UML. *Telektronikk, Languages for Telecommunication Applications*, 4, 2000.

[130] W. Müller, W. Rosenstiel, and J. Ruf, editors. *SystemC: methodologies and applications.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[131] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[132] M. Neufeld, A. Jain, and D. Grunwald. NsClick: bridging network simulation and deployment. In *Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*, pages 74–81. ACM Press, 2002.

[133] X. Nie, L. Gazsi, F. Engel, and G. Fettweis. A new network processor architecture for high-speed communications. In *Signal Processing Systems, 1999. SiPS 99. 1999 IEEE Workshop on*, pages 548–557, 1999.

[134] P. Noel, F. Zarkeshvari, and T. Kwasniewski. Recent advances in high-speed serial i/o trends, standards and techniques. In *Electrical and Computer Engineering, 2005. Canadian Conference on*, pages 1292–1295, 2005.

[135] NP Forum. The Network Processor Forum (NPF) website (merged in 2006 with the Optical Internetworking Forum (OIF). web.archive.org/web/20060202140159/http://npforum.org, 2006.

[136] Object Management Group. UML resource page. www.uml.org.

[137] OI Forum. The Optical Internetworking Forum (OIF) website. www.oiforum.com.

[138] OPNET Technologies. OPNET Modeler and Simulator. www.opnet.com.

[139] M. Ozdemir and A. McDonald. A queuing theoretic model for IEEE 802.11 DCF using RTS/CTS. In *Local and Metropolitan Area Networks, 2004. LANMAN 2004. The 13th IEEE Workshop on*, pages 33–38, 2004.

[140] G. Panic, D. Dietterle, Z. Stamenkovic, and K. Tittelbach-Helmrich. A system-on-chip implementation of the IEEE 802.11a MAC layer. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pages 319–324, 2003.

[141] T. Parks. *Bounded Scheduling of Process Networks.* PhD thesis, University of California at Berkeley, December 1995.

[142] J. Parssinen, N. von Knorring, J. Heinonen, and M. Turunen. UML for protocol engineering-extensions and experiences. In *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*, pages 82–93, 2000.

[143] H. D. Patel and S. K. Shukla. *SystemC Kernel Extensions For Heterogenous System Modeling: A Framework for Multi-MoC Modeling & Simulation.* Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[144] P. Paulin, J. Frehel, M. Harrand, E. Berrebi, C. Liem, F. Nacabul, and J.-C. Herluison. High-level synthesis and codesign methods: An application to a Videophone Codec. In *Design Automation Conference, 1995, with EURO-VHDL, Proceedings EURO-DAC '95., European*, pages 444–451, 1995.

[145] P. Paulin, C. Pilkington, and E. Bensoudane. StepNP: a system-level exploration platform for network processors. *Design & Test of Computers, IEEE*, 19(6):17–26, 2002.

[146] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, and G. Nicolescu. Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(7):667–680, 2006.

[147] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu. Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pages 48–53, 2004.

[148] L. Paulson. The ins and outs of new local I/O trends. *IEEE Computer*, 36(7):13–16, 2003.

[149] PCI Special Interest Group. The PCI standards (PCI, PCI-X, PEX) specification industry consortium website. www.pcisig.com.

[150] PCI Special Interest Group. PCI Express base specification, rev. 1.0a. www.pcisig.com, Apr. 2003.

[151] H. P. Peixoto and M. F. Jacome. Algorithm and architecture-level design space exploration using hierarchical data flows. In *IEEE International Conference on Applications-Specific Systems, Architectures and Processors (ASAP)*, pages 272–282, 1997.

[152] Philips Semiconductors. I2S Bus Specification, now part of NXP. www.nxp.com/acrobat_download/various/I2SBUS.pdf, June 1996.

[153] A. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *Computers, IEEE Transactions on*, 55(2):99–112, 2006.

[154] A. Pimentel, S. Polstra, F. Terpstra, A. van Halderen, J. Coffland, and L. Hertzberger. Towards efficient design space exploration of heterogeneous embedded media systems. In *Embedded processor design challenges. Systems, architectures, modeling, and simulation - SAMOS*, volume 2268 of *LNCS*, pages 57–73. Springer Verlag, 2002.

[155] A. D. Pimentel, P. Lieverse, P. van der Wolf, L. Hertzberger, and E. F. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, Nov. 2001.

[156] Ptolemy II project homepage. ptolemy.eecs.berkeley.edu.

[157] L. Pucker. Is the wireless industry ready for component based system development? *Communications Magazine, IEEE*, 43(6):s6–s8, 2005.

[158] QDR SRAM. The QDR consortium homepage. www.qdrconsortium.org.

[159] Rambus Inc. homepage. www.rambus.com.

[160] RapidIO Trade Association. homepage. www.rapidio.org.

[161] RapidIO Trade Association. RapidIO interconnect specification, rev. 1.2. www.rapidio.org, June 2002.

[162] R. Recio. Server I/O networks past, present, and future. *ACM SIGCOMM Workshop on Network-I/O Convergence*, Aug. 2003.

[163] S.-A. Reinemo, T. Skeie, T. Sdring, O. Lysne, and O. Trudbakken. An overview of QoS capabilities in InfiniBand, Advanced Switching Interconnect, and Ethernet. *IEEE Communications Magazine*, 44(7):32–38, 2006.

[164] M. Ruggiero, F. Angiolini, F. Poletti, D. Bertozzi, L. Benini, and R. Zafalon. Scalability analysis of evolving SoC interconnect protocols. In S. Albl, editor, *International Symposium on System-on-Chip, 2004*, November 2004.

[165] S. Samadi, A. Golomohammadi, A. Jannesari, M. Movahedi, B. Khalaj, and S. Ghammanghami. A novel implementation of the IEEE 802.11 medium access control. In A. Golomohammadi, editor, *Intelligent Signal Processing and Communications, 2006. ISPACS '06. International Symposium on*, pages 489–492, 2006.

[166] C. Sauer, M. Gries, and S. Dirk. Hard- and software modularity of the NOVA MPSoC platform. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1102–1107, San Jose, CA, USA, 2007. EDA Consortium.

[167] C. Sauer, M. Gries, S. Dirk, J.-C. Niemann, M. Porrmann, and U. Rückert. A Lightweight NoC for the NOVA Packet Processing Platform. In *Workshop on Future Interconnects and Network-on-chip*, along with DATE. http://async.org.uk/noc2006, 2006.

[168] C. Sauer, M. Gries, J. Gomez, S. Weber, and K. Keutzer. Developing a Flexible Interface for RapidIO, Hypertransport, and PCI-Express. In *Parallel Computing in Electrical Engineering, 2004. PARELEC 2004. International Conference on*, pages 129–134, 2004.

[169] C. Sauer, M. Gries, J. I. Gomez, and K. Keutzer. *Towards a Flexible Network Processor Interface for RapidIO, Hypertransport, and PCI-Express*, volume 3, chapter 4, pages 55–80. Morgan Kaufmann Publishers, 2005.

[170] C. Sauer, M. Gries, C. Kulkarni, and K. Keutzer. Analyzing the peripheral-processor interaction in embedded systems. Technical report, University of California, Berkeley, July 2003.

[171] C. Sauer, M. Gries, and H.-P. Löb. SystemClick: a domain-specific framework for early exploration using functional performance models. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 480–485, New York, NY, USA, 2008. ACM.

[172] C. Sauer, M. Gries, J.-C. Niemann, M. Porrmann, and M. Thies. Application-driven development of concurrent packet processing platforms. In *Parallel Computing in Electrical Engineering (PARELEC)*, pages 55–61, 2006.

[173] C. Sauer, M. Gries, and S. Sonntag. Modular domain-specific implementation and exploration framework for embedded software platforms. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 254–259, 2005.

[174] C. Sauer, M. Gries, and S. Sonntag. Modular reference implementation of an IP-DSLAM. *10th IEEE Symposium on Computers and Communications (ISCC)*, Cartagena, Spain, June 2005.

[175] D. Sciuto, F. Salice, L. Pomante, and W. Fornaciari. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *CODES*, 2002.

[176] SDL forum. The System Description Language forum homepage. www.sdl-forum.org.

[177] N. Shah. Understanding Network Processors. Master's thesis, Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2001.

[178] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A programming model for the Intel IXP1200. In *Network Processor Design: Issues and Practices*, volume 2. Morgan Kaufmann, 2003.

[179] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. NP-Click: a productive software development approach for network processors. *IEEE Micro*, 24(5):45–54, 2004.

[180] S. A. Shah, S. Khanvilkar, and A. Khokhar. RapidIO traffic management and flow arbitration protocol. *Communications Magazine, IEEE*, 44(7):45–52, 2006.

[181] S. A. Shah, H. Mouftah, and J. Saunders. Interconnect and fabric technology standards. *Communications Magazine, IEEE*, 44(7):22–23, 2006.

[182] H. Shahriari and R. Jalili. Using CSP to model and analyze Transmission Control Protocol vulnerabilities within the broadcast network. In *Networking and Communication, 2004. INCC 204. International Conference on*, pages 42–47, 2004.

[183] A. K. Sharma. *Advanced Semiconductor Memories: Architectures, Designs, and Applications.* Wiley-IEEE Press, Oct. 2002.

[184] T. Shono, Y. Shirato, H. Shiba, K. Uehara, K. Araki, and M. Umehira. IEEE 802.11 wireless LAN implemented on software defined radio with hybrid programmable architecture. *Wireless Communications, IEEE Transactions on*, 4(5):2299–2308, 2005.

[185] R. Short and B. Stuart. Windows XP crash data. Driver Development Keynote, WinHEC, Anaheim, 2001.

[186] F. Slomka, M. Dorfel, and R. Munzenberger. Generating mixed hardware/software systems from SDL specifications. In *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, pages 116–121, 2001.

[187] N. Smyth, M. McLoone, and J. McCanny. Reconfigurable hardware acceleration of WLAN security. In M. McLoone, editor, *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*, pages 194–199, 2004.

[188] I. Sommerville. *Software Engineering.* International Computer Science Series. Pearson Addison Wesley, 8th edition, 2006.

[189] S. Sonntag, M. Gries, and C. Sauer. SystemQ: A queuing-based approach to architecture performance evaluation with SystemC. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS V)*, volume 3553 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 2005.

[190] S. Spitz, F. Slomka, and M. Dörfel. SDL - an annotated specification language for engineering multimedia communication systems. In *Proceedings of the Sixth Open Workshop on High Speed Networks, Stuttgart, Germany*, October 1997.

[191] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. F. Deprettere. System design using kahn process networks: The compaan/laura approach. In *DATE*, pages 340–345, 2004.

[192] Synopsys Inc. CoCentric SystemStudio. www.synopsys.com.

[193] Texas Instruments. TNETW1130 Converged Single-Chip MAC and Baseband Processor for IEEE 802.11 a/b/g. Product Sheet, www.ti.com, April 2003.

[194] The Click Modular Router Project. http://pdos.csail.mit.edu/click/.

[195] L. Thiele, S. Chakraborty, M. Gries, and S. Kunzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 880–885, 2002.

[196] W. Tibboel, V. Reyes, M. Klompstra, and D. Alders. System-Level Design Flow Based on a Functional Reference for HW and SW. In *44th ACM/IEEE Design Automation Conference (DAC)*, pages 23–28, 2007.

[197] O. Tickoo and B. Sikdar. A queueing model for finite load IEEE 802.11 random access MAC. In *Communications, 2004 IEEE International Conference on*, volume 1, pages 175–179 Vol.1, 2004.

[198] J. Trodden and D. Anderson. *HyperTransport System Architecture*. Mindshare, Inc., Addision-Wesley, Feb. 2003.

[199] M. Tsai, C. Kulkarni, C. Sauer, N. Shah, and K. Keutzer. A benchmarking methodology for network processors. In *1st Network Processor Workshop (NP1), at the 8th International Symposium on High Performance Computer Architectures (HPCA8), Boston MA*, Feb. 2002.

[200] USB Implementers Forum, Inc. homepage. www.usb.org.

[201] A. Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM), Prague, Czech Republic*, June 2001.

[202] M. Varsamou, T. Antonakopoulos, and N. Papandreou. From protocol models to their implementation: a versatile testing methodology. *IEEE Design &amp; Test of Computers*, 21(5):416–428, 2004.

[203] S. Weber. *Tiny Instruction Processors and Interconnect.* PhD thesis, University of California, Berkeley, January 2005.

[204] S. Weber, M. W. Moskewicz, M. Gries, C. Sauer, and K. Keutzer. Fast cycle-accurate simulation and instruction set generation for constraint-based descriptions of programmable architectures. In *International Conference on Hardware/Software Codesign (CODES)*, pages 18–23, September 2004.

[205] B. Wheeler and L. Gwennap. *A Guide to Network Processors.* The Linley Group, 7th edition, Dec 2005.

[206] A. Wieferink, T. Kogel, R. Leupers, et al. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In *DATE*, Apr. 2004.

[207] T. Wild, A. Herkersdorf, and R. Ohlendorf. Performance evaluation for system-on-chip architectures using trace-based transaction level simulation. In *Design, Automation and Test in Europe (DATE)*, volume 1, pages 1–6, 2006.

[208] T. Wolf and N. Weng. Runtime support for multicore packet processing systems. *Network, IEEE*, 21(4):29–37, 2007.

[209] H. Wu, Y. Liu, Q. Zhang, and Z.-L. Zhang. SoftMAC: Layer 2.5 Collaborative MAC for Multimedia Support in Multihop Wireless Networks. *Mobile Computing, IEEE Transactions on*, 6(1):12–25, 2007.

[210] H. Wu, X. Wang, Y. Liu, Q. Zhang, and Z.-L. Zhang. SoftMAC: layer 2.5 MAC for VoIP support in multi-hop wireless networks. In *Sensor and Ad Hoc Communications and Networks, 2005. IEEE SECON 2005. 2005 Second Annual IEEE Communications Society Conference on*, pages 441–451, 2005.

[211] Xilinx Inc. HyperTransport Single-Ended Slave (HTSES) Core v2.1. Product Specification, October 2003. Slices fehlen!

[212] Xilinx Inc. RapidIO Physical Layer Interface Core (DO-DI-RIO-PHY). Product Specification, December 2004.

[213] Xilinx Inc. Gigabit Ethernet MAC v6.0. Product Specification, April 2005.

[214] Xilinx Inc. PCI Express Endpoint Cores v3.0. Product Specification, December 2005. Slices fehlen!

[215] Xilinx Inc. RapidIO Logical (I/O) and Transport Layer Interface v3.0.1. Product Specification, June 2005.

[216] J. Yeong, X. Rao, M. Shajan, Q. Wang, J. Lin, and X. Qu. 802.11a MAC layer: firmware/hardware co-design. In *Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on*, volume 3, pages 1923–1928 vol.3, 2003.

[217] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, and A. Jerraya. A generic wrapper architecture for multi-processor SoC cosimulation and design. In *CODES*, Apr. 2001.

[218] R. Y. Zaghal and J. I. Khan. EFSM/SDL modeling of the original TCP standard (RFC793) and the Congestion Control Mechanism of TCP Reno. Technical Report TR2005-07-22, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, http://medianet.kent.edu/technicalreports.html, March 2005.