Application-driven Development of Concurrent Packet Processing Platforms

Christian Sauer, Matthias Gries
Access Communication Solutions (COM AC)
Infineon Technologies AG
81726 Munich, Germany
christian.sauer@infineon.com

J.-C. Niemann[1], M. Porrmann[1], M. Thies[2]
[1]Heinz Nixdorf Institute
[2]Faculty of CS, EE and Mathematics
University of Paderborn, Germany
niemann@hni.upb.de

Abstract

We have developed an application-driven methodology for implementing parallel and heterogeneous programmable platforms. We deploy our flow for network access platforms where we have to trade off flexibility against costs and performance. Our methodology therefore focuses on characterizing the application domain as early as possible. With this input, we can narrow the design space to one major design trajectory that starts with the most flexible solution and refines the platform architecture systematically to meet performance and costs constraints. Our flow includes an efficient path to implementation in hardware and software. The software implementation framework takes a modular application description and generates code for embedded processors that can easily be ported to different platforms and used for profiling. Different communication architectures, co-processors, and specializations of programmable processing elements can be derived from profiling results to affect the platform hardware. A DSL Access Multiplexer (DSLAM) is used as an example throughout the paper to depict the different phases of our design process.

1. Introduction

Why do we need highly concurrent packet processing platforms for the network access? We recognize that network services are converging. The future "triple play" scenario combines voice, data, and video. Adding video services will significantly increase the bandwidth demand in access networks and result in a processing bottleneck. One may argue that high-end network processors (NPUs) like Intel's IXP 2xxx line can handle this high per-packet computation. With respect to performance this statement is true, but it neglects the tight constraints on costs access network equipment has to cope with. In this application domain it is mandatory to find a feasible trade off between flexibility, performance, and costs of the platform.

We have developed a design flow for access network platforms that takes the importance of the application domain into account. One major focus of our methodology is the proper characterization of the domain and implementation of a representative reference application. By this means, a flexible platform architecture template can be customized and the complexity of the design space can significantly be reduced to meet costs constraints. Our design methodology can be described by the following five phases. They can be seen as a realization of the Mescal methodology [5] 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 since we anticipate the use of our SoC as an open platform that can be programmed by the customer.

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.

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 most often used, they can potentially hide performance bottlenecks of a heterogeneous processing platform.

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

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

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. 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 therefore use this phase to derive a good starting point for exploring the architecture design space.

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 we use a set of embedded general-purpose processors and other commodity parts. Such a generic solution will be 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 and power. As a result, a virtual platform prototype with an efficient programming environment is implemented.

5. *Implementation and Deployment.* Implement the platform and provide a code generation framework to ease the deployment with efficient programming abstractions.

The virtual prototype serves as executable specification for the actual platform hardware and software implementation and serves as golden model for verification. In addition, it can make sense to revise an internally used programming abstraction after the exploration if a programmable architecture is released as an open platform to the customer.

The main contribution of this paper is the presentation of a comprehensive application-driven design flow. That means, the exploration of the platform's design space particularly depends on systematic analysis phases of the application domain and reference benchmarks. In the following, we present the phases of our design flow and illustrate them looking at future IP-based DSL Access Multiplexers.

2. Application Domain Analysis

The goal is to understand the application domain and derive design requirements and system environments for benchmarking. A comprehensive analysis based on customer feedback, standards, and trends in research enables us to define a representative set of applications and setups.

A new application area where benchmarks have to be developed is packet processing in access networks. We use one example throughout this paper: Digital Subscriber Line Access Multiplexers (DSLAMs). DSLAMs link the customer side of the network with the service provider. Most of today's equipment will become obsolete and be replaced by highly reliable DSLAMs that become the central access points for all narrow- and broadband services; separate voice, data, and video networks are converging. DSLAMs are built modularly out of several line cards aggregating individual xDSL lines, trunk cards aggregating multiple line cards and providing access to the service provider network, and a backplane or dedicated links connecting line and trunk cards, as shown in Figure 1.

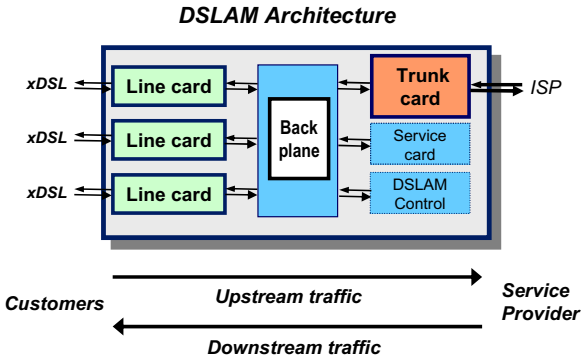

Figure 1. DSLAM system architecture.

2.1. Trends

The design trade-offs to be investigated are not bound to the architecture of a line card or trunk card, but also affect the functional partitioning between these cards. We need to develop a reference application that must capture the following trends in access networks [19]:

- ATM is slowly vanishing and replaced by Ethernet. We must support Ethernet as well as ATM layer-2 protocols.
- Since Ethernet protocol will increasingly be used for the first mile, our DSLAM has to support Quality-of-Service (metering, scheduling, etc.) based on Internet Protocol. IP eventually will be the only layer-3 protocol and is used end-to-end.
- Although IPv4 currently dominates the network infrastructure IPv6 needs to be supported in the near future.
- Emerging applications, e.g. video broadcast, are implemented efficiently if a DSLAM supports multicast traffic.

2.2. Traffic Scenarios

Traffic Scenarios define the stimuli our reference implementation is working with. They are an integral part of the benchmark to allow comparing design alternatives under defined conditions. To avoid a customer's non-conforming traffic to affect other customers, a line card should handle the worst-case of small packets at peak rate. We have defined a set of line and trunk card configurations in terms of number of ports and supported bandwidth for representative subscriber line technologies: ADSL, VDSL(2+), and SHDSL. Our traffic mix has an average packet length of about 260 Byte versus 64 Byte for the worst-case (see [19]).

2.3. Parameterization

As a result of our analysis we can derive a set of configurations that are used for quantitative design space exploration. They represent deployment scenarios and affect the dimensioning of the DSLAM. We divide the various design and configuration axes for our reference implementation in function, system, and environment categories. Table 1 summarizes the configuration parameters and their value range.

Table 1. Setups for DSLAM implementations.

	Parameter	Value, Range
Function	DSL port protocol	ATM, Ethernet
	IP Version	IPv4, IPv6
	Uplink protocol	ATM, Ethernet
System	Number of line cards	1 – 32
	Ports per line card	1 – 72
	Function split (line-/trunk card)	centralized, distributed, decentralized
Environment	Packet length	QoS traffic partition, Traffic mix
	Traffic setup	QoS, Multicast, ICMP/errors
	Port throughput	ADSL/VDSL (asymmetric), SHDSL (symmetric)

3. Application Reference Model

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 has to be modular so that customizations can be achieved quickly. We model the DSLAM in Click [10], a domain-specific framework for describing network applications. Click models are modular, executable, implementation independent, and capture inherent parallelism in packet flows and dependencies among elements (Click components). Furthermore, Click's abstraction level and the existing extensible element library allow us to focus on application specifics during the implementation. By using Click, a functionally correct model of the application can be derived quickly. The subsequent performance optimization can focus on individual elements and the partitioning of elements onto processing cores.

3.1. General Concept

A DSLAM model consists of 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 DSLAM is working in. The DSLAM is partitioned into upstream and downstream flow directions, which are implemented independently since they do not interact directly at the data plane level.

3.2. System Functionality

The DSLAM system function can be described by a set of primary tasks. Each task may contain several Click elements. In [19] we show how eleven primary tasks can be used to implement an IP-DSLAM, which *distributes* the processing among line and trunk cards so that most processing is done near the ingress port of network traffic. Due to the modular nature of our application representation, centralized or decentralized organizations can quickly be modeled as well.

The corresponding Click implementation is shown in Figure 2. As an example we illustrate the line card for the upstream flow that embodies different protocol scenarios. It implements four ports that are configured for the four possible protocol combinations Ethernet/IPv4, ATM/IPv4, ATM/IPv6, and Ethernet/IPv6. To a large extent these steps are implemented per DSL port, i.e., per customer. The traffic is then aggregated into one set of QoS queues per line card.

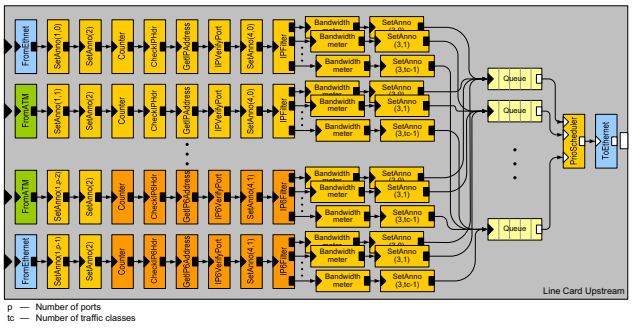

Figure 2. Upstream IPv4/IPv6 line card.

The pure Click description of an exemplary distributed DSLAM is compact and requires only 266 lines-of-code (without comments), representing 932 Click elements for four line cards, one trunk card, four ports per line card, three QoS classes per customer, and layered traffic sources and sinks. Click's modularity in addition enables us to quickly adapt the reference model to new requirements if needed.

4. Architecture-independent Profiling

The goal of this step is to determine architecture-independent characteristics of the reference application by static analysis and/or simulation, which helps pruning the design space and finding a good starting point in architecture space before implementation. Analysis and simulation focus on the following features:

- *Scalar data types*. In the DSLAM, most processing is done on header fields, 32-bit or less (e. g. TTL/hop limit [8 bits], IPv6 flow identifier [20 bit], DiffServ codepoint [8 bit]).
- *Data types* used for communication. Packet descriptors [32B] are exchanged between tasks that contain addresses, packet length, payload ID, flow ID, and so on.
- *Primitive operations* applied. The DSLAM uses frequent field matching operations that require bit masks and shifts in addition to primitive arithmetic and logic operations.
- *Frequent sequences of operations* (templates), e. g. move and logic operations combined with bit masks and shifts (see the "load word & xor" operation described later).
- *Frequencies* of execution for each component (e. g. Click element), and frequencies of communication between components ("traffic").
- *Typical sequences* that determine temporal and spatial execution and communication patterns (loops, potential critical paths, and so on).
- *Storage requirements*, e.g., queue lengths due to scheduling bottlenecks.

We derive these characteristics by annotating the Click specification of our reference application and executing the model. Several conclusions can be drawn from the results of the analysis to prune the design space. The affinity to certain hardware building blocks (general purpose cores, DSPs, ASICs, or FPGAs) has been formalized in [20, 3]. Design decisions on the communication architecture can be drawn from analyzing the traffic between components. As an 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 [15], 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.

As expected from packet processing, a DSLAM implementation will benefit from accelerating frequent field accesses by introducing bit-level arithmetic and logical operations. The communication patterns are regular due to dataflow dominant processing paths. The analysis also reveals many concurrent and independent traffic flows. A scalable communication architecture can therefore be derived (see Section 6).

5. Design Space Exploration

We start with determining an upper bound on the required computational resources. We use the reference to profile the combination of core and compiler for a set of 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 processors. We have developed a framework called CRACC [18] that generates embedded code from Click descriptions.

5.1. Click for Embedded Processors

Click [10] is written in C++. Since most embedded processors can only be programmed in C, we have developed a framework that allows us to generate C code from Click sources. After a functionally correct model of the application has been derived, which can usually be done quickly, our Click front-end is used to generate a netlist and the corresponding configurations for C elements. The C code can then be cross-compiled and profiled on the respective embedded platform. A subsequent performance optimization can focus on individual elements and on the partitioning onto several processing cores.

5.2. Profiling Embedded Cores

We have profiled our IP-DSLAM reference on a representative set of ten embedded cores using cycle-accurate simulators. Fig. 3 compares the performance of individual cores measured as packet throughput per second (Pkts/s) and their area normalized to 90nm. The figure reveals that among the modern general purpose 32bit cores (ARM, MIPS, and PowerPC) the MIPS32 has the best performance/area trade-off for the chosen reference. Among the smaller cores which trade-off speed against area (ARM7, N-core [12]) or deploy a much reduced instruction set (processing engine), N-core would be a reasonable choice. The N-core, a lean 32bit RISC

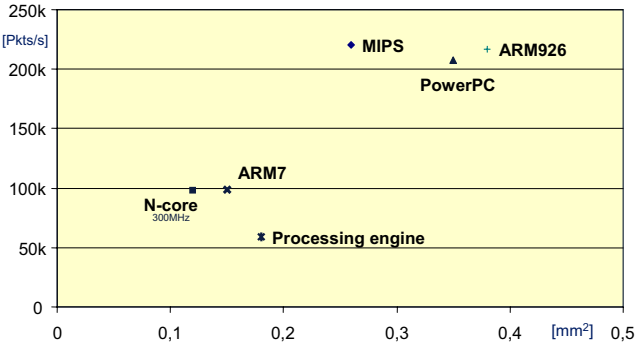

Figure 3. Performance of selected processor cores over area for the distributed line card, upstream scenario (any packet size).

architecture, has been 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 handcoding critical parts in assembly.

5.3. Required on-Chip Parallelism

Based on these profiling results and our traffic scenarios, we determine an upper bound on the required number of on-chip processing elements, buffer memory, and communication buses using an analytical approach [6] and thus reduce the design space systematically to a limited set of solutions. For the discussed line card, for instance, we either need four MIPS processors at 350 MHz or six N-cores at 400 MHz to support the worst case of processing minimum size packets. The on-chip communication architecture is determined by patterns of the application, e. g., packet descriptors that are passed between elements. Communication between ele-

ments on the same core is handled using simple function call semantics, whereas interprocessor communication requires message passing semantics. For our platform we use specialized network-on-chip communication interfaces that provide descriptor queues in hardware.

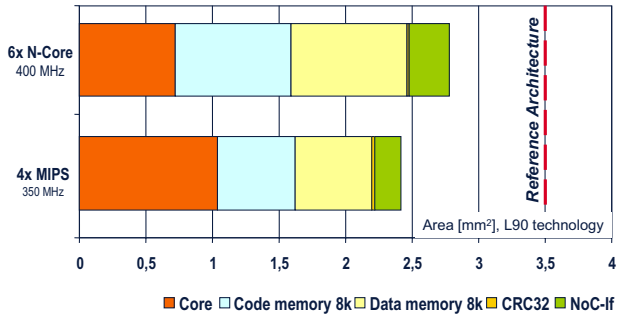

Figure 4. Area of selected on-chip MP clusters for the distributed line card.

Fig. 4 compares the area consumption of the two solutions with an existing and less performing system (aka *reference architecture*). For the comparison we only look at the area of the cores and their subsystems. The remaining SoC components' area (e.g. I/O, co-processors, peripherals) is kept constant and therefore not shown in the figure. Both solutions achieve a considerable area gain of 27% and 40%, respectively, over the reference architecture. Between the MIPS and N-core solutions the difference in area is 12%, therefore, almost negligible in the context of a complete SoC die.

6. Architecture Refinement

We continue the design space exploration and architecture modeling along the following axes:

- *Application-specific instruction set extensions* that still can be utilized by a state-of-the-art compiler.
- *Tightly/loosely coupled coprocessors* for standardized and complex tasks (such as header check or CRC, see below).
- *Memory hierarchy and data layout*, e. g., for large shared structures (on or off-chip, SRAM or DRAM, and so on).
- *Communication architecture* (such as hierarchical, globally shared, hybrid (shared bus and next neighbor communication, and networks-on-chip).
- *Customized I/O interfaces* that exploit domain specifics.

Since these axes are interrelated the exploration is an iterative process. Next, we describe in more detail our methods for evaluating instruction set extensions and coprocessors. Future applications are likely to demand processing power that exceeds the capabilities of small processor arrays. Therefore, we are currently investigating scalable communication architectures that support a large number of parallel processors.

6.1. Instruction Set Extensions

Our approach for deriving application-specific instruction set extensions [9] relies on an automatically generated compiler and simulator for the existing N-core. Based on data collected by the simulator, a frequently occurring pair of data-dependent instructions is selected and combined into a new super instruction. This refinement process is repeated until no further suitable instruction pairs can be found or until opcode space has been exhausted. As it is an iterative process, larger groups of data-dependent instructions can be combined into a single super instruction. For network applications, our analyses have shown that a combination of only two instructions LDW (load word) and XOR is already promising. The implementation of the resulting super instruction XORLDW has no impact on the critical path of the processor and increases the processor area only by 0.8%. With this minor modification, a speed-up of 1.1 can be achieved for certain kernels. Table 2 shows the resource requirements for N-core. The power consumption is reduced by nearly 10% for the used technology.

Table 2. Original and extended processor.

Core 90nm	*Total area*		*Total energy*	
	absolute (μm^2)	*relative increase*	*absolute nWs*	*relative decrease*
N-core	121280	-	2.37	-
with XORLDW	122314	0.845%	2.13	9.93%

6.2. Coprocessors

Whereas super instructions combine functionality at a fine-grained level, coprocessors form a more coarse-grained supplement. Our tool chain supports exploration of tightly and loosely coupled coprocessors. Tightly coupled coprocessors, which operate synchronously and have direct access to the registers of the core, can be employed either automatically by suitable tree patterns for the code selection phase, or the programmer has to address them explicitly through intrinsic functions.

If the simulation has revealed profitable functionality for a coprocessor, matching hardware needs to be implemented. In case of the DSLAM, simulation results unsurprisingly demand a coprocessor for CRC calculations and suggest a coprocessor for the IP header check as a second step. The hardware accelerator that has been implemented for IP header check needs 8 clock cycles to perform the complete functionality, corresponding to 228 million header checks per second if the accelerator is working at maximum clock frequency of 1.82 GHz (90 nm). Due to the parallel implementation of the CRC, it processes 32 bit input data in one clock cycle. The CRC unit performs CRC8 as well as CRC32 checks and achieves a speed-up of up to 630 compared to an implementation on N-core. Table 3 gives an overview over the area requirements for different frequencies.

Table 3. Area for accelerators at 90nm.

IP header check		*CRC*	
max. frequency (GHz)	*req. area* (mm^2)	*max. frequency* (GHz)	*req. area* (mm^2)
1.82	0.0146	1.41	0.0098
0.49	0.0078	0.775	0.0066

6.3. Communication Architecture

The analysis of communication patterns has shown that a single shared bus is sufficient to perform message passing between processors on a line card (four to six cores). A memory bus is kept separately. To avoid known scalability issues with multiplexer-based buses for a higher number of processing elements as, for instance, needed by a platform for trunk cards, we follow a hierarchical approach. Small clusters of processors are connected via on-chip buses while switch boxes connect these clusters on the higher level [2]. Their interconnection can form arbitrary topologies, such as meshes, tori, or butterfly networks. The communication architecture is parameterizable in respect to the number of processors per cluster, the number of clusters, and the bandwidth of the on-chip communication channels. Network processing applications reveal a high level of concurrency due to the high number of independent traffic flows. The organization in processing clusters therefore is a viable approach.

6.4. Virtual Prototype

Our prototype consists of a cycle-accurate simulation model of the N-core, parameterizable caches, and building blocks for the communication architecture. It is refined iteratively, i.e., for a more detailed analysis all building blocks of our system architecture as well as the proposed instruction set extensions and coprocessors are described in VHDL. The prototype serves as executable specification for the actual platform hardware and software implementation and as golden model for verification.

7. Platform Implementation and Deployment

A successful deployment of a programmable platform can only be achieved if the programmer is relieved from understanding the architecture in full detail. The programmer is faced with the problem of coordinating the execution of software portions on different processing elements manually that must be optimized individually on assembly level [11]. This burden frequently leads to suboptimal usage of hardware resources and is tedious. Our programming tools that are based on the described Click/C design flow [18] contain library elements and code generators that are responsible for interfacing hardware resources and synchronizing functional components that are mapped on different resources. We apply the idea of wrappers [21] to fit hardware resources together, which can affect hardware and software parts (drivers).

In our framework, the inter-processor communication of elements follows message passing semantics including a destination address. The address must allow the identification of both the target processor and the associated task graph on that processor. For that purpose special transmit and receive elements in combination with a FIFO are inferred after partitioning that annotate the packet with the required information. These elements furthermore convert push and pull chains as needed (see Fig. 5 for an example). Finally, these elements encapsulate the processor's hardware communication interface. We use specialized on-chip communication interfaces that provide ingress packet queues in hardware. In this way, we can quickly port our application to other platforms since hardware-dependent software parts are clearly separated from the actual application.

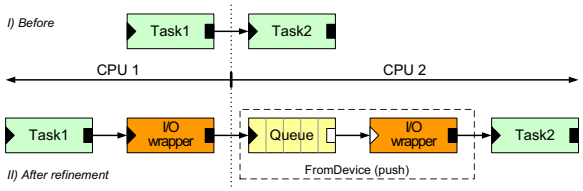

Figure 5. Push chain comm. wrappers.

8. Related Work

Design Frameworks. StepNP [13] is a design framework for network processors that uses Click for specifying the application. The application development environment [14] is based on C++ and provides SMP and message passing programming abstractions. We rely on common ANSI-C compilers in our library-based approach for efficiency reasons. C is also often the only programming abstraction above assembler provided by the embedded core manufacturer. Metropolis [1] is a general design framework where arbitrary models of computation can be expressed by using the meta modeling language. Its generality however makes the modeling and implementation effort more complex and less intuitive than by restricting the designer to one consistent representation for particular application domains. Our design flow is tailored to network processing. Artemis [16] is targeted at media processing (streamed) applications and based on Kahn process network representations. The main use of Artemis is modeling and performance evaluation of systems, whereas we put emphasis on efficient hardware and software implementations.

Domain analysis and reference application. There are various benchmarking efforts, see [5]. No system-level reference applications have been defined so far that specify both comparable and representative DSLAM usage scenarios. Existing benchmarks focus on kernels only, e.g. packet forwarding.

Software implementation. In [5], Shah et al. show how Click, augmented with abstracted architectural features, can be used as a programming model for the Intel IXP processor. Teja C [4] extends C for programming network processors and thus requires a proprietary compiler. We encapsulate

platform-specific aspects in library elements so that ANSI-C compilers can be used and maintain a Click front-end.

Platform architecture refinement. Commercial tools for customizing the micro-architecture of processing elements include CoWare's LISATek and Tensilica's Xtensa. Academic efforts based on ADLs include EXPRESSION [8] and Tipi [5]. They all can be reasonable tools for architecture refinement. The actual choice for one approach has to trade off ISA compatibility (reusing compilers and firmware), application-specific optimizations (e.g. irregular data paths), licence fees for commercial cores, licence fees for design frameworks, and flexibility of the platform. We have decided to rely on our own core and tool development for the cost-sensitive access network segment. The state-of-the-art in building and refining virtual prototypes using SystemC can be found in [7].

Partitioning and mapping. The partitioning of functionality onto hardware resources is a manual process in our framework. By using the boundaries of Click elements and architecture-independent profiling of the reference application, the number of reasonable choices can be reduced significantly so that this procedure is feasible. Automatic approaches exist, such as the heuristics in [17] and the ILP model in [5], that can potentially be combined with our methodology.

9. Concluding Remarks

The goal of our work is a design flow for the systematic and application-driven development of flexible packet processing platforms that are heterogeneous and may use many processing cores. In this paper, we have addressed five phases: 1) Application domain analysis, 2) reference application development, 3) architecture-independent profiling, 4) profiling-driven design space exploration, and 5) platform deployment.

We have shown how the comprehensive analysis of the application domain allowed us to identify primary tasks and realistic system environment that a future DSLAM has to support. Based on our findings we could restrict the design space to a set of reasonable alternatives.

We use Click to model an executable specification. The modularity of Click enables a natural encapsulation of local data and processing and thus reduces the number of mapping and partitioning alternatives considerably. Our path from a functional specification to an implementation on embedded cores using code generators permits rapid prototyping and performance feedback of design alternatives. The software implementation of a full DSLAM data plane can be done within a few days. Therefore, we promote the use of Click/C beyond the design space exploration phase as an efficient programming environment for the deployment phase. We finally customize our platform architecture by application-driven selection of instruction set extensions and coprocessors.

Acknowledgments. This work has been supported by the German government (BMBF), grant 01AK065A (PlaNetS).

References

[1] F. Balarin, Y. Watanabe, H. Hsieh, et al. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36(4), Apr. 2003.

[2] A. Brinkmann, J.-C. Niemann, I. Hehemann, D. Langen, M. Porrmann, and U. Rückert. On-chip interconnects for next generation system-on-chips. In *15th Annual IEEE Int. ASIC/SOC Conference*, pages 212–215, 25 - 28 Sept. 2002.

[3] L. Cai, A. Gerstlauer, et al. Retargetable profiling for rapid, early system level design space exploration. In *DAC*, 2004.

[4] K. Crozier. A C-based programming language for multiprocessor network SoC architectures. In *Network Processor Design: Issues and Practices*, volume 2. Morgan Kaufmann, Nov. 2003.

[5] M. Gries and K. Keutzer, editors. *Building ASIPs: The Mescal Methodology*. Springer, 2005.

[6] 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, Nov. 2003.

[7] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, May 2002.

[8] A. Halambi, P. Grun, et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *DATE*, 1999.

[9] U. Kastens, D. K. Le, A. Slowik, and M. Thies. Feedback driven instruction-set extension. In *LCTES*, June 2004.

[10] E. Kohler, R. Morris, B. Chen, et al. The Click modular router. *ACM Trans. on Computer Systems*, 18(3), Aug. 2000.

[11] C. Kulkarni et al. Programming challenges in network processor deployment. In *CASES*, Oct. 2003.

[12] D. Langen, J.-C. Niemann, M. Porrmann, et al. Implementation of a RISC processor core for SoC designs – FPGA prototype vs. ASIC implementation. In *IEEE Workshop on Heterogeneous reconfigurable Systems on Chip (SoC)*, 2002.

[13] P. Paulin, C. Pilkington, et al. StepNP: a system-level exploration platform for network processors. *IEEE Design & Test of Computers*, 19(6), Nov. 2002.

[14] P. Paulin, C. Pilkington, et al. Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In *CODES+ISSS*, 2004.

[15] H. Peixoto and M. Jacome. Algorithm and architecture-level design space exploration using hierarchical data flows. In *ASAP*, 1997.

[16] A. Pimentel, L. Hertzberger, P. Lieverse, et al. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11), Nov. 2001.

[17] R. Ramaswamy, N. Weng, and T. Wolf. Application analysis and resource mapping for heterogeneous network processor architectures. In *Network Processor Design: Issues and Practices*, volume 3. Morgan Kaufmann, 2005.

[18] C. Sauer, M. Gries, and S. Sonntag. Modular domain-specific implementation and exploration framework for embedded software platforms. In *Design Automation Conf. (DAC)*, 2005.

[19] C. Sauer, M. Gries, and S. Sonntag. Modular reference implementation of an IP-DSLAM. In *ISCC*, June 2005.

[20] D. Sciuto, F. Salice, L. Pomante, and W. Fornaciari. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *CODES*, 2002.

[21] 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.