Modular Domain-Specific Implementation and Exploration Framework for Embedded Software Platforms

Christian Sauer, Matthias Gries, Sören Sonntag
Infineon Technologies, Corporate Research, Munich, Germany
{Christian.Sauer|Matthias.Gries|Soeren.Sonntag}@infineon.com

ABSTRACT

This paper focuses on designing network processing software for embedded processors. Our design flow *CRACC* represents an efficient path to implementation based on a modular application description, while avoiding much of the overhead of existing component-based techniques. We illustrate results for a real-world application implementing a full IP-based DSL Access Multiplexer (IP-DSLAM) system. We quantify overhead and optimization potential incurred by our modular implementation. We also point out how CRACC can be deployed for HW-SW partitioning and design space exploration.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments. D.3.4 [**Programming Languages**]: Processors - *code generation.*

General Terms

Design, Languages, Performance.

Keywords

Software Development, Programmable Platforms, Design Space Exploration, DSLAM, Network Processing.

1. INTRODUCTION

Increasing time-to-market pressure is leading to heterogeneous programmable architectures as a main platform of customizable embedded systems for different application domains. Current best practice in developing embedded software under tight memory and run-time constraints is based on programming in assembler. Examples are applications for cell phones and the data plane in network processors. Clearly, this approach is in conflict with other design criteria, such as ease of maintainability and portability among architectures. 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 [12].

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 on the other hand. In this context we have to recognize that programming in C is the only abstraction above assembler that is established as a programming model for embedded processors [7]. In order to reach wide acceptance among embedded software engineers, we have to provide a programming 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. The overhead for granting a higher abstraction to stimulate reuse and to establish a disciplined development flow must not exceed the requirements of a plain implementation significantly to be economically justifiable. As a side effect, apart from optimizing the design process of firmware, a better abstraction enables the option to release the architecture as an open platform to the customer.

Flexible and programmable platforms are particularly popular in the domain of network processing, which is the focus of this paper. The results can be extended to other dataflow-oriented application domains as well. The modular application description framework Click is the basis for our design flow CRACC (Click Rapidly Adapted to C-Code) that enables the fast migration of network processing functionality to different embedded processing cores. We will show that real-world network processing applications can be implemented using CRACC with minor overhead with respect to code size, while providing a reasonable abstraction for rapid exploration of different implementation alternatives, e.g. in terms of different mappings and hardware-software partitioning trade-offs on a heterogeneous programmable platform. Contrary to CRACC, current best-practice requires the programmer to optimize resource allocation and scheduling across the whole application in assembly.

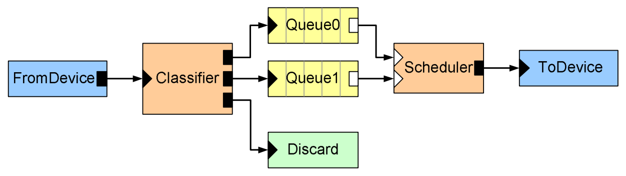
The paper is structured as follows. In the following section, we introduce the main concepts of Click and discuss related work. In Section 3 we describe our approach called CRACC, which is the main focus of this work. Section 4 gives the results of our case study where we implement a digital subscriber line access multiplexer (DSLAM) using CRACC, followed by a discussion in Section 5. We conclude in Section 6.

2. EXPLOITING MODULARITY

As a first step, we implement our functional IP-DSLAM scenarios in Click, a domain-specific framework for describing network applications [5]. We have chosen Click for several reasons: 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 following performance optimization can then focus on individual elements and the partitioning of elements onto processing cores. This systematic approach leads to improved design productivity and simplifies reuse.

2.1 Click Characteristics

In Click, applications are composed in a domain-specific language from elements written in C++ that can be linked by directed connections. The elements describe common computational network operations, whereas connections specify the flow of packets (i.e. data) between elements. Packets are the only data type that can be communicated. All application state is kept local within elements. Two patterns of packet communication are distinguished in Click: push and pull. *Push* communication is initiated by a source element and models the arrival of packets at the system. *Pull* communication is initiated by a sink and models space that becomes available in an outbound resource. Figure 1 shows a simple illustrative Click example using a common graphical syntax to represent the interaction of elements. In the example, packets are inserted by the *FromDevice* element into the system (push outputs [black]). The packets are then fed into a *Classifier*. This element forwards a packet to one of its output ports depending on the result of the internal processing, e.g. based on filtering header fields. Two outputs are connected to queues. In Click, queues are explicit elements that have push inputs and pull outputs (white). Thus, the *Scheduler* can pull packets out of the queue at its own rate removing them from the system.



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

Figure 1: Click Example.

2.2 Related Work

Related work to our approach can be found in three areas: I) work extending the Click software to exploit specific architectures, II) frameworks that use Click for application modeling and architecture design space exploration, and III) methods that use Click to map hardware description language content to particular processing hardware.

Click [5] was originally implemented on Linux using C++. Recent extensions to Click include SMP-Click [2] and NP-Click [11]. In [2], a multi-threaded Linux implementation for general-purpose PC hardware is described. In [11], 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. In Section 3, we will provide a way to map Click software onto embedded processor cores.

StepNP is a design framework for network processors [8] that uses Click for specifying the application. The corresponding application development environment [9] is based on C++ and provides common SMP and message passing programming abstractions. Contrary to that, we rely on common ANSI-C compilers in our library-based 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 [7]. Finally, Teja C [3] extends the C language for programming network processors and thus requires a proprietary compiler. In our approach, we encapsulate platform-specific aspects in library elements so that ANSI-C compilers can be used that are prevalent in the domain of embedded systems.

Click is also used to encapsulate hardware description language content. In the CLIFF work [6], Click elements are mapped to FPGAs while reusing standard back-end tools. CRACC's front-end can be adapted to generate netlists for CRACC and CLIFF, given a Click input. By incorporating CLIFF, we could provide a comprehensive framework for hardware-software partitioning and implementation of network applications based on standard deployment tools. In this way, profiling-based input to systematic mapping tools, e.g. based on ILP [1] or analytical models [13], can be generated seamlessly.

3. CRACC – Click for Embedded Processors

Click [5] 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++ [7]. Especially in case of 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 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 that Click offers:

- Domain-specific language for hierarchical composition, configuration, and interaction of elements.
- Run-time re-configurability of element connections.
- Modular programming style using object oriented features.
- Concurrency and scheduling of elements at run-time.
- Timers and timed rescheduling of elements.
- Communication interfaces between elements.

In this section, we first describe CRACC (Click Rapidly Adapted to C Code), a framework which adapts Click to ANSI-C elements that can be executed on embedded processor cores. Then, we discuss how CRACC is mapped onto embedded cores. We discuss concepts for heterogeneous multi-processors and combined hardware/software solutions at the end of this section.

3.1 Click Rapidly Adapted to C Code

The CRACC design flow (see Figure 2) starts with an implementation in Click. The application programmer can model the functionality on any host where 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 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.

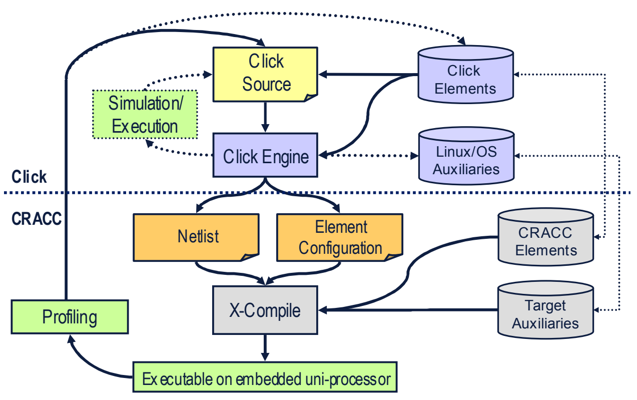


Figure 2: CRACC Framework.

3.1.1 Preserving Modularity

CRACC preserves Click's modularity by emulating object-oriented programming. Particularly objects and their construction using *new*, inheritance, and virtual function calls are supported using function pointers in *structs*, e.g. described in [4]. That means, a Click element is represented in C by a *struct* that contains pointers to element functions in addition to its element data. In that way, Click's push/pull function call semantics can be preserved. In the following we discuss these issues using our CRACC implementation.

- **An object in C** – Object methods require in contrast to C++ 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 *new*, 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 quite 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 replacing an element function with another one even at run-time.

3.1.2 Packets and Other Data Types

CRACC implements a *Packet* data type that allows the modification of its data. In Click *Packets* and *WriteablePackets* 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.

3.1.3 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.

3.1.4 Simulation Engine

A simulation engine is provided by CRACC to enable 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. That means, 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.

3.1.5 Preserving the Click Language

The Click language describes the composition of a packet processing application from elements and its configuration. Our CRACC implementation 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 2).

Since the CRACC configuration is done after processing of the Click configuration strings, only preprocessed valid data structures are used for configuring the CRACC elements.

3.2 Targeting CRACC to Embedded Cores

So far, we targeted CRACC 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 were using 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).

3.3 Platform Mapping, HW Encapsulation

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

3.3.1 *Packet I/O*

Packet I/O in CRACC uses the same concepts as Click, i.e. *From-* and *ToDevices* that encapsulate interface specifics. 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, our *From-* and *ToDevices* are mapped onto the physical I/O interfaces, e.g. Gigabit Ethernet and Utopia. The communication with neighboring elements then follows the on-chip communication scheme as described in Section 3.3.3. *From-* and *ToDevices* can be replaced by artificial packet sources/sinks if our simulation engine is used.

3.3.2 *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 using the API within CRACC implementations or indirectly by particular 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.

3.3.3 *Inter-Element Communication*

CRACC elements pass packet descriptors between elements. These descriptors contain extracted packet header and payload data, annotated flow information, 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 the packet descriptor. Elements on the same processor require an additional *element 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 are inferred after partitioning that annotate the packet with the required information. These elements furthermore convert push and pull chains as needed (see Fig. 3,4). In case of a pull chain conversion, shown in Figure 4, user-specified rate information is currently required for the transmitter. Finally, these elements encapsulate the processor's hardware communication interface similar to other *From-* and *ToDevices*. We anticipate using specialized on-chip communication interfaces for the processor subsystems that provide ingress packet queues in hardware. These queues will 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 SW and HW** – The interface of a cluster of hardware elements to the on-chip communication network must support the same message passing semantics as for inter-processor communication, e.g. ingress queues. Depending on the interface between hardware elements the HW side of transmitters and receivers need to convert the communication semantics accordingly. In case of CLIFF HW elements [6] and queue interfaces, for instance, a three way handshake needs to be generated using full and empty signals of the queues.

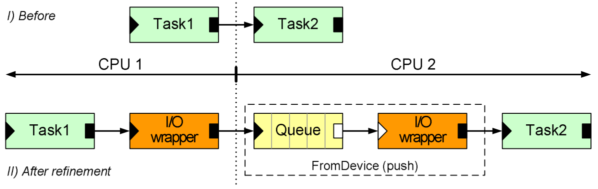


Figure 3: Synthesis of Push Chain Communication Wrappers.

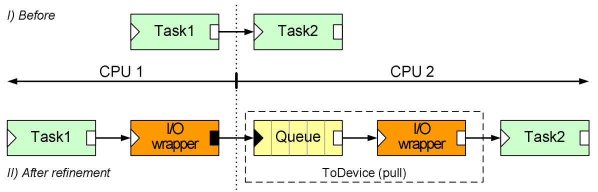


Figure 4: Synthesis of Pull Chain Communication Wrappers.

3.3.4 *Data Layout and Distinct Memories*

Communication with memories is encapsulated within CRACC elements. CRACC assumes a unified access to the memory hierarchy. Besides the local data memory all distributed/shared on- and off-chip memories are mapped into the processors memory space. Data objects are either *local* or *shared,* and are mapped explicitly by the programmer to a particular memory. In order to provide a hardware independent solution shared data should be encapsulated into objects that define its 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 [11] can be implemented. In case of read-only data the encapsulation can be omitted.

4. CASE STUDY

We apply the Click/CRACC framework to the domain of access networks and model a complete IP based DSL access multiplexer (DSLAM) [10] on the functional level in Click. We use Click for the functional verification of the model and the CRACC execution profiles on embedded processors for further design space exploration.

4.1 Setup

Figure 5 shows a generic upstream line card in Click that deploys Ethernet or ATM based links and operates on IPv4 packets. The processing steps experienced by each packet on its way through the card are: Layer-2 protocol termination (Ethernet or AAL5), IP Header verification, IP address validation, 5-tuple classification to determine the destination port and traffic class, policing and scheduling for QoS, and forwarding. Due to limited space, we report the specification for a line card only. In general, a DSLAM consists of several line cards – connecting customers to the DSLAM – and one trunk card that bundles customer traffic to the service provider. Usually up- and downstream packet flows require different processing steps.

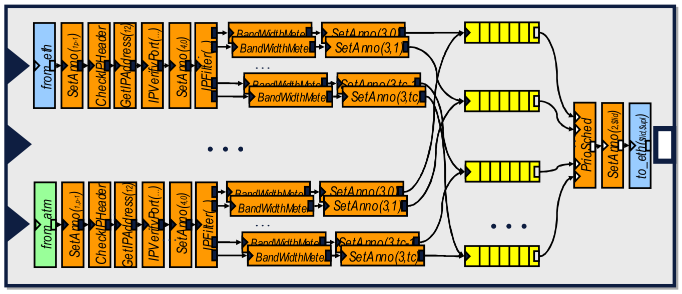


Figure 5: Upstream Line Card Functionality in Click.

4.2 CRACC Resource Requirements

As an indication for the required resources in both development and deployment we look at the modeling effort for the DSLAM and CRACC implementation, and the efficiency of the resulting code on selected embedded processors.

4.2.1 *Modeling Effort*

For the DSLAM domain, we had to extend Click's library with a few functional elements with approximately 2000 lines-of-code in total (ranging from 50 to 730 lines per element).

To date more than 50 elements have been implemented in CRACC that enable modeling of arbitrary IP-DSLAM systems. The implementation of the CRACC library and evaluation environment took about five man-months.

The pure Click description of the IP-DSLAM is rather small requiring only 266 lines-of-code (without comments), representing 932 connected Click elements. 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. Implementation and testing of the functional DSLAM model after completion of the CRACC library took a couple of days only. Composing the Click elements to form a DSLAM was finished within a day, whereas the actual configuration of filters, classifiers, and so on with corresponding test runs took several days.

4.2.2 *Code Efficiency*

The CRACC element implementations are much smaller than their Click counterparts due to the removed OO overhead and simplified packet abstractions. The compiled *IPVerifyPort* object, for instance, is 7025 bytes in Click's C++ framework, and only 1268 bytes in CRACC's C environment (g++, gcc, –O3, x86 platform). In average, CRACC's object code size is only 17% of the size of the Click objects. Table 1 gives an overview of code sizes of the CRACC framework on selected embedded processors. We distinguish between code that implements data plane functionality and code for performance evaluation (e.g., simulator and traffic sources). As one can see, the whole DSLAM can be implemented with less than 20 KByte of code memory that can be provided as on-chip memory.

Table 1: CRACC Library Code Size on Different Platforms.

Code Size [Bytes]	PowerPC	ARM	MIPS
Base	5268	3728	4484
Deployed elements	14340	10296	13024
Sum	19608	14024	17508
Environment/Traffic gen.	4140	2880	3552
Simulation engine	2992	3168	3040
Unused elements	8640	7656	7884
Sum	15772	13704	14476
Total	35380	27728	31984

4.3 DSLAM Performance Results

For the study, we evaluated different function partitions between line and trunk cards of the IP-DSLAM. In Figure 6, the profiling results are shown for a distributed DSLAM implementation. The data generated by cycle-accurate simulators of the corresponding cores is scaled down to the number of cycles required for the processing of a single packet. Assuming a core frequency of 300MHz, one core can process up to 210K packets-per-second (pps) in upstream direction or 650K pps in downstream direction on a line card. On a trunk card, the processing capability is 720K pps upstream and 150K pps downstream respectively. This would be sufficient to support a 64 port ADSL line card with one single processor assuming minimum-size (worst-case) packets and today's ADSL line rates (512kb/s upstream, 3Mb/s downstream).

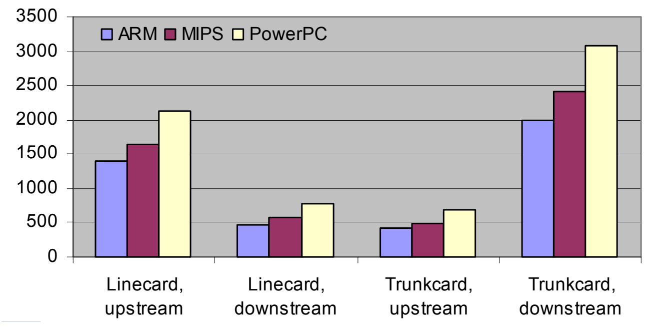


Figure 6: Profiling Results [clock cycles per packet].

5. DISCUSSION

We have introduced a comprehensive framework for evaluating and implementing network processing applications on heterogeneous embedded platforms. The advantages of using a modular specification can be summarized as follows:

A disciplined design flow for the exploration of partitioning and mapping decisions leads to a foreseeable, more deterministic path to an optimal design with respect to performance. A Click element is a natural encapsulation of local 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.

The run-time overhead for modularity is well justifiable by strongly reduced implementation time. There is potential for optimization after the mapping has been decided, as discussed in the next subsection.

The Click application can easily be specified using a graphical front-end. This supports documentation and simplifies reuse of components.

Platform-specific hardware is encapsulated in auxiliary modules or distinct CRACC elements. Only these few components must be ported to a new platform. CRACC thus provides a rapid prototyping path to different architectures.

Since Click is increasingly being used for specifying applications, we believe it is crucial and beneficial to support Click as input to implementation on embedded processors. Due to this focus, CRACC understands Click and is internally based on C.

5.1 Potential for Optimization

Applying modularity as provided by Click comes at the price of some overhead in run-time and code size. This overhead can be reduced significantly after partitioning and mapping steps have been completed.

- **Static push/pull element resolution** – CRACC supports push/pull agnostic elements. Their type is currently determined at run-time by the neighboring elements. We expect much of CRACC's function-call overhead to be removed if this resolution can be done during the initialization phase. We have performed a couple of experiments with this respect and were able to achieve an application speedup of up to 30%.
- **Push/pull concatenation** – The actual packet processing code in push or pull chains can be concatenated into one element and scheduled at once. On the one hand, this would remove most of the modularity runtime overhead. On the other hand, by combining several elements into one, the worst-case blocking penalty for a high-priority packet due to the processing of a low priority packet increases due to non-preemptive scheduling of elements. This clearly is a design trade-off to be made for any modular framework. A reasonable implementation is in between these corner-cases, i.e. small elements like counters should be combined with their surrounding elements.
- **Lean element implementation** – The current element module implements all local methods by function pointers for generality. Most of them, however, are not used. In addition, most of those used are never redefined. This would allow a leaner version that only implements the required functions. A second phase then could even distinguish between non-virtual and virtual methods, although this means different calling conventions.
- **Elements in assembly** – CRACC library elements are currently written in C for portability. Performance critical elements can of course also be implemented in assembly.

Finally, if simulator speed would be an issue, scheduling of timed elements can be made faster. Currently, timers are the only way to trigger push/pull chains in simulation mode. In case the CRACC simulation is used extensively a fast round-robin like way similar to task scheduling in Click would be an option.

6. CONCLUDING REMARKS

Click models the data plane of a network processing application. The configuration of the data path is the duty of the control plane. It is possible to expose Click element configurations to the control plane via NPF-compliant API calls, which remains to be implemented for completeness.

We have shown that a full DSLAM can be implemented on embedded processors using our framework CRACC "out-of-the-box" by requiring less than 20 KByte of code memory only. A processor running at 300 MHz on a DSLAM line card can support 64 of today's ADSL ports for worst-case traffic. Assuming that no elements must be added to the CRACC library, the software implementation of a full DSLAM data plane can be done within a few days.

The current CRACC library size reflects the functionality of an IP-DSLAM. The results achieved so far in terms of code size and efficiency are promising. We intend a public release of CRACC.

7. ACKNOWLEDGMENTS

This work has been supported by the German government (BMBF) grant 01AK065A (PlaNetS). The initial idea goes back to class projects of the MESCAL team at U.C. Berkeley. We especially thank S. Weber and M. Shilman for their input. In addition, we would like to thank W. Brunnbauer (TU Munich) for his help implementing our framework and M. Thies (U. Paderborn) for his helpful feedback on issues related to ANSI C.

8. REFERENCES

[1] A. Bender: MILP Based Task Mapping for Heterogeneous Multiprocessor Systems, *EURO-DAC*, 1996

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

[3] K. Crozier: A C-Based Programming Language for Multiprocessor Network SoC Architectures, *Network Processor Design*, vol. 2, Morgan Kaufmann, Nov. 2003

[4] A.I. Holub: *C + C++: Programming With Objects in C and C++*, McGraw-Hill, 1991

[5] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek: The Click Modular Router, *ACM Transactions on Computer Systems,* 18(3), Aug. 2000

[6] C. Kulkarni, G. Brebner, G. Schelle: Mapping a Domain Specific Language to a Platform FPGA, *DAC*, 2004

[7] P. Marwedel: Embedded Software: How to Make it Efficient? *Digital System Design, Euromicro*, Sept. 2002

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

[9] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, G. Nicolescu: Parallel Programming Models for a Multi-processor SoC Platform Applied to High-Speed Traffic Management, *CODES+ISSS*, 2004

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

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

[12] I. Sommerville: *Software Engineering*, International Computer Science Series, 7th edition, Addison Wesley, 2004

[13] L. Thiele, S. Chakraborty, M. Gries, S. Künzli: A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures, *Design Automation Conference (DAC)*, 2002