Towards Adaptive Networking for Embedded Devices based on Reconfigurable Hardware

Enno Lübbers, Marco Platzner
and Christian Plessl
University of Paderborn, Germany
Email: enno.luebbers@uni-paderborn.de

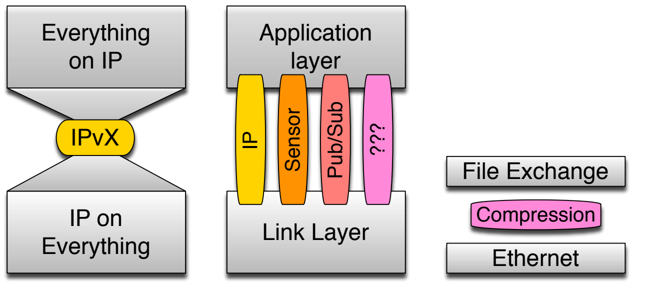
Ariane Keller and Bernhard Plattner
ETH Zurich, Switzerland
Email: ariane.keller@tik.ee.ethz.ch

***Abstract*—Research in communication networks has shown that the Internet architecture is not sufficient for modern communication areas such as the interconnection networks of super computing centers or sensor and mobile networks. Stringent requirements with respect to performance, cost, and power consumption paired with an increasing demand for flexibility ask for run-time optimization of the computing and communication functionalities. Specifically, the ability to adapt the communication protocol stack to the current needs of the application at run-time is a key element for optimally operating such a network of dedicated compute nodes.**

In this paper, we introduce the concept of a reconfigurable system-on-chip infrastructure for implementing adaptive protocol stacks. Our proposed architecture leverages research in the areas of adaptive networks and reconfigurable computing to provide a hardware/software platform that allows for runtime reconfiguration of existing network protocol stacks, deployment and removal of protocols, migration of packet processing tasks from software to hardware and vice versa, and efficient packet forwarding between different processing tasks.

I. Introduction

In recent years communication between computing nodes became more and more important. Initially, the communication infrastructure was built on commodity computers that provided just a small amount of computing performance. With the increase in data volume and complexity in the processing steps the implementation of networking tasks was shifted more and more to hardware implementations. This worked well, since the predominant network architecture, TCP/IP, is quite static. TCP/IP was designed for a fixed infrastructure without resource limitations, and without changing environment and mobility in mind. This is well reflected in the *Internet hourglass* depicted in Fig. 1(a). TCP/IP allows for flexibility in the link layer and in the application layer, but it requires the IP protocol to glue these layers together. The introduction of IPv6 has proven that it is difficult to change anything on the IP layer. While the IP architecture has proven to be very successful in the Internet, current network research focuses on softening this strict architecture in order to allow for more specific networking stacks and for the adaptation of those stacks at run-time. Such stacks are useful in several environments ranging from high-performance compute node interconnects over small sensor node networks to mobile networks. The requirements in these extreme environments are too diverse that a one-size-fits-all architecture as the TCP/IP architecture would fit. For example, small sensor network nodes that communicate only locally would waste a lot of energy when supporting a whole TCP/IP stack, or network nodes in a wearable computing scenario may profit from adapting their networking protocols depending on whether the user is at home or in an unknown, potentially hostile, environment. A network architecture that fits those diverse needs is depicted in Fig. 1(b).



(a) Internet architecture (b) Adaptable network architecture (c) Stack without IP protocol

Figure 1. Comparison of the Internet protocol stack and an adaptable network architecture.

However, with the introduction of evolvable protocol stacks, new demands are imposed on the hardware on which these architectures can be implemented. To provide sufficient performance and energy efficiency, it is desirable that not only the software, but also the hardware platform can be adapted at runtime. For this purpose, we use dynamically reconfigurable architectures as hardware implementation platform for our work.

The contribution of this paper is the presentation of an adaptive networking environment which employs a flexible networking stack mutable during run-time. The driving idea is to split networking protocols into smaller units that can communicate with each other. At run-time, we choose the set of units and their composition which is best suited for the given application and context. Moreover, our new flexible networking stack utilizes run-time reconfigurable hardware technology in order to optimize processing performance and power consumption by dynamic hardware/software partitioning. Our approach leverages and combines previous research in autonomous network architectures and run-time reconfigurable hardware, especially operating system support for reconfigurable logic.

The remainder of this paper is structured as follows: in section II we review the concept of adaptive networks and introduce ANA, a network architecture delivering the concepts that also appear in our proposed architecture. Section III gives an overview of ReconOS, an operating system tailored for embedded reconfigurable devices. Section IV shows how to combine ANA with ReconOS in order to arrive at a run-time reconfigurable adaptive networking architecture for embedded devices. Section V concludes the paper.

II. Adaptive Networks

A. Related Work in Adaptive Networks

The composition of protocol stacks dedicated for a specific use has been an ongoing research area for the past 15 years, dating back to the x-kernel [1] which defines an explicit architecture for constructing and composing network protocols. In the following years several projects were suggested and with the Click modular router [2] one project achieved widespread use. In the Click project the networking functionality is divided into small elements where each performs a simple computation such as decrementing a TTL or queuing a packet.

However, these projects are not targeted to a future network environment in which the network is more heterogeneous and should adaptively optimize its service. More recently, many projects have been launched (e.g., within the FIND initiative [3] in the US and the FIRE initiative [4] in Europe) that target different aspects of future networks [5]. There is a broad consensus in the community that in the future, network functionality should be organized in *flexibly composable small building blocks*. The ANA [6] project is fully in line with this approach and provides a networking architecture in which the available services can be adapted continuously. ANA even allows for running completely distinct protocol stacks and is thus well suited for a heterogeneous and mutable computing environment. We present an overview over ANA in section II-B and use it in section IV as a conceptual basis for our architecture.

B. The Autonomic Network Architecture Project

The concepts developed in the Autonomic Network Architecture (ANA) project [7] form the foundation of our adaptive networking architecture. In order to provide high flexibility, ANA divides networking functionality into *functional blocks (FB)* that may be combined as required by any given situation. The functionality of a single FB can range from a full monolithic network stack down to a small entity like computing a checksum.

The dynamic combination of FB rests upon the concept of indirection and is realized with so called *information dispatch points (IDPs)*. IDPs are inspired by the work on network pointers [8] and are also somewhat similar to file descriptors and sockets in Unix systems. Instead of a tight binding between networking functions, where one function directly calls another function, data is first sent to an IDP from where it is forwarded to the next networking function. This layer of indirection allows us to dynamically adapt the networking stack by changing the bindings of IDPs to FBs.

ANA also introduces the concept of *compartments*. A compartment is a set of FBs and IDPs with some commonly agreed set of communication principles, protocols and policies. Typical network compartments are an Ethernet segment, the public IPv4 Internet, a private IPv4 subnet, the DNS, or peer-to-peer systems like Skype.

In order to allow the construction of arbitrary network stacks, ANA provides a compartment API (application programmer interface) which describes all communication between different compartments and bases on a publish/resolve architecture. The two key terms are the *service* and the *context*. A service describes a specific functionality, and a context describes where this functionality is available. With these two terms ANA offers the following six functions (noted in a simplified C-style syntax).

- $\texttt{IDP}_s$ `publish(`$\texttt{IDP}_c$`, CONTEXT, SERVICE)`
- `int unpublish(`$\texttt{IDP}_c$`, `$\texttt{IDP}_s$`)`
- $\texttt{IDP}_r$ `resolve(`$\texttt{IDP}_c$`, CONTEXT, SERVICE)`
- `int release(`$\texttt{IDP}_c$`, `$\texttt{IDP}_r$`)`
- `void* lookup(`$\texttt{IDP}_c$`, CONTEXT, SERVICE)`
- `int send(`$\texttt{IDP}_r$`, DATA)`

An FB advertises its functionality by publishing it in a certain context. A published service can be resolved by other FBs, and the resulting IDP can be used to send data to the resolved service. Besides service resolution, one can also lookup a service to obtain reachability information. This is useful for services such as the DNS (Domain Name System).

If a resolved service is no longer needed, the corresponding IDP can be released. Finally, if an FB stops offering a service it will unpublish it.

In the function prototypes above, the IDP_c identifies the compartment in which the API call will be executed. IDP_s is created by the publish primitive and identifies a published service. IDP_r is created by the resolve primitive and identifies a communication channel that can be used to send data.

In addition to network compartments, ANA has a special compartment called the *node compartment*. ANA considers each networking node to be itself a network composed by the functional blocks running on the host. The node compartment thus encompasses all FBs and IDPs within a node. In the node compartment, the API is implemented by a node-local controller, called Minmex. When an FB is loaded onto a node, the FB registers itself with the Minmex and receives in return the IDP of the node compartment. This IDP will be used for further communication from the FB to the node compartment. Additionally, the Minmex keeps track of all the loaded FBs and of the mapping from IDPs to FBs.

To illustrate the use of this API, we present a simple communication setup that will end up with the protocol stack depicted in Fig. 1(c).

Since each node is internally organized as a compartment the communications *inside* a node are also setup via the compartment API. For example, a functional block implementing the Ethernet protocol can publish itself inside the node compartment with the following primitive:

```
e ← publish(n, ".", "ETHERNET")
```

The first parameter is the IDP n which is bound to the node compartment and provided to each FB when it registers with the Minmex. The second parameter is the context. In our example, the context is set to ".", a special context that restricts all operations to the local node. The publish function creates and returns the IDP e (randomly generated) which is now bound to the Ethernet FB. Subsequently, a second FB which wants to use the Ethernet FB will be able to resolve its IDP with the following request:

```
e ← resolve(n, ".", "ETHERNET")
```

The calling FB (e.g., a simple "file exchange application" called DemoEx) can now use the IDP e to publish its service in the Ethernet compartment.

```
i ← publish(e, "*", "DemoEx+contact")
```

In this example the application publishes "DemoEx+contact" in the Ethernet compartment. The context "*" specifies the largest possible scope as understood by the compartment. With regard to the DemoEx application the previous function call announces that the application is willing to share a file identified by "contact".

With the resolve primitive, DemoEx can search for other "contact" files and establish a connection to a remote host offering this file:

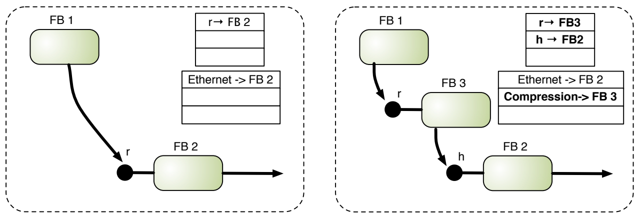
```
r ← resolve(e, "*", "DemoEx+contact")
```

This primitive causes the Ethernet compartment to search for a node that offers "DemoEx+contact" and to build an information channel to it. This information channel is identified by IDP r and data can be sent to it with the following primitive:

```
send(r, "get file")
```

This step completes the protocol stack setup for this example. The current packet flow and the required management tables in the Minmex are depicted in Fig. 2(a).

However, since the environment may change during a communication, this stack can be updated while a file is transfered. In the following part of this example, we assume that a monitoring module has determined that the utilization of the transmission medium is quite high and that the load should be reduced to avoid congestion. To reduce the number of bits that will be transmitted, a compression module can be inserted into the network stack. Within the ANA architecture this is done by loading a Compression functional block, e.g., a Huffman Coding FB [9], and re-mapping the IDP bound to the Ethernet block to this Compression block. Due to this re-mapping, all packets are forwarded to the Compression block, which will be able to forward the data to the Ethernet block. This re-mapping is shown in Fig. 2(b). Note, this reconfiguration of the protocol stack can be done without any explicit support of any of the functional blocks.



(a) FB 1 (DemoEx) is sending to IDP r that is mapped to FB 2 (Ethernet).

(b) FB 1 is still sending to IDP r, but now it is mapped to FB 3 (Compression).

Figure 2. The protocol stack can be easily adapted by changing the mapping of IDPs to functional blocks.

Evidently, the dynamic change in the communication protocol during runtime has to be synchronized between sender and receiver. For this purpose, ANA also uses the same publish/resolve mechanism. A detailed discussion of protocol changes is out of the scope of this paper. Many other aspects were also only touched, e.g., the interpretation of the keywords used to setup the protocol stack and the configuration of FBs that are included in the protocol stack.

Obviously, the ANA architecture allows for much more complex protocol stack setups than this simple two-layer example.

III. Programming Reconfigurable Hardware for Adaptive Networking Functions

Our target system is a *reconfigurable Systems on Chip* (rSoC) – an embedded device equipped with an FPGA that contains an embedded CPU. We have shown in our previous work [10] in the area of wearable computing, that reconfigurable hardware is a highly suitable technology to address the need for energy efficient and high performance processing. In many cases, reconfigurable hardware is able to outperform general-purpose CPUs by many orders of magnitude in performance and energy efficiency and can achieve almost ASIC-like performance, while still being fully programmable. Moreover, to achieve highest flexibility, we use FPGA families that allow us to change the configuration of the logic cells at runtime. Partial run-time reconfiguration enables the processing of data in one part of the device, while another part is being reprogrammed. We will need this feature to dynamically include new functional blocks in the hardware.

Admittedly, FPGAs are somewhat difficult to program as current implementation tool flows still require substantial knowledge of digital logic design. Moreover, in rSoCs hardware accelerators are traditionally implemented as slave co-processors to a single microprocessor. They are typically connected to a central memory bus together with other peripherals like memory and I/O controllers. In order to program such a platform, the developer needs specific knowledge of all employed accelerators, which hinders design productivity, portability, and scalability – factors of major importance when considering the rising complexity of reconfigurable hardware devices. Furthermore, the highly specific interfaces and protocols of the accelerator unnecessarily complicate the use of partial reconfiguration to adapt the system's hardware/software partitioning during run-time.

In order to provide a practical abstraction for programming reconfigurable CPU/FPGA systems that hides the hardware/software boundary, we have extended the multithreaded programming model – already in widespread use within software-based systems – towards reconfigurable hardware. This approach models hardware modules as hardware threads on the same level with software threads. Hardware threads interact with other threads using the same programming model primitives as their software counterparts. In our previous research on multithreaded reconfigurable hardware, we have augmented existing operating system kernels, such as Linux, with hardware and software extensions for integrating hardware and software threads into a single execution environment called ReconOS [11] [12].

ReconOS applications are thus typically crafted from the following operating system objects:

Threads are the basic units of execution which make up an application. It is up to the developer to partition an application into threads, which then communicate and synchronize using other operating system objects.

Semaphores and mutexes provide means for high-level *synchronization*; they can be used to sequentialize execution of threads, to protect critical code regions, or to manage exclusive access to shared resources.

Shared memory, message queues and mailboxes are used for inter-thread *communication*. Generally, access to shared memory must be protected by synchronization primitives, as is necessary for any shared resource. Message queues and mailboxes occupy a special niche among the operating system objects – they provide both communication and synchronization at the same time.

The fact that all inter-thread activity is carried out using only these objects provides complete transparency within these interactions; a thread does not need to know whether its communication or synchronization partners are located in hardware or software – which, in turn greatly facilitates design space exploration with respect to the hardware/software partitioning. Furthermore, the decomposition of an application into threads provides a suitable set of modules with a single, well-defined interface which is highly amenable to partial reconfiguration. Also, as long as the interfaces to the respective operating system objects are supported, the interoperability and portability of threads can be easily maintained when moving to a different target platform.

A. ReconOS Hardware Architecture

Being an extension to conventional software operating systems, ReconOS relies heavily on the existing host OS kernel for the implementation of its operating system objects. The main part of any ReconOS system is consequently a central microprocessor executing the host operating system kernel. Current implementations of ReconOS use a two-bus architecture for control and data communication between hardware threads and the OS kernel, although the concept is not limited to using a bus topology. Hardware threads are located within specified regions of the reconfigurable fabric, called slots, which are connected to the communication infrastructure through a dedicated hardware module, the OS interface (see Section III-C). Individual hardware threads can be exchanged during run-time by means of partial reconfiguration of the FPGA, which provides a means of dynamically adapting the system hardware/software partitioning to changing requirements.

The area of the individual slots is defined upon system design and should reflect the area requirements of the anticipated hardware threads. Upon the rescheduling of hardware threads area requirements should be considered in order to avoid hardware fragmentation.

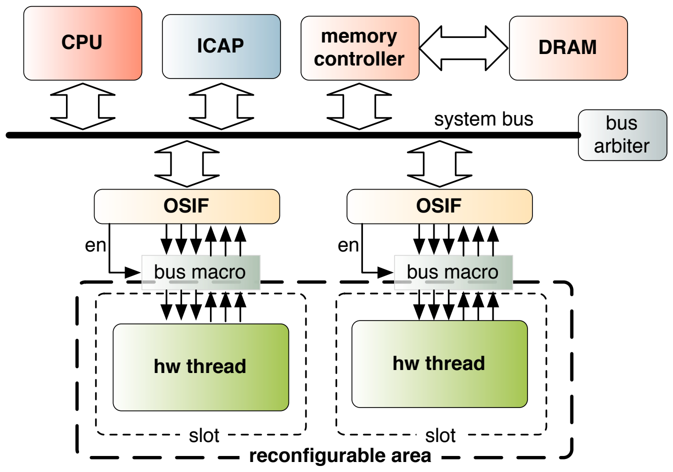


Figure 3. ReconOS system architecture.

The architecture of an example ReconOS system with two slots is depicted in Figure 3. Besides the main CPU and the operating system interfaces, the main bus also connects a DRAM controller and the FPGAs internal configuration access port (ICAP), which is used for dynamic partial reconfiguration.

B. Hardware Threads

In the programming model presented by ReconOS, hardware threads are written in VHDL and usually comprise two parts: a controlling finite state machine (FSM), which handles all operating system interactions, and custom user logic, which performs the actual computations of a thread. Both parts are completely user-defined and communicate using thread-specific handshake signals. The FSM is connected to a dedicated hardware module, the OSIF, and sequentializes all accesses to operating system functions from within the thread. To perform OS calls, the thread designer embeds functions from a VHDL library provided by ReconOS into the FSM. Because of the special structure of the FSM, which is required by ReconOS, its state transitions can be suspended by the operating system, effectively providing a hardware thread with the means to perform blocking operating system calls.

C. *Operating System Interface*

The operating system interface (OSIF) is a dedicated module connecting a hardware thread located inside a reconfigurable slot to the system's communication infrastructure. The OSIF offers a wide set of operating system services to its hardware thread, some of which it can perform directly in hardware (such as shared memory accesses), while others must be relayed to the OS kernel on the main CPU. There are a number of specialized communication extensions available to the OSIF, such as side-channel communication networks or point-to-point FIFOs, which increase the available communication bandwidth between hardware threads at the cost of flexibility. These communication extensions are transparently mapped onto the ReconOS programming model primitives already available to hardware threads, allowing them to transparently take advantage of the increased bandwidth.

D. *Delegate Threads*

In order to provide the host OS services to its hardware threads, ReconOS associates a dedicated software thread, the delegate, with every hardware thread instance. Whenever a hardware thread requests an operating system call that is to be handled in software, the OSIF forwards this call to the appropriate delegate thread by means of an interrupt. The delegate thread then retrieves the call parameters and performs the actual OS API call on behalf of the hardware thread. Any return values are transparently passed back to the initially calling hardware thread. Thus, all hardware threads appear to the host OS kernel as regular software threads, disguised by their delegate, which provides excellent flexibility, transparency and portability of the programming model across different target platforms and host operating systems.

IV. An Evolvable Network Architecture for Embedded Devices

In this section we introduce our evolvable network architecture for embedded devices. It relies on the ReconOS/Linux operating system to implement adaptive networking concepts from the ANA project on a modern reconfigurable hardware/software platform.

A. *Network Modules*

Our proposed architecture takes up on ANA's modularization of networking functionality into functional blocks (FB) and maps each FB to a collection of software and hardware threads running under ReconOS. Fig. 4 displays the organization of an FB consisting of one software control thread, one software data processing thread and an arbitrary number of hardware data processing threads. The software control thread is the only required thread and responsible for setting up connections between FBs. The control thread thus implements the compartment API with the primitives publish, unpublish, resolve, release and lookup. Those functions are not time critical operations, therefore it is no performance bottleneck if they are implemented in software. Data processing is implemented by another software thread or, depending on the performance requirements and suitability, by one or more hardware threads.

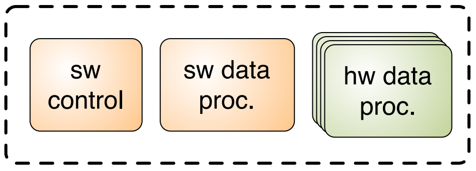


Figure 4. Example of a network module.

B. *System-on-Chip Architecture*

Figure 5 sketches the architectural layout of a compute node. Based on the ReconOS multithreaded programming model and architecture, a compute node employs a CPU core, a number of reconfigurable hardware cores, and memory and I/O blocks. To allow for an efficient exchange of network packets, the architecture foresees two on-chip interconnects, a shared system bus which all processing cores can access and a specialized multibus for interconnecting the hardware cores.

The shared system bus enables communication between software cores and between software and hardware cores, respectively. Table I summarizes the performance of the different communication primitives provided by ReconOS. From there it is obvious that the right communication mechanism has to be chosen with great care. Related software prototypes for adaptive networking architectures such as the current ANA implementation involve copying network packets as messages from one FB to another. Although this has the advantage that the two FBs cannot corrupt each others address space, it is rather inefficient in ReconOS as can be seen in Table I. Hence, for communication between software threads we foresee a shared memory approach which allows for an order of magnitude higher communication bandwidth.

The interface between software and hardware FBs is implemented through a combination of ReconOS message boxes and burst read and write operations. Messages are written from the hardware thread's local RAM to the system memory with the ReconOS burst interface. However, the software thread is notified of new messages by sending a notification through a message queue. To allow for an optimal performance, each hardware thread has not only one but two local RAMs, one storing the packets traveling upstream and the other storing the packets travelling downstream. With this arrangement the two flows can be processed independently until they are sent to a software thread, at which point they need to be synchronized, since only a single system memory exists.

Communication between hardware threads is more involved since all hardware cores can execute in parallel. The shared memory bus cannot provide the throughput required for multiple parallel transactions. For example, in order to support 1 Gbit/s Ethernet for simultaneous data read and write we require a bandwidth of 2 Gbit/s. Although the shared system bus uses 64 bit words and operates at 100 MHz, delivering a theoretical bandwidth of 6.4 Gbit/s, the actual maximum transfer rate between hardware and memory as shown in Table I only reaches close to 1.6 Gbit/s, which is not enough to allocate one single 1 Gbit/s transmission between software and hardware, even without additional data forwarding between hardware threads.

Our architecture allows the processing of packets in a pipelined manner where each pipeline stage consists of a hardware thread. Unlike conventional pipelining, our packet processing pipeline is not linear but may have branches and mergers. As underlying interconnect, we employ a multibus as shown in Fig. 5. Each hardware thread is assigned a bus from which it receives data via an FIFO-like interface. A hardware thread can further send data to the buses assigned to other threads. Consequently, sending requires an arbitration mechanism. For arbitration, round robin or thread priority based schemes can be used. Alternatively, more complex arbitration schemes with access control based on the priority of individual packets are conceivable.

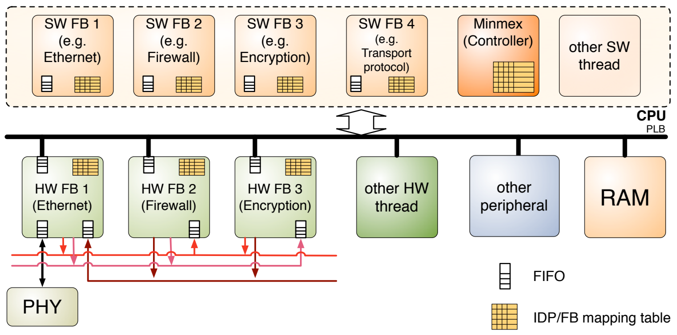


Figure 5. Networking functionality can either be executed in HW or in SW.

Table I
PERFORMANCE OF RECONOS COMMUNICATION PRIMITIVES.

Primitive	Throughput $[MB/s]$
memory $\rightarrow$ hardware (burst read)	168
hardware $\rightarrow$ memory (burst write)	193
software $\leftrightarrow$ software (memcopy)	13
hardware $\leftrightarrow$ hardware (mbox read and write)	127
hardware $\leftrightarrow$ software (mbox read and write)	< 0.1
hardware $\leftrightarrow$ software (mqueue read and write)	16

Table II
AREA REQUIREMENTS OF TYPICAL HARDWARE THREADS.

module	slices	flip-flops	LUTs
Ethernet	1947 (4.6 %)	2335 (2.8 %)	3426 (4.0 %)
Blowfish	12090 (28.7 %)	17095 (20.3 %)	19138 (22.7 %)
RC5	13368 (31.7 %)	13655 (16.2 %)	24564 (29.1 %)

The percentages given in parentheses refer to the relative logic utilization of a Virtex-4FX100 target platform.

C. Thread Switching and Dynamic Reconfiguration

The dynamic assembly of networking modules is supported by modeling them as threads in ReconOS. Adding or exchanging software threads is straight-forward, and exchanging hardware threads or exchanging software threads with hardware threads and vice versa is enabled by the partial reconfiguration capability of ReconOS.

Table III
RECONFIGURATION OVERHEAD FOR DIFFERENT SLOT SIZES.

slot size [slices]	max. bitstream size [kBytes]	overhead [ms]
1024	94.8	0.2
4096	379.3	1.0
8192	758.5	1.9
12288	1137.8	2.9
16384	1517	3.9

All estimations are based on slots with multiples of 64 slices in height on a Virtex-4FX100 target platform.

The physical size of the reconfigurable areas on the FPGA (the slots) is determined at design time, taking into account the actual sizes of the possibly required hardware threads. As an example, Table II lists the area requirements of an Ethernet core and two example cores implementing cryptographic functions. Since most FBs are less complex than cryptographic functions, the average FB size will be closer by the size of the Ethernet FB. The resource requirements are also given relative to a medium-sized FPGA device. The results show that, depending on the actual sizes of hardware threads and the selected FPGA, implementations with 10-20 hardware threads are realistic on today's technology. However, it is impossible to provide all functionality that will be required during the runtime of the system upon startup, since networking requirements are highly dynamic and depend on user behavior. There are two aspects that demand a reconfiguration of the system at runtime.

1) Reconfiguration upon changes in protocol stack.
As discussed in Section II the protocol stack can be adapted as required. There are several scenarios with respect to the frequency of adapting the protocol stack. In a statically assembled system, all required software and hardware modules are compiled together and loaded at startup. In a more interesting scenario, the single node is able to serve several applications and adapts the protocol stack whenever a new application with new requirements is started. In a third scenario networking functionalities such as encryption or compression can be added to existing network traffic flows at runtime. As all three scenarios happen relatively infrequently, the thread exchange overhead of the system does not have a significant impact on its functionality.
2) Reconfiguration upon changes in network traffic mix.
Depending on the fraction of traffic that needs a certain protocol functionality the number of hardware threads that are providing this functionality has to be adapted. If a particular functionality is used by many packets this functionality could be implemented as more than one hardware thread and a functionality that is only used rarely may be implemented completely in sw. An optimal hardware/software partitioning depends on the current traffic mix, the processing time of a packet in a given FB and the system overhead. Therefore a monitoring block collects information such as utilization of each FB in hardware, CPU usage of FBs in SW,

available battery power, peak packet rate of packets from a given protocol, running applications, etc.

Based on our estimations of the reconfiguration overhead for differently sized slots shown in Table III, it appears feasible to re-evaluate current environmental conditions and network traffic composition several times per second to determine a more optimal hardware/software partitioning of the functional blocks that make up the current protocol graph.

While the individual threads can be exchanged at any time, the communication infrastructure will not be dynamically reconfigured.

In the software implementation the mapping of IDPs to functional blocks is done centrally in the Minmex. Such a central Minmex is, however, impractical for the proposed architecture since we provide parallel execution of FBs. A central Minmex would form a performance bottleneck, no matter whether it is implemented in software or hardware. For this reason, each hardware thread holds a partial replication of the Minmex IDP-to-FB mapping table. The updating of these tables is initiated by the Minmex and forwarded to the hardware threads by the software control thread of a network module.

V. Conclusion

We have shown that technologies developed in the research fields of reconfigurable computing and adaptive networks are a perfect match to address the primary architectural challenges in a compute node architecture that can adapt its compute and network performance to the varying requirements of the users and applications running in the network while being power efficient at the same time.

We have introduced a new hardware and software architecture as a platform for energy efficient adaptive networking. In this architecture, we leverage the ANA adaptive network architecture model and the ReconOS operating system for reconfigurable computers. We have extended the ANA architecture with the functionality to implement network services also in hardware. This allows us to maintain the full advantage of adaptive networking systems, while providing the necessary performance by migrating performance critical or delay sensitive network services to hardware, while uncritical functionality can still be implemented in software. Whenever the network's traffic mix (e. g., the fraction of packets that requires a specific processing) or the available protocols change, the choice which networking functions are off-loaded to hardware and which remain in software has to be reevaluated. Since we implement all networking modules that shall be considered for hardware execution as a ReconOS hardware thread, the ReconOS scheduler allows for seamlessly migrating this functionality between hardware and software.

For evaluating the proposed architecture, we have developed a prototype based on a Xilinx Virtex-II-Pro FPGA evaluation board. This prototype has helped us to get first insights into the practical realization of the architecture. We have ported Linux and the ANA protocol stack to the development platform. As a proof-of-concept for a hardware networking module, we have implemented a functional block that processes Ethernet packets.

Ongoing work focuses on two main aspects. First, we are working on the implementation of encryption modules and on integrating them into the ANA framework for studying the interaction of software and hardware networking modules with more complex applications targeted at larger FPGA devices (e. g., Virtex-4FX). Second, building on top of the existing implementation, we will further complete and refine our architecture by developing algorithms for (a) determining the interconnections of the networking modules and for (b) choosing the hardware/software partitioning. Both algorithms need information about their node and the environment in which they are running. While determining the module interconnection mainly requires information about running applications and the connectivity to the peer nodes, choosing a proper hardware/software partitioning can be based on node-local information such as power consumption, available hardware slots or the frequency of packets that need processing by a given module. To enable these online algorithms, we will implement dedicated monitoring modules which gather relevant data at run-time.

Acknowledgment

This work was supported partly by the ANA (Autonomic Network Architecture) project [6] which is funded by the European Union Information Society Technologies Framework Programme 6, and partly by the German Research Foundation under project number PL 471/2-2.

References

[1] N. C. Hutchinson and L. L. Peterson, "The X-Kernel: An architecture for implementing network protocols," *IEEE Trans. Softw. Eng.*, vol. 17, no. 1, pp. 64–76, 1991.

[2] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[3] "FIND – Future Internet Design (FIND) - US National Science Foundation," at http://www.nets-find.net/, (May 09).

[4] "Future Internet Research and Experimentation (FIRE) initiative," http://cordis.europa.eu/fp7/ict/fire, (May 09).

[5] Subharthi Paul, Jianli Pan and Raj Jain, "Architectures for the Future Networks and the Next Generation Internet: A Survey," 2009, http://www.cse.wustl.edu/~jain/papers/i3survey.htm(Oct09),.

[6] "Autonomic Network Architecture - EU Project (2006-2009)," http://www.ana-project.org(Oct09),.

[7] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May, "The autonomic network architecture (ANA)," *Selected Areas in Communications, IEEE Journal on*, vol. 28, no. 1, pp. 4 –14, Jan. 2010.

[8] C. Tschudin and R. Gold, "Network pointers," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 23–28, 2003.

[9] D. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the I.R.E.*, 1952, p. 1098–1102.

[10] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, L. Thiele, and G. Tröster, "The case for reconfigurable hardware in wearable computing," *Personal and Ubiquitous Computing*, vol. 7, no. 5, pp. 299–308, Oct. 2003.

[11] E. Lübbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems Special Issue (CAPA)*, 2009, to appear.

[12] ——, "ReconOS: An RTOS supporting hard- and software threads," *IEEE Int. Conf. on Field Programmable Logic and Applications*, 2007.