A Distributed Object System Approach for Dynamic Reconfiguration

Ronald Hecht, Stephan Kubisch, Harald Michelsen, Elmar Zeeb, Dirk Timmermann
University of Rostock
Institute of Applied Microelectronics and Computer Engineering
Richard-Wagner Str. 31, 18119 Rostock-Warnemünde, Germany
{ronald.hecht, dirk.timmermann}@uni-rostock.de

Abstract

Managing reconfigurable hardware resources at runtime is expected to be a new task for future operating systems. But due to the mixture of parallel and sequential parts of dynamically reconfigurable applications, it is not entirely clear so far, how to use and to program such systems. A new interpretation of dynamically reconfigurable applications is presented. It will be shown, that the parallel computing concept of distributed object systems may be adapted for dynamically reconfigurable architectures. This approach answers many open questions concerning communication, interruption, and relocation of reconfigurable modules. It is explored by means of an extended Linux operating system in conjunction with a SystemC model of a dynamically reconfigurable FPGA.

1. Introduction

Operating systems supporting dynamic reconfiguration of hardware are heavily discussed since the appearance of the first dynamically reconfigurable devices, the Xilinx XC6200 series. Many OS prototypes have been presented in the past, ranging from specialized hardware operating [6] systems and on-chip reconfiguration managers using soft core processors [3] to extensions of a real-time Linux system [1].

All implementations share the concept of a scalable communication backbone with fixed interfaces to reconfigurable modules. The most promising approach of a Network-on-Chip (NoC) [5] is further investigated in [2] showing, that hardwired NoCs may be an integral part of future FPGA devices (see Figure 1) superseding soft-wired System-on-Chip (SoC) buses. They will not only support dynamic reconfiguration, but also improve system efficiency and reduce costs in terms of chip area. In addition, a NoC leads to message based communication in contrast to memory reads and writes. Assuming future multi billion transistor devices integrating a whole bunch of hardwired and reconfigurable computing elements, on-chip communication will play a central role. Sun's slogan "The Network is the Computer" will scale down to those reconfigurable SoCs.

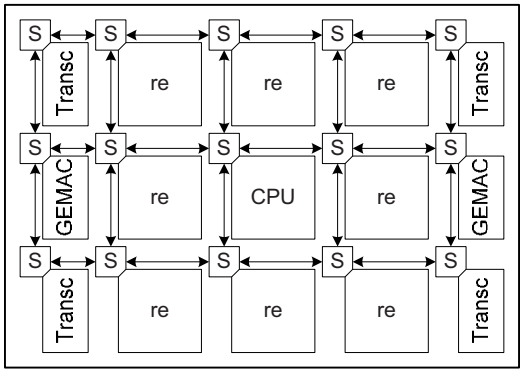

Figure 1. A Hardwired NoC on future FPGAs

Whereas the management tasks for operating systems supporting dynamic reconfiguration are well understood [7, 1], the communication, relocation, and programming strategies are not entirely clear. Especially interruption of reconfigurable cores is significantly difficult. Two major approaches are followed here: reading back the bitstream on the one hand and a controlled shutdown generating a context on the other. However, in this paper a completely new understanding of NoC-based reconfigurable systems is presented. Starting from an object oriented approach [4] in Section 3, the whole system is interpreted as a distributed object system, well understood in the parallel computing world. On top of this idea, a unified programming and communication model is derived. The problems of interruption and relocation are tackled. In Section 4 and Section 5 the experimental environment comprising a standard Linux system and a FPGA simulation model are outlined.

1-4244-0054-6/06/$20.00

Practical applications and their implementation are shown in Section 6. A conclusion and remarks for further improvements summarize this contribution.

2. NoC-based Reconfigurable Systems

Before concentrating on the new ideas of this paper, the overall architecture of a NoC-based reconfigurable system should be outlined. Starting from the structure of dynamically reconfigurable applications, the main building blocks of the operating system and the underlying hardware will be defined. The terminology favored here, contrasts in some points to previously presented architectures, particularly with regard to processes and computing resources.

2.1. Applications for Reconfigurable Systems

Modern applications are designed in an object oriented manner using powerful tools allowing for a graphical design with UML. This does not need to be changed for dynamically reconfigurable systems [4]. Even if hardware synthesis is not fully functional yet, this approach leads to a modular design with communicating computing elements. Every method call is equivalent to a message sent to an object. Thus, applications will have software parts being highly control-driven and hardware parts with a high potential of parallelism. These hardware parts, further on called IP cores, are hidden behind computing objects. These objects own a well defined interface – the methods or messages the object will understand.

2.2. Tiles, IP Cores and Processes

Dynamically reconfigurable devices are and will be partitioned into rectangular areas independently reconfigurable. As shown in Figure 2, these tiles are configured with IP cores. An IP core may span multiple tiles. Thus, IP cores with different sizes and degrees of parallelism are possible.

As IP cores should exploit the parallelism and efficiency of hardware, they must be able to serve multiple clients. For example, if one process accesses only every now and then a crypto core, another process should be able to use the same core to increase its utilization. Following the client-server analogy, this means running multiple processes on an IP core, each serving a client process. Whether supported or not and how it is implemented in a particular core, is not of interest here. The main idea is, that IP cores are computing elements being able to execute multiple processes. Consequently,

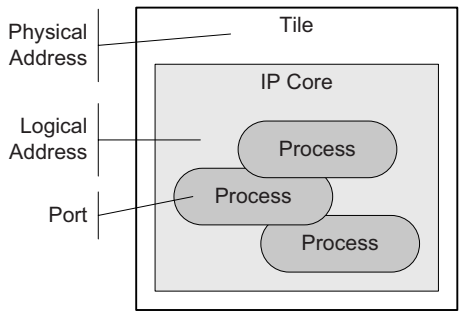

Figure 2. Tiles, IP Cores and Processes

instruction set processors (ISP) are a specific form of IP cores.

2.3. Accelerators and Decelerators

Implementing a certain computing problem often aims at performance improvements. Thus, a hardware realization is called accelerator. Accelerators may come in multiple implementations supporting different levels of parallelism, power, and area consumption. On the contrary, the software implementation of an IP core is called decelerator. It is required to emulate the hardware behavior when reconfigurable resources are exhausted or not existent at all. But they are also useful when improvements of an accelerator are not necessary at the moment.

Accelerators and decelerators of an IP core share the same functional interface. Speaking in object oriented terms, they understand the same methods or messages.

2.4. IP Core Relocation

The process of relocation describes a logical movement of an IP core within the system. An accelerator may be moved into another tile or replaced by a decelerator running on a ISP. Due to increased performance requirements, a decelerator or an accelerator may be substituted by a more powerful accelerator. All this requires to save and to restore the current state of an IP core. This state information is called context.

Besides the functional interface of accelerators and decelerators, the context and the interruption mechanisms have to be compatible as well. This makes the approach of bitstream read-back very inefficient and complicated, as it is highly dependent on the underlying FPGA architecture and the physical structure of the accelerators. Thus, the context should to be extracted by the core itself and not by an external instance.

2.5. Communication

Dynamic reconfiguration involves disconnecting and establishing physical links at the boundaries of reconfigurable modules at runtime. But it also means to control the logical connections of IP cores. This makes the NoC approach highly attractive. Besides fixed physical interfaces, it comes with stacked abstraction layers. Each layer may be substituted by another implementation. Thus, from the application point of view, the physical layer is not of interest.

A NoC allows for a unified communication methodology. Packet based communication facilitates message passing being a fundamental part of an object oriented approach. A shift from physical interconnection details to a high level abstraction of IP core communication is achieved.

It must be stressed, that even if NoCs adapt concepts of Internet communication, they focus on throughput, low power consumption, and small scale communication. Consequently, a full blown TCP/IP stack is not reasonable here.

2.6. Management

The management of the NoC and reconfigurable resources is assigned to an operating system or a middleware. This includes scheduling strategies, allocation and deallocation of tiles and their configuration. IP cores and decelerators must have suitable representations within the operating system. NoC routing tables and logical links between hardware and software are to be controlled during relocation. On the lower end, device drivers for the configuration interface and the NoC must be existent. Figure 3 summarizes the fundamental components of a NoC-based dynamically reconfigurable system in a three layered manner.

3. Dynamic Reconfiguration and Distributed Object Systems

Whereas the system architecture outlined in the previous section is widely accepted, the abstraction and programming models are still heavily discussed. Main problems arise from the diverging paradigms of hardware and software. Unifying both worlds in one description language has been proved to be very difficult, particularly in the field of design automation. Relocation of IP cores within a heterogeneous environment is basically understood, but also lacks of a programming model and design automation. Thus, even if the whole idea of dynamic reconfiguration is promising and has definitely a commercial potential, it is still extremely

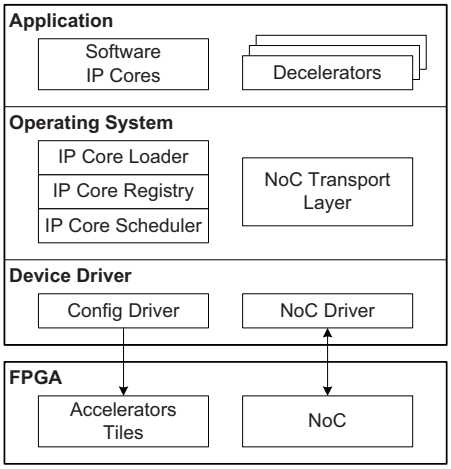

Figure 3. Dynamically rec systems

tricky. But often, complicated things are solved with very simple ideas.

3.1. Everything is an Object

Recent developments in the field of system level modeling languages reveal the strengths of an object oriented design not only for software but also for hardware. As software objects are data containers representing a thing in the real world, they are also used to represent hardware modules. In traditional RTL design, modules have signal ports defining their physical interface. But transaction level modeling, emerged from system level design, teaches to raise the abstraction level from the physical details to a functional description of the interface. This description is basically a collection of methods a module understands. Modules are allowed to have multiple interfaces. A network controller has a separated receive and transmit interface for instance. Interfaces may be stacked. This means, that different abstraction layers of an interface may exist. A protocol stack is an example here.

Figure 4 illustrates the basic concept of hardware modules being objects. At a high abstraction level, every module has one or more functional interfaces aggregating methods. How the methods are mapped to a physical interface is defined at design and compile time and depends on the desired throughput and costs. Pure combinational interfaces, handshakes, FIFOs, bus interfaces, and even networks fit for this methodology.

3.2. How Objects communicate

The second fundamental concept of object oriented thinking says, that objects communicate by the use of

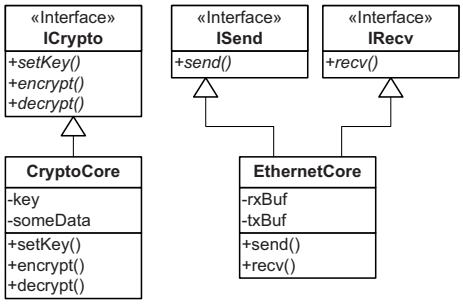

Figure 4. Hardware Modules are Objects

messages. Objects may understand these messages, if they know a corresponding method. Due to performance reasons, most object oriented languages have abandoned the distinction between messages and methods, thus, dealing only with methods. But objects representing hardware modules are remote from a calling instance. Methods can not be called directly.

This problem is solved with remote method invocation (RMI). If one object wants to invoke a method of a remote object, a real message is generated. This message is sent over a network to the remote object. A reply message containing the result of the method call is sent back to the invoking instance. As Figure 5 shows, the translations from methods to messages and vice versa are performed by a proxy and a skeleton. Whereas the proxy resides at the client side imitating the remote object, the skeleton interfaces the real object to the network. Thus, proxy, skeleton, and network act as invisible intermediate layers between client and object.

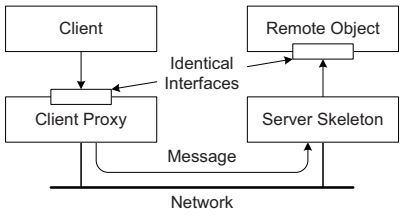

Figure 5. Client Proxy and Server Skeleton

As the reconfigurable system features a NoC, this concept is easily adapted. Most prototypes follow the approach of sending messages to communicate with reconfigurable cores, even an object oriented design not in mind.

3.3. Distributed Objects

Generalizing the idea of IP cores being remote objects, leads to distributed object systems, such as CORBA, DCOM, and Java RMI. From the application point of view, all objects seem to be located on the local host. But objects are allowed to be on remote machines, concealed by client proxies. Objects can be allocated and deallocated at will. Even remote objects may allocate other objects being local or remote. Objects can be shared and can migrate between the client and remote hosts. A lookup or directory service assists locating objects within the network.

But what does that mean if applied to dynamically reconfigurable systems? If an application wants to use an IP core, it simply creates an instance of it. The proxy and, if enough resources available on the FPGA, the accelerator are loaded. The decelerator equals an object on the local machine. As shown in the following example, communication with the core is simply done by method calls. The NoC is hidden by the proxy and the skeleton. If the application does not want to use the core anymore, it just destroys the instance. Reconfigurable resources are freed.

```
// Load an IP core, Decelerator or
// Accelerator are initialized
myCore = myCore::createInstance();

// Do something with it,
// Remote method invocation
myCore->doSomething();

// Unload the IP core,
// Free reconfigurable resources
myCore->releaseInstance();
```

Relocation of IP cores is possible, if it is, speaking in an object oriented terminology, serializable. Serializing means to save the internal state information after an atomic operation. Thus, not all registers of an IP core have to be saved, but only those representing the state between atomic operations. To access IP cores, a remote reference has to be used. This not only allows the client but also other IP cores to communicate with it. According to distributed object systems, a naming and lookup service has to be implemented to find IP cores within the network.

Obviously, a dynamically reconfigurable system may be understood as a distributed object system in the small. Even if this analogy does not simplify dynamic reconfiguration, it definitely aids developing future concepts and architectures.

4. Middleware Architecture

Based on the proposed analogy, a new middleware architecture for dynamically reconfigurable systems has

been developed. It resides on top the NoC transport layer supporting point-to-point connections, a logical addressing scheme and ports. Alike TCP/IP, logical addresses specify IP cores and ports the processes on IP cores.

As shown in Figure 6, applications use the client side representation of IP cores – the proxy. It hides the complexity of communication and reconfiguration. A proxy comprises the application protocol translating methods to messages and a naming service responsible for dynamic reconfiguration. The naming service acts as an interface to the IP core lookup service and the IP core loader. The IP core lookup service searches for existing IP cores matching the requested service and returns its remote reference. A remote reference consists of a service ID describing the function, the logical address of the IP core, and the port of the service. If the requested service is not available within the network, a new IP core will be loaded by the IP core loader creating a new remote reference.

IP cores have at least two implementations, the accelerator and the decelerator. Both share the same skeleton translating messages into methods. Whereas the accelerator is only accessible over the network, the decelerator has an additional interface to bypass network communication. This speeds up the interaction between software and decelerators. The network interface of decelerators is required to allow a communication between IP cores even if not running in hardware.

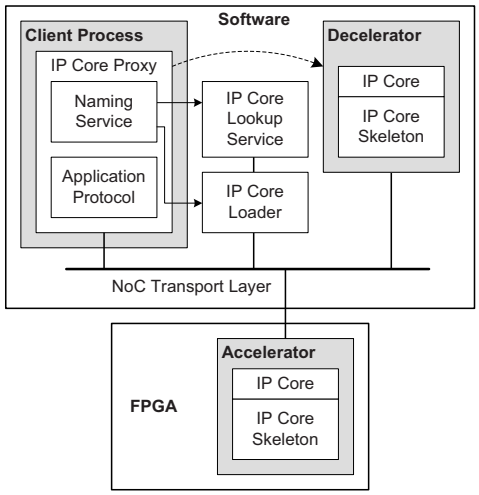

Figure 6. The NoC Object System

Besides a software interface to proxies, the IP core lookup service and the IP core loader are visible within the network. This allows IP cores to find, to create, and to use other IP cores by themselves. Consequently, hardware and software have equal rights to trigger dynamic reconfiguration and to communicate among each other. This approach is a fundamental improvement compared to previously presented systems, where dynamic reconfiguration is requested by a central instance, in most cases the software part of the application.

5. Experimental Environment

As shown in Figure 7, besides a Debian Linux system with kernel version 2.6.9, an abstract FPGA simulation model was used. This model is written in SystemC 2.1 and features a NoC and 16 dynamically reconfigurable tiles. The reason for such a model is, that dynamic reconfiguration is very cumbersome today. Tool support is still under development. Tile-based dynamic reconfiguration of Virtex-4 is possible but not supported yet. A soft-implemented NoC consumes to much resources, resulting in just a few reconfigurable tiles. Instead, the SystemC model assumes a dynamically reconfigurable FPGA with an hardwired NoC. The FPGA-model is attached to virtual Linux devices allowing configuration of the tiles and NoC communication.

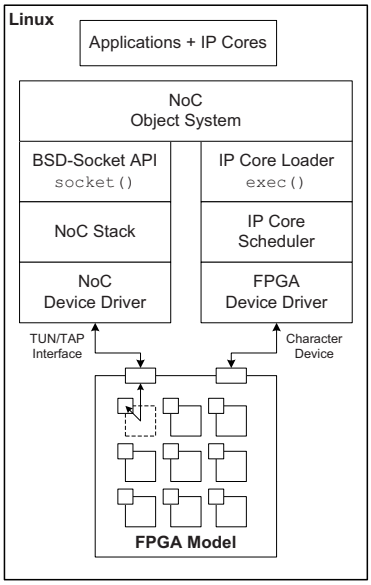

Figure 7. Linux Experimental Environment

The Linux NoC extensions are integrated similar to other network systems. The transport layer is accessible with standard BSD socket API calls. Thus, direct communication with IP cores is as simple as TCP/IP networking. Moreover, this implementation is very fast, as the whole Linux network subsystem is highly optimized for packet based communication.

The object system managing dynamic reconfiguration resides on top of the NoC transport layer. It makes

use of the BSD socket API and the `exec()` system call to load new IP cores. For this purpose, the Linux ELF loader was extended to support IP core description files. They contain the names of accelerator and decelerator files.

Decelerators are represented by Linux processes. Accelerators run within the FPGA model. Bitstream download is not modeled. Configuration is simply done by selecting a certain accelerator and a tile over the configuration interface. As this contribution focuses on design and programming paradigms and the overall system architecture rather than efficient resource sharing, a very simple scheduler was implemented. If the tiles are exhausted, the decelerator is used.

6. Case Studies

To evaluate the proposed architecture and its impact on the design flow, various case studies have been done. Two of them will be outlined here, starting with an AES crypto core demonstrating the acceleration of compute intensive applications. A second example shows how to integrate a communication controller alike the hardwired components depicted in Figure 1. Both case studies will highlight the seamless refinement process from a pure software solution to the final IP core implementation comprising the client side, the decelerator, and the accelerator.

6.1. AES Crypto Core

Following the principle "IP cores are objects", a software implementation of the AES taken from the Linux kernel was encapsulated within a class. This class features an interface describing the messages an AES core will understand. The very basic interface methods are `setKey`, `encrypt` and `decrypt`. The next step was to define the communication protocol and the message format. A message starts with a request/reply qualifier followed by an opcode specifying the method to be invoked. The parameters of request messages and results of reply messages are appended.

The translation between methods and messages are put in a client proxy and a server skeleton. The latter is attached to the AES class, whereas the client proxy is used by an application. Communication with the NoC transport layer and IP core look-up resist in precompiled libraries. Now, the decelerator was implemented. Decelerators are standard console applications comprising the AES software implementation class and the server skeleton.

The Linux extensions allow to load IP cores by the use of the `exec()` system call. It returns the logical address of the IP core being the same as the Linux process ID of the decelerator. IP core executables are text files containing the names of the decelerator and accelerator file. The AES IP core executable `aes.core` looks like

```
dec:AES.dec
acc:AES.acc
```

To verify the application and the decelerator only, the accelerator was omitted first. To use the AES core within an application, an instance of the AESCore class has to be created

```
AESCore* myAESCore;
myAESCore = AESCore::createInstance();
```

The creation of a virtual connection to the decelerator and the loading mechanism of IP cores is completely hidden behind this creation. The core is ready for use now. The methods may be called.

```
myAESCore->setKey(aKey);
aCypher = myAESCore->encrypt(aMessage);
```

When all desired operations are done, the IP core must be released. This does not necessarily mean, that the IP core is unloaded. It may stay within the system to serve other applications. On the other hand, this approach allows IP cores to serve more than one client. Such a release does only close the connection to the core and frees its utilized computing resource to be available for other applications.

```
myAESCore->releaseInstance();
```

After the decelerator was working as expected, the implementation of the accelerator was started. As a SystemC model instead of a real FPGA was used, this design step was very simple. The AES class was put together with the server skeleton in a dynamically reconfigurable SystemC module. These modules feature an reconfigurable interface to the transport layer of the NoC. They are configured on request through the configuration interface of the FPGA simulator.

If a real FPGA is available, the AES accelerator has to be refined for RTL synthesis using SystemC, VHDL or Verilog. As modern tools support mixed language simulation, the FPGA model can be reused.

6.2. Synchronous and Asynchronous Messages

As described above, every method call results in a request and a reply message. The caller halts until the reply message is received. This is formally known as

synchronous message passing. Communication and computation are not pipelined which highly degrades the obtained throughput. Especially streaming applications will suffer from this methodology.

In contrast, asynchronous message passing allows deferred reply messages. To integrate this communication scheme into the AES core, asynchronous method calls for encryption and decryption were added to the client proxy. Results are received with handlers to be registered before communication.

```
void resultHandler(result) { ... };
myAESCore->resultEvent(resultHandler);
myAESCore->encrypt(aMessage);
```

At the beginning, the client proxy is informed about a result handler. This function will consume encryption results. An encryption is still triggered with `encrypt()`, but it does not return the result by itself. Instead it is handled by the function `resultHandler()`.

This programming technique is only to be concerned for software parts of an application and decelerators. In hardware, the request and reply data path may be buffered and parallelized to achieve asynchronous message passing.

6.3. Remote references

The previous code examples demonstrated, how to explicitly load IP cores. Instead of pointers addressing dynamically created IP cores, remote references may be used. They contain the specific service ID of the core, its logical address and port number. One advantage of this approach is to defer the creation of an IP core until it is actually used. In the following example the creation is deferred until the encrypt method is called.

```
AESCoreRef myAESCore;
myAESCore->encrypt(aMessage);
```

Thus, the user does not have to explicitly create an instance of the AES core. This is hidden behind the operator `->`, reducing the code overhead for the user.

Besides simplification, this is a consistent approach to share IP cores between applications and other IP cores. Instead of pointers, remote references exactly address a service end point within the system and are thus relocatable. The entire communication rests upon them, decentralizing the dynamically reconfigurable system.

6.4. Relocation

Relocating an IP core within a dynamically reconfigurable system requires to save its internal state or to serialize it, speaking in terms of distributed objects. In both worlds, this is not straight forward and sometimes even not possible. The approach taken highly depends on the internal structure and functionality of a particular core. The amount of data to be saved must be balanced with the time to transfer and store it and the time required to stop and to restart the core. Therefore, it is not reasonable to save all internal register values nor to completely dismiss the state. IP cores should be allowed to finish atomic operations before interruption, alike general purpose processors.

In the case of the AES core, only the key committed by the application is saved. This is due to the fact that key expansion is very fast. In our hardware reference design, it takes only 48 clock cycles and is much faster than to transfer the expanded key.

To integrate interruption and restoration, every IP core features beside its functional interface a relocation interface. As shown in Figure 8, this interface defines methods to start, to stop, and to reset an IP core.

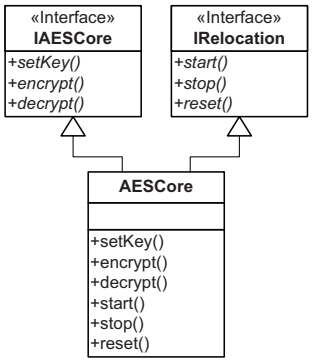

Figure 8. IP Core Relocation Interface

To transmit relocation requests over the NoC, the methods are translated to messages extending the application protocol of the core. These messages are generated and processed by the distributed object system and scheduled by the IP core scheduler.

6.5. Network Controller

Besides classical accelerators, IP cores may implement communication controllers for off-chip devices, such as multi-gigabit or Ethernet transceivers. It is expected that those cores are not relocatable, due to their connection to specific pads of the FPGA.

The network controller is again treated as an object. It will understand messages to receive and to transmit packets as well as to adjust communication parameters. Even though an asynchronous interface should be preferred, the client side may do

```
myEthCore->setSpeed(100);
myEthCore->send(sData);
rData = myEthCore->recv();
```

Thus, the transmitted data is encapsulated in NoC messages. But an Ethernet controller is hardly accessed from an application directly. It rather should be integrated as a traditional network device. For this purpose a low level Linux kernel interface to NoC sockets was realized. It allows to use regular socket operations within the kernel context. Figure 9 depicts the control and data flow from a conventional Linux network device to the communication controller in the NoC.

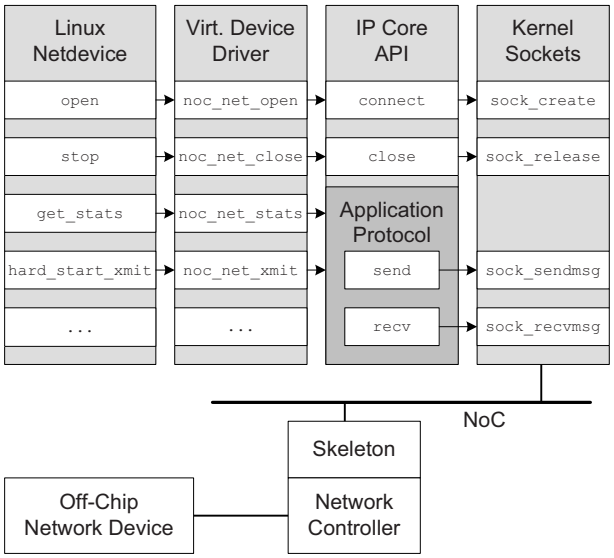

Figure 9. The NoC as a Network Bridge

The Linux network device accesses the virtual Ethernet device with usual device functions. The device functions are mapped to IP core methods and encapsulated into messages. The messages are sent over the NoC and processed by the IP core.

7. Conclusion and Outlook

By applying an approved methodology from the parallel computing world to dynamically reconfigurable systems, a new approach to design dynamically reconfigurable applications was presented. Reconfigurable modules are treated as objects communicating with software or other modules by the use of messages. The reconfigurable system is understood as a distributed object system, allowing remote objects to relocate between reconfigurable tiles or processors. The question of interrupting and restoring modules is simply answered by the concept of serializing objects. Even if this is also not simple in distributed object systems, it sets up a formalized foundation for further investigations on dynamically reconfigurable systems.

Taking a crypto accelerator and a communication controller as examples, the implications of the presented analogy were studied. It was shown, that the design process from a pure software solution to an accelerator is driven by a step-by-step refinement procedure. This is well supported by object oriented modeling languages, such as SystemC. The extensions made to a Linux system and the use of a SystemC FPGA model provided an excellent development and evaluation platform for dynamically reconfigurable applications.

As the design process is straight forward, it may be automated by tools. In particular, the creation of messages, the client proxy, and the server skeleton from a module interface has to be simplified by the use of interface description languages (IDL) as already used in CORBA. In addition, sharing remote references must be further studied. It has high potentials in dynamic creation and interconnection of reconfigurable modules and leads to a more general understanding of dynamically reconfigurable systems.

References

[1] A. Bartic et al. Network-on-Chip for Reconfigurable Systems: From High-Level Design Down to Implementation. In *FPL 2004*, Leuven, Belgium, August 2004.

[2] R. Hecht, S. Kubisch, A. Herrholz, and D. Timmermann. Dynamic Reconfiguration with hardwired Networks-on-Chip on future FPGAs. In *FPL 2005*, pages 527–530, Tampere, Finland, August 2005.

[3] M. Hübner, K. Paulsson, and J. Becker. Parallel and Flexible Multiprocessor System-On-Chip for Adaptive Automotive Applications based on Xilinx MicroBlaze Soft-Cores. In *RAW 2005*, Denver, Colorado, USA, April 2005.

[4] M. Hübner, K. Paulsson, M. Stitz, and J. Becker. Novel Seamless Design-Flow for Partial and Dynamic Reconfigurable Systems with Customized Communication Structures Based on Xilinx Virtex-II FPGAs. In *ARCS 2005, Workshops Proceedings*, LNI, Innsbruck, Austria, March 2005. GI.

[5] T. Marescaux et al. Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs. In *FPL 2002*, Montpellier, France, September 2002.

[6] H. Walder and M. Platzner. A runtime environment for reconfigurable hardware operating systems. In *FPL 2004*, pages 831–835, Leuven, Belgium, August 2004.

[7] G. Wigley and D. Kearney. Research Isssues in Operating Systems for Reconfigurable Computing. In *ERSA'02*, Las Vegas, Nevada, USA, June 2002.