ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für
Technische Informatik und
Kommunikationsnetze

Richard Huber

A Dynamic Hardware Architecture for Future Networks

Master Thesis MA-2011-27
December 2011 - June 2012

Advisor: Ariane Keller, ariane.keller@tik.ee.ethz.ch
Co-Advisor: Dr. Stephan Neuhaus, stephan.neuhaus@tik.ee.ethz.ch
Co-Advisor: Daniel Borkmann, daniel.borkmann@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

Contents

Chapter 1

Introduction

1.1 Motivation

The increasing number and diversity of networking devices is challenging the network research community. More and more applications and protocols are developed to address the needs of different platforms and use cases. For performance reasons, these protocols and applications are often tightly bound to each other. The resulting systems may then perform well, but lack of flexibility in the underlying architecture. This inflexibility is well demonstrated by the long lasting and still ongoing switch from IPv4 to IPv6 in the TCP/IP protocol stack of the current Internet architecture.

The Autonomic Network Architecture (ANA) [**?**] targets this problem with a fundamental change of paradigms. In ANA, all functionality is encapsulated in so called functional blocks. The size of functional blocks is a priori not defined. It may range from a minimalistic checksum calculation up to a huge holistic protocol stack. ANA provides a set of abstraction models and generic communication methods that the functional blocks use to interact with each other. This framework allows a completely isolated development of the algorithms running in the functional blocks. The operating system then interconnects the functional blocks and builds up an individual protocol stack. The device chooses the functional blocks for this protocol stack depending on the capabilities of the platform and the requirements of the running applications. The result is a flexible and dynamic protocol stack, that is able to adapt itself to any given situation and may change at runtime. Figure 1.1 shows such a dynamic protocol stack and compares it to the inflexible, static and strongly layered protocol stack of the current Internet.

The existing implementations of ANA [**?**], [**?**] do not yet reach the performance of the traditional Internet architecture. This is due to two reasons. First of all, the level of indirection introduced by ANA trades some performance optimization features of the application for a higher flexibility and modularity of the overall system. The second reason is that the traditional Internet architecture benefits from a wide range of hardware support in the networking interface cards, which is not yet available for ANA. In order to compensate the performance penalty of ANA, a hardware acceleration for ANA is needed. It is not feasible to use application specific integrated circuits (ASIC) for this purpose, since the functionality of these devices must be completely specified at design time. This static characteristic of ASICs would conflict with ANA's effort towards flexibility and adaptability.

The evolving partial reconfiguration capabilities of modern field programmable gate arrays (FPGA) combine the flexibility of a software implementation with the high-performance acceleration features of ASICs. Lübbers et. al. [**?**] and Keller et.

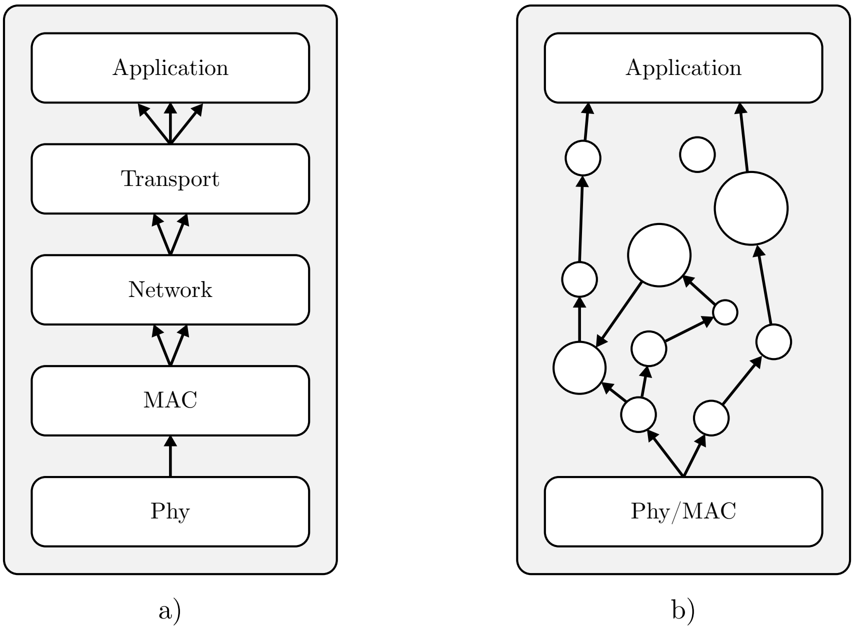

Figure 1.1: a) The strongly layered static protocol stack of the current Internet, b) The dynamically adaptable protocol stack of an ANA node

al. [?] therefore came up with the idea of porting ANA to a reconfigurable system on chip (rSoC). On these devices, a microprocessor core (either soft- or hardcore) is combined with a programmable logic fabric. The software running in the microprocessor may then decide to offload the execution of computational intensive and frequently used functional blocks from the microprocessor to the programmable logic. Based on the partial reconfiguration features, this mapping of functional blocks to hardware or software can be adapted at runtime.

In order to reduce the engineering effort, Lübbers et. al. [?] and Keller et. al. [?] further suggest to use ReconOS [?] as base system. In ReconOS, partially reconfigurable logic slots are used as so called hardware threads. These hardware threads appear to the software like normal software POSIX-threads. ReconOS enables the hardware threads to access operating system objects like semaphores, message boxes and shared memory. The combination of these mechanisms simplifiess the interaction of software and hardware elements. The suggested approach benefits from the hardware acceleration provided by ReconOS by implementing some functional blocks as hardware threads. Figure 1.2 shows the architecture of ReconOS and how the functional blocks are mapped to the hardware threads.

However, the major problem of this approach is the shared bus architecture of ReconOS. Especially with a data intensive application like ANA, the shared bus soon becomes the bottleneck of the system.

1.2 Aims of this Thesis

In this thesis we present an extension of the ReconOS framework with a high-throughput communication infrastructure shown in figure 1.3. This infrastructure contains a packet based network on chip (NoC) that guarantees each hardware thread a worst-case incoming and outgoing data throughput. Additionally, a ring buffer based gateway is designed and implemented to speed up the transfer from

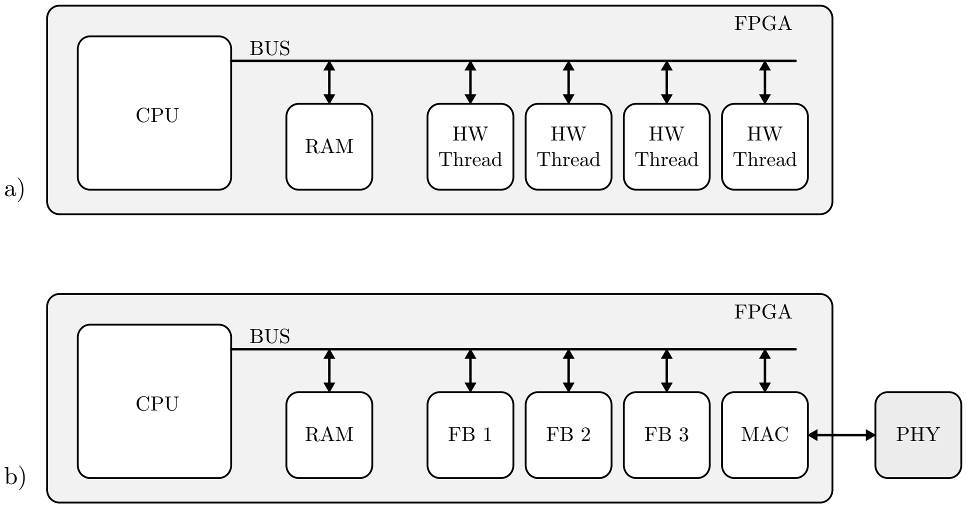

Figure 1.2: a) The ReconOS architecture: partially reconfigurable logic slots are used as hardware threads. b) Using ReconOS for ANA: a set of functional blocks is executed in hardware threads.

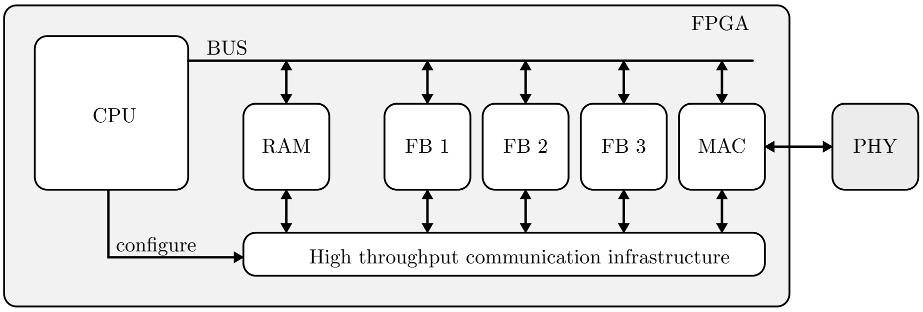

Figure 1.3: The high throughput communication interface guarantees the functional blocks a worst case incoming and outgoing bandwith.

software to hardware and vice versa.

1.3 Organization of the Thesis

The reminder of this thesis is organized in the following way: TODO!

Chapter 2

Design

We start this chapter by introducing the design goals of the ReconOS extension in section 2.1. This is followed by an evaluation of architectures and NoC topologies in the sections 2.2 and 2.3, respectively. Section 2.4 describes the design of the hardware/software data transfer mechanism.

2.1 Design Goals

In ReconOS, multiple hardware threads can execute tasks independent of the CPU. To this end, the hardware threads and software running in the CPU still need to exchange data with each other. ReconOS provides different mechanisms for this purpose, most notably message boxes and shared memory. All of these mechanisms have in common that they use a shared bus for data transmission. This shared bus architecture is not well suited for networking applications like ANA. Imagine a packet that has to be processed by a set of functional blocks, all of them mapped to hardware threads. In this case, the bus has to repeatedly forward the whole packet from one functional block to the next. The overall data transfer on the bus therefore increases linearly to the number of participating functional blocks.

In this thesis, we want to extend the ReconOS architecture with a high-throughput communication infrastructure. The data rate at which each functional block can send data over the communication infrastructure to a specific destination should be independent of the traffic passing by targeting a different destination.

We distinguish three different applications of data transfer:

- Data transfer from a hardware thread to a software thread
- Data transfer from a software thread to a hardware thread
- Data transfer from one hardware thread to another hardware thread

The case of data transfer from one software thread to another software thread is out of the scope of this thesis.

ANA processes packets in two distinct directions. *Ingress* packets arrive at the physical interface, traverse the protocol stack upward and finally reach the corresponding application. *Egress* packets have their origin in the application layer, traverse the protocol stack downwards and then leave the device on the physical interface.

Throughput In this thesis, we assume that the size of a packet does not significantly change on its way through the protocol stack. We can therefore set an upper bound of the total data rate at the input and output port of each functional block.

This upper bound is equal to the data rate of the physical interface. In our development environment we use a Xilinx ML605 evaluation kit [?]. Its physical interface is a Gigabit Ethernet Module.

We therefore specify the following requirement for the communication infrastructure: The outgoing data rate of each functional block is upper bounded by 1 Gigabit/s. The communication infrastructure then forwards 1 Gigabit/s of data to each hardware-mapped functional block, independently of the origin of the data.

Latency Data transfers that cross the hardware/software boundary cause an orders of magnitudes higher latency than data transfers in the hardware domain [?]. This is caused by shared memory access and by the high overhead of unavoidable interrupts. The total latency of an ingress or egress of packet is therefore mainly influenced by the performance of the hardware to software or software to hardware data transfer, respectively. We therefore neglect latencies caused by the communication infrastructure between hardware threads and focus on the latencies induced by crossing the hardware/software boundary.

Hardware Resource Utilization The desired communication infrastructure is only a helper construct for the actual application running on the device. Hence, we want to keep it as small as possible, so that there is a maximum of programmable logic still available for the application.

2.2 Evaluation of Architectures

In this section, we discuss possible design principles for the communication infrastructure. The discussion is based on the results of Lee et. al. [?]. In subsection 2.2.4, we then conclude which design principle best fits to the design goals stated in section 2.1.

2.2.1 Point to Point Interconnection (P2P)

In a P2P architecture, all hardware threads are directly connected to each other. The data transmission rate and the latency is optimal, since the packets do not need to share resources with any other transmission. The receiving hardware thread is responsible for the serialization of packets that concurrently arrive on different input ports. Lee et. al. [?] show that the logic and routing resource utilization of this architecture scales very bad with an increasing number of peers.

2.2.2 Multibus

The multibus architecture is a mixture of the original shared bus architecture of ReconOS and the P2P architecture. In this architecture, all hardware threads read from their own individual bus. In order to send a packet to a hardware thread, the source hardware thread writes the packet to the bus of the destination hardware thread. An individual bus arbiter is required for each hardware thread.

The data rate on this architecture is optimal, since each hardware thread can receive data with the full data rate of its bus. The latency depends on the response time of the bus arbiter. Since all hardware threads have their own bus that must connect to each hardware thread, we assume that this architecture has a high routing resource utilization. We further assume, that the long bus wires and the parasitic capacities of the high impedance bus interfaces limit the clock rate on the bus when the number of hardware threads increase.

2.2.3 Network on Chip (NoC)

A NoC is a packet based communication architecture. Intellectual property (IP) cores (hardware threads in the context of ReconOS) are connected to a dedicated switch. The switches are then interconnected with each other according to a selected topology. When a packet arrives on an incoming link of a switch, the switch selects an outgoing link depending on the packet's header and forwards the packet on this link.

A NoC is subject to many configuration issues, e.g. the topology, the number of IP cores per switch or the routing policy. This configuration possibility enables the developer to adapt the NoC architecture to any given requirements.

The latency of the packets and the flow through the network strongly depend on the selected topology and the routing policy. The NoC architecture has a higher latency compared to the Multibus architecture. The reason for this is that the packets must traverse the network hop by hop. Thereby every hop induces a latency of one clock cycle and possibly an additional latency caused by the routing mechanism. On the other hand, the length of the hop signals - and with this also their clock rate - is independent of the number of IP cores. This is not the case for the Multibus architecture. We can therefore state, that the NoC architecture scales better with increasing number of hardware threads than the Multibus architecture.

2.2.4 Conclusion

Based on the discussion in subsections 2.2.1 to 2.2.3, we assigned grades from 1 to 10 to each presented architecture. The total grade of the architectures is a weighted average of the individual grades. The grades are:

- **Data Rate** (weight 5): The average throughput of the architecture. 10 is the highest throughput.
- **Latency** (weight 2): The average latency of the packets. 10 is the lowest latency.
- **Chip Area** (Weight 3): The expected chip area consumed by the architecture. 10 is the smallest chip area.
- **Engineering Effort** (weight 2): The engineering effort for developing an implementation of the architecture. 10 is the least effort (e.g. ReconOS as is).
- **Scalability** (weight 4): The scalability of the architecture in all the above mentioned criteria for an increasing number of hardware threads. 10 is the best scalability.

The assigned grades are shown in figure 2.1. As we expected, the global bus architecture (weighted average 3.81) and the P2P architecture (weighted average 5.94) get significantly lower grades than the Multibus architecture (weighted average 7.13) and the NoC architecture (weighted average 7.63). The higher weighted average of the NoC architecture compared to the Multibus architecture is mainly due to the better scalability.

Based on these results, we decided to implement the communication infrastructure as NoC. An evaluation of NoC topologies is presented in section 2.3.

2.3 Evaluation of NoC Topologies

The performance of a NoC heavily depends on the topology used to interconnect the switches and the corresponding routing policy. In this section, we first present the

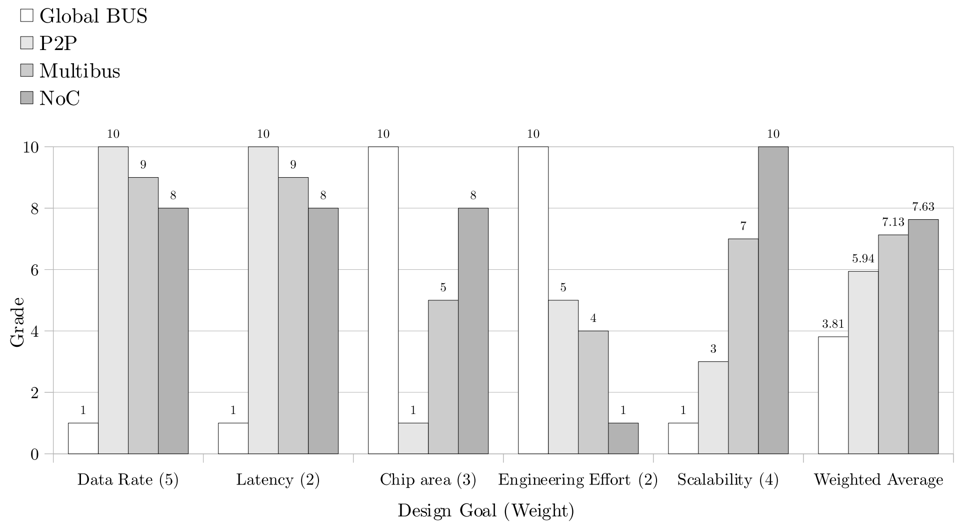

Figure 2.1: For each design goal a grade ranging from 1 to 10 is assigned to each architecture. The weighted average of the grades represents the overall suitability of the architectures for the problem at hand.

evaluated topologies and routing mechanisms in subsection 2.3.1. Subsection 2.3.2 then presents the *total link flow* measure used to estimate the resources consumed by the links of a given NoC topology. In subsection 2.3.3, we then conclude which topology fits the requirements best.

In this thesis we ignore load balancing since any benefit from this approach would be cancelled out by the worst-case traffic pattern.

2.3.1 Topologies and Routing Policies

The topology of an NoC describes how the switches are interconnected with each other and how the functional blocks are connected to the switches. The routing policy specifies the outgoing link of a switch on which an incoming packet is forwarded depending on the destination address of the packet. The topology is specified at design time and does not change at runtime. Each switch has therefore a global knowledge on the topology and can use this knowledge for the routing decision.

Grid topology In the grid topology, the switches are arranged in a mesh. Figure 2.2 a) shows an example of the grid topology. We evaluated this topology with the xy-routing policy. In this routing policy, an x- and an y-coordinate is assigned to each switch. The switches forward packets in the x-dimension until the x-coordinate of the destination address and the x-coordinate of the processing switch is equal. The switches then forward the packets in the y-dimension until the packet reaches its destination.

An advantage of the grid topology is, that it is easily mappable to the available chip area. A big drawback is that it is not applicable to all possible number of functional blocks. If the number of functional blocks is not a square number, the resulting topology will not be optimal.

Torus topology The torus topology is build in the same way as the grid topology with the exception that border switches are also interconnected. An example of the

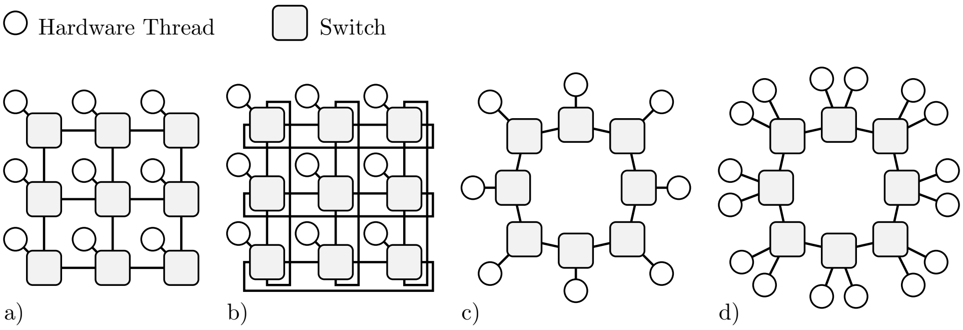

Figure 2.2: Examples of NoC topologies: a) Grid b) Torus c) Ring with one hardware thread per switch d) Ring with two hardware threads per switch

torus topology is shown in figure 2.2 b). For the torus topology we also used the xy-routing policy. The advantages and disadvantages of the torus topology are much the same as for the grid topology.

Ring topology In the ring topology, the switches are arranged in a ring as shown in figure 2.2 c) and d). We distinguish two different routing policies: *unidirectional* routing and *bidirectional* routing. Switches using the unidirectional routing policy forward all packets to the same switch. Therefore all packets travel in the same direction, e.g. clockwise. Switches using the bidirectional routing mechanism first determine the number of hops to the destination of the packet in the clockwise and anticlockwise direction. They then send the packet in the direction where the number of hops is smaller.

We further extend the ring topology by connecting multiple functional blocks to the switches. The destination address of the packets then consists of a *global address* and a *local address*. The global address represents the switch to which the destination functional block is connected. The local address identifies the functional block in the set of functional blocks connected to the same switch. The combination of global and local addresses is then unique within the NoC.

The routing mechanism of the ring topology (especially when using unidirectional routing) is easier than xy-routing. We therefore expect that a switch in the ring topology requires less hardware resources than a switch in the grid- or torus topology.

2.3.2 The Total Link Flow Measure

In order to guarantee the required throughput for the hardware threads, each link in the NoC must be able to forward an individual worst-case data rate. We adapt the capacity of each link to its worst-case data rate by manipulating the number of parallel channels the link contains. We assume that the hardware resources needed by a link is approximately proportional to its number of parallel channels.

We define the *total link flow* of a topology as the sum of the worst-case data rates on the individual links of the whole topology. According to the above argumentation, this measure is approximately proportional to the total hardware resources used by the links of the NoC.

Algorithm In order to estimate the resources used by the links of an NoC, we designed an algorithm that calculates the total link flow for an arbitrary NoC given the NoC's topology and routing policy. In this algorithm we assume, that each hardware thread is able to consume and produce data at exactly the rate 1.

The idea of the algorithm is to virtually send explorer packets from each hardware thread to all other hardware threads, containing the ID of the source and destination hardware thread. Every time a packet traverses a link, the link adds the source and destination ID pair to a link-local list. This list is then interpreted as a bipartide graph with the senders in one partition and the receivers in the other partition. The worst-case data rate of a link is then determined by the cardinality of the maximum matching of the bipartide graph.

Here is how the algorithm works:

1. Set up the network as a directed graph $G_{net} = (V, E)$ with
   (a) V the set of all nodes v
   (b) E the set of all links $e = (v_i, v_j)$ able to forward packets from v_i to v_j.
2. Let $E_{out,i}$ be the set of negative incident edges of node v_i.
3. For all nodes v_i in V select a routing policy $R_i : v_{dest} \rightarrow e_{out}$ where $v_{dest} \neq v_i$ is the destination of the routed packet and $e_{out} \in E_{out,i}$ is the outgoing link chosen to forward the packet.
4. Send explorer packets
   (a) All nodes $v_{src} \in V$ send exactly one packet $p = (v_{src}, v_{dest})$ to all nodes $v_{dest} \in V, v_{src} \neq v_{dest}$.
   (b) All links $e_i \in E$ refer to a bipartide graph $G_{link,i} = (S_i \cup D_i, L_i)$. S_i, D_i and L_i are initialized as empty sets. Upon forwarding packet $p = (v_{src}, v_{dest})$ on the link e_i, the link
      i. adds v_{src} to S_i (if $v_{src} \notin S_i$),
      ii. adds v_{dst} to D_i (if $v_{dest} \notin D_i$),
      iii. adds the edge (v_{src}, v_{dest}) to L_i.
5. Select the worst case data rate on all links e_i
   (a) For the bipartide graph $G_{link,i}$ find a maximum matching M_i (e.g. using the Hopcroft-Karp algorithm).
   (b) From graph theory we know, that for a bipartide graph, the cardinality $|M_i|$ of a maximum matching is equal to the cardinality of the maximum flow $|f_i|$ from one partition of the graph to the other.
   (c) The worst-case data rate on the link e_i is therefore equal to $|M_i|$.
6. The total link flow f_{tot} of the given topology and routing policy is then $f_{tot} = \sum_{v_i \in V} |M_i|$

Results We implemented the algorithm presented above using the Java programming language. Table 2.1 presents the results for a set of topologies and routing algorithms. Additionally, figure 2.3 shows a graphical visualization of the most representative values.

We see from these results, that the simplest version of the ring topology (with 1 functional block per switch) has a much higher total link flow than the grid topology. With an increasing number of functional blocks connected to the switches in the ring topology, the difference in the total link flow decreases more and more. Finally, the total link flow of the ring topology with 4 functional blocks per switch is comparable to the total link flow of the grid topology.

Nodes		1	4	6	9	12	16	20	25
Topology	Routing								
Grid	XY	0	8		36		96		200
Torus	XY	0	8		36		96		200
Ring (1 FB/SW)	Unidirectional	0	12	30	72	132	240	380	600
Ring (1 FB/SW)	Bidirectional	0	12	30	72	132	240	380	600
Ring (2 FB/SW)	Unidirectional	0	12	24	53	84	144	220	349
Ring (3 FB/SW)	Unidirecitonal	0	10	18	36	60	110	159	248
Ring (4 FB/SW)	Unidirectional	0	8	16	33	48	80	120	197

Table 2.1: Total link flow of NoCs with a specific topology and routing algorithm.

The most surprising result is that the total link flow of the grid topology is exactly the same as that of the torus topology. The total link flow of the various ring topologies is also independent of the chosen routing algorithm.

2.3.3 Conclusion

The evaluation of NoC topologies results in a trade-off decision. On one hand we have the grid topology with a relatively complex routing mechanism and a low resource consumption for the links. On the other hand we have the ring topology with a very simple routing mechanism and a high resource consumption for the links. The ring topology with multiple functional blocks per switch seems to be a good trade-off between these two extremes. We therefore decided to implement the ring topology with an adaptable number of functional blocks per switch. Figure 2.4 shows an example of the resulting architecture with two functional blocks per switch and three parallel channels between the switches.

2.4 Hardware - Software Data Transfer

In sections 2.2 and 2.3 we focused on the exchange of data between two hardware threads. In this section we now want to concentrate on the data transfer between hardware- and software threads.

2.4.1 Utilities provided by ReconOS

Message Boxes In ReconOS, a thread - either hardware or software - can write 32 bit words to a message box. These words stay in a message box until any thread reads them. The write and read access of message boxes is automatically thread-safe. When a thread reads from an empty message box, the corresponding system call blocks until another thread writes data into the message box. ReconOS raises an interrupt at the CPU whenever a data word is written to a message box. It is therefore not feasible to handle high volume data transfers with message boxes, since this would cause a very high interrupt load. Message boxes are more appropriate for notifying other threads about infrequent events.

Shared Memory ReconOS provides hardware and software threads with concurrent access to shared memory. Typically, a software thread allocates the shared memory segment and then sends the starting address of the segment to the hardware threads using message boxes. The access to shared memory is inherently not thread-safe. Therefore, threads must define appropriate critical sections when accessing the shared memory. The data transfer rate of shared memory is much higher

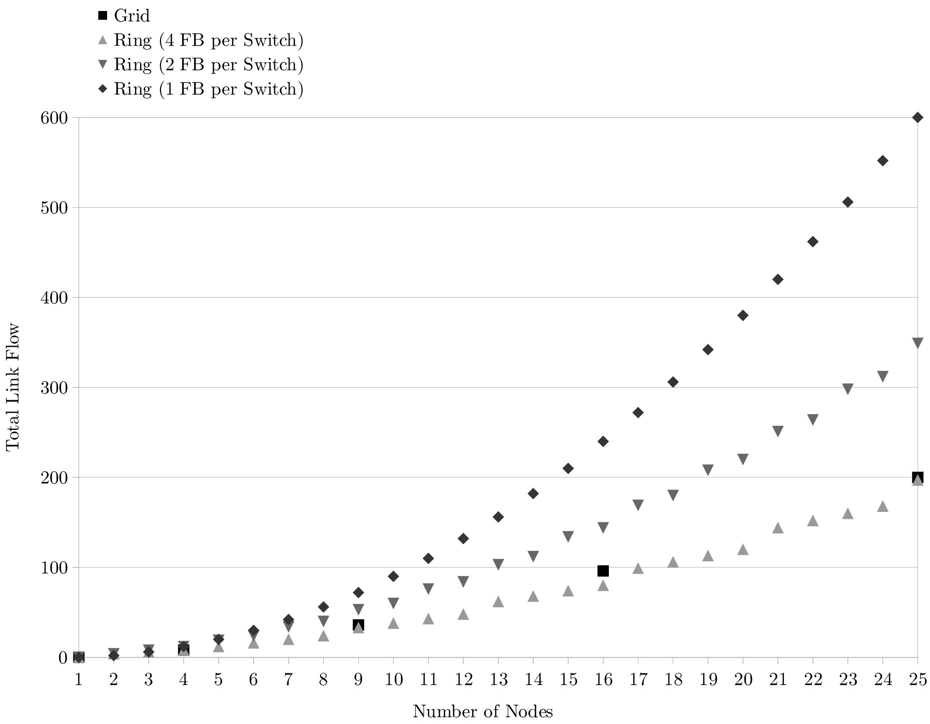

Figure 2.3: Total link flow of the ring and grid topology

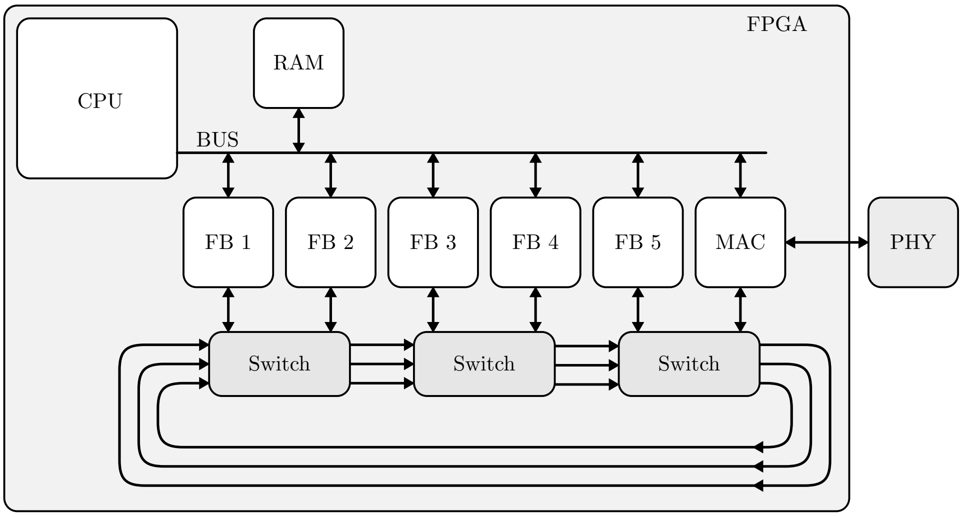

Figure 2.4: An example of the resulting NoC topology: a ring with two functional blocks per switch and three parallel channels between the switches

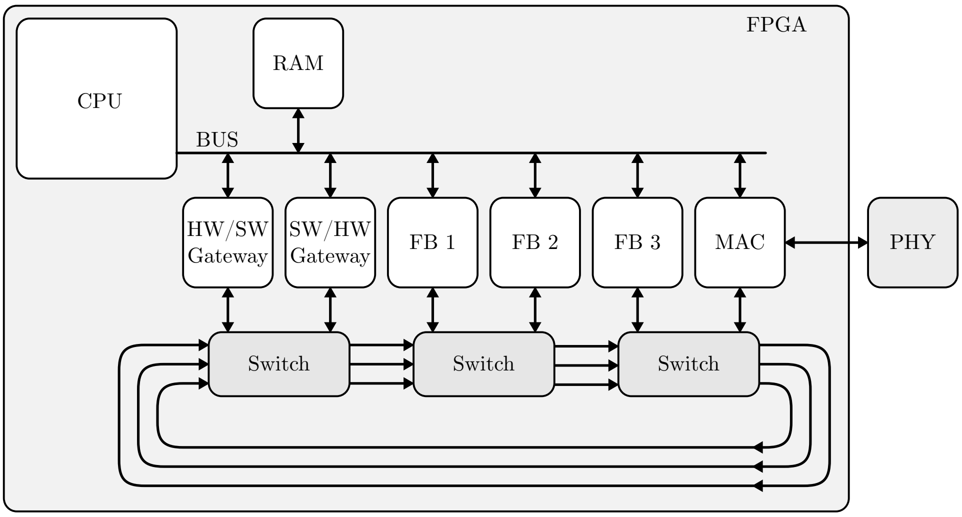

Figure 2.5: Two dedicated hardware threads for a hardware-to-software and a software-to-hardware gateway, respectively

than the data transfer rate of message boxes. However, threads waiting for new data are not notified about a write access of other threads and must therefore poll the shared memory.

2.4.2 Distributed versus Centralized

We can see from figure 2.4 that the hardware threads are now at the same time connected to the NoC and the shared bus. The hardware threads can therefore simultaneously transfer packets to other hardware threads via the NoC and to software threads via the ReconOS interface.

This distributed approach is not optimal in the case where two or more hardware threads send packets to the software at the same time. In this scenario, all sending hardware threads raise interrupts at the CPU for notifying the software thread about the availability of new data.

In order to avoid this problem, we decided to implement a more centralized architecture. In this approach, two hardware threads form a *hardware-to-software-* and a *software-to-hardware gateway*, respectively. Hardware mapped functional blocks that want to transfer packets to a software thread send these packets via the NoC to the hardware-to-software gateway. The gateway is then responsible for transferring the packets to the software. The software-to-hardware gateway works just the same way. Software threads send packets to the software-to-hardware gateway which then redistributes the packets to their destination hardware thread using the NoC. Figure 2.5 visualizes this centralized architecture.

These gateways can aggregate multiple packets that have to cross the hardware/software boundary within a specific time interval and therefore reduce the total number of required interrupts. The higher latency of the packets caused by the additional transfer on the NoC is well compensated by the reduced interrupt handling time.

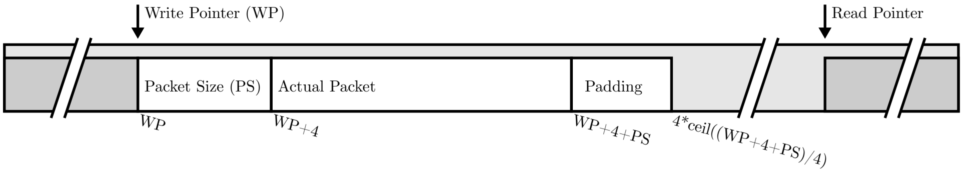

Figure 2.6: The format a packet in the ring buffer

2.4.3 Ring Buffer

The hardware-to-software gateway and the software-to-hardware gateway are very similar in concept. We therefore generalize the gateway's mechanism in order to reduce the documentation effort. Both gateways consist of a *producer* thread and a *consumer*. In the hardware-to-software gateway, the producer is a hardware thread and the consumer is a set of software threads. Conversely, in the software-to-hardware gateway, the producer is a set of software threads and the consumer is a hardware thread.

Both gateways transfer data across the hardware/software boundary by using ring buffers in the shared memory. The producer and the consumer both hold a copy of the read and write pointer of the ring buffer. When a packet is ready to be transferred, the producer first checks whether there is enough free space available in the ring buffer for the packet. If this is the case, the producer then writes the size of the packet and the actual packet to the shared memory. A padding block may be necessary to fill the gap between the end of the packet and the the next word offset. After this write procedure, the producer sets its write pointer to the first offset after the packet. Figure 2.6 illustrates the format of the packets in the ring buffer.

Eventually, the producer decides that it is reasonable to notify the consumer about the packets in the ring buffer. The producer then writes its current write pointer to a message box. Upon receiving a new write pointer via the message box, the consumer knows that the area between the read pointer and the newly received write pointer is filled with valid packets. After the consumer processed all packets in this area, it updates its read pointer with the value of the received write pointer. The consumer then writes its read pointer to another message box, which is read by the producer. When the producer receives the newly read pointer of the consumer, it knows that the corresponding area of the ring buffer can be used again for the transfer of new packets.

The tricky question is now: when should the producer initiate such an exchange of read and write pointers? The answer to this question is a trade-off. On one hand, we want to aggregate as much packets as possible in order to reduce the interrupt load. On the other hand, we need an upper bound for the latency of the packets. We further want to provide a non-blocking write access to the producer. We therefore require that a minimum amount of free space is always available in the ring buffer. Additionally, some packets may be very delay sensitive. We want to support these urgent packets and even allow them to decrease overall performance of the system. We therefore specify the following rules for the pointer exchange:

- The producer starts a timer when it writes a packet to the ring buffer for the first time after sending the write pointer to the consumer. An pointer exchange is initiated when this timer expires. This guarantees that the packets are not blocked in the ring buffer for more than one timer period.

- After the producer has written a packet to the ring buffer, it checks the amount of free space in the ring buffer. If this value is below a specific threshold, the producer initiates a pointer exchange. This enables the producer to continue writing packets to the ring buffer during the pointer exchange procedure.

- Delay sensitive packets have a specific flag set in their packet header. After writing such packets to the ring buffer, the producer immediately initiates a pointer exchange.

Appendix A

Original Task Description

A.1 Introduction

This master thesis is in the context of the EPiCS project. The goal of the EPiCS project is to lay the foundation for engineering the novel class of proprioceptive computing systems. Proprioceptive computing systems collect and maintain information about their state and progress, which enables self-awareness by reasoning about their behaviour, and self- expression by effectively and autonomously adapting their behaviour to changing conditions. Concepts of self-awareness and self-expression are new to the domains of computing and networking; the successful transfer and development of these concepts will help create future heterogeneous and distributed systems capable of efficiently responding to a multitude of requirements with respect to functionality and flexibility, performance, resource usage and costs, reliability and safety, and security.

In this thesis we focus on the networking aspect of EPiCS which we call *EmbedNet*. EPiCS uses the network architecture developed in the ANA project as a basis. The ANA network architecture is a novel architecture that enables flexible, dynamic, and fully autonomous formation of network nodes. In the EPiCS project we develop the ANA architecture further. On the one hand we will develop mechanisms to adapt the functionality provided by the protocol stack at runtime, on the other hand we will develop mechanisms that map the networking functionality dynamically to either hardware or software.

The objective of this Masters Thesis is to refine the hardware architecture and the hardware/software interface of EmbedNet.

A.2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

A.2.1 Objectives

The goal of this Master thesis is to develop a hardware architecture for future dynamic protocol graphs. The developed architecture has to partition networking functionality into individual blocks. The architecture should allow for the dynamic reconfiguration of whole networking blocks as well as for the runtime configuration of communication parameters in the networking blocks. An interconnect between the functional blocks should be provided that allows for the forwarding of packets at line rate. The transmit data path should not interfere with the reception data path.

A.2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

Familiarization

- Study the available literature on ANA, EmbedNet and Reconos [1, 2, 3, 4].
- Setup a FPGA development environment with the Xilinx tools 12.3 (exact) and ModelSim SE version 6.1f (or higher).
- Familiarize yourself with git/github and clone the reconos source code repository [5].
- Install the eCos source code and cross compile toolchain [6].
- Verify your toolchain by running first the sort_demo_thermal and then the pr_demo.
- In collaboration with the advisor, derive a project plan for your master thesis. Allow time for the design, implementation, evaluation, and documentation of your software.

Architecture and hardware design

- Develop a dedicated communication interface between the different hardware networking blocks. Adapt the hardware blocks where needed. The interface should be able to forward data at line rate (1Gbit/s). Transmission and reception direction should be treated independently. The interface should support different priorities of either sending networking blocks or individual packets.
- Develop a configuration interface for the hardware networking blocks. Each hardware block needs to be configured with the information required for forwarding the individual packets to the next protocol. The configuration interface should be able to set, delete, and replace configuration entries. The configuration should be initiated by a software program.
- Develop a data path for sending data between hardware and software networking blocks. Consider using two ring buffers per networking block, one for packets to be transmitted and one for packets to be received. The reader/writer synchronization should be handled properly. An interface should be developed to send and receive packets from software.
- Optional: Adapt the data path to work with the LANA protocol stack [7].
- Optional: Develop a monitoring framework for the measurement of hardware parameter such as temperature, utilisation or power consumption. The framework should collect the data in hardware and make them accessible to a software program.

Implementation

- Determine an appropriate version control system. The EPiCS project is hosted at github. You might want to put your code in the same repository.
- Implement a static version of the dedicated communication interface together with some dummy networking blocks for validation purposes.
- Add the possibility for partial reconfiguration of hardware networking blocks to the communication interface.
- Implement the configuration interface. Depending on the overall EPiCS project status use either the eCos operating system or the Linux operating system (preferred).
- Implement the data path for the hw/sw interface. Depending on the overall EPiCS project status use either the eCos operating system or the Linux operating system (preferred).
- Optional: Implement the interface to LANA.
- Optional: Implement the monitoring framework.

Validation

- Validate the correct operation of your implementation after each implementation step. Use for your evaluation different packet sizes (short, long, even or odd number of bytes, etc.).
- Check the resilience of the implementation, including its configuration interface, to uneducated users.

Evaluation

- Do a performance evaluation of your implementation.
- Optional: Determine the bottlenecks of your implementation.
- Optional: Do a performance comparison between packet forwarding for different combinations of hardware and software networking functional blocks.

Documentation

- Appropriate source code documentation.
- Write a step-by-step how to that describes the compilation of your code, the loading of the code into the hardware and the execution of your code.
- Write a documentation about the design, implementation, validation and evaluation of your work.

A.3 Milestones

- Provide a ”project plan” which identifies the mile stones.
- Two intermediate presentations: Give a presentation of 10 minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.

- Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.

- Any software that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.

- Final report. The final report must contain a summary, the assignment, the time schedule and the Declaration of Originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

A.4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.

- The student is encouraged to write all reports in English; German is accepted as well.

- The core source code will be published under the GNU general public license.

A.5 References

[1] ReconOS: Multithreaded Programming for Reconfigurable Computers: Description of the hardware/software architecture
[2] Reconifgurable Nodes for Future Networks: Description of how we would like to use the hw/sw architecture to build reconfigurable networks
[3] The Autonomic Network Architecture (ANA): Description of the ideas and sw prototype for configurable networks
[4] https://github.com/EPiCS/epics-org/blob/master/deliverables/D3-1_Architecture_And_Tool_Flow/D3-1_Architecture_And_Tool_Flow.pdf
[5] https://github.com/EPiCS/reconos/
[6] webpage: ecos.sourceware.org, a version that is compliant with reconos is in the github repository, cross compilation tools: gcc-4.1.2 glibc-2.3.6
Webpages:
http://www.ana-project.org
http://www.epics-project.eu
http://www.reconos.de