Resource efficient Communication in network-based Reconfigurable on-chip Systems

Dissertation
zum Erlangen des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
in der Wissenschaftsdisziplin
Technische Informatik

eingereicht an der
Mathematisch-naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Philipp Mahr

Potsdam, den 11.06.2012

Betreuer:
Prof. Dr. Christophe Bobda

Gutachter:
Prof. Dr. Christophe Bobda
Prof. Dr. Christian Haubelt
Prof. Dr. Dietmar Tutsch

Prüfungskommission:
Prof. Dr. Ulrike Lucke
Prof. Dr. Christian Haubelt
Prof. Dr. Christophe Bobda
Prof. Dr. Helmut Asche
Prof. Dr. Tiziana Magaria
Prof. Dr. Klaus Rebensburg
Prof. Dr. Torsten Schaub
Prof. Dr. Bettina Schnor
Prof. Dr. Michael Gössel

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Dissertation selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Dissertation ist bisher keiner anderen Fakultät vorgelegt worden. Weiterhin erkläre ich, dass ich bisher kein Promotionsverfahren erfolglos beendet habe und dass eine Aberkennung eines bereits erworbenen Doktorgrades nicht vorliegt.

Potsdam, den 11.06.2012 ....

(Philipp Mahr)

Danksagung

Sowohl für die finanzielle Unterstützung als auch für die Möglichkeit zum Austausch mit anderen Forschern bedanke ich mich bei der Deutsche Forschungsgemeinschaft (DFG), die das AMoC Projektes (Adaptive Multiprocessor-on-Chip) finanzierte.

Meinem Doktorvater Christophe Bobda danke ich für seine Unterstützung und die vielfältigen Entwicklungsmöglichkeiten.

Den weiteren Gutachtern Christian Haubelt, Michael Hübner und Dietmar Tutsch danke ich für die Bereitschaft meine Arbeit zu lesen.

Meinen Kollegen Ali Zarezadeh, Felix Mühlbauer, Kevin Cheng, Lars Middendorf und Robert Hartmann danke ich für die fruchtbare Zusammenarbeit und Hilfestellung bei den vielfältigen Problemen, die im Unialltag auftreten. Weiterhin gilt mein Dank Benjamin Andres, Steffen Christgau, Max Frohberg, Sebastian Fudickar, Martin Gebser, Kerstin Rießelmann, Jens Rohrlack, Christian Spruch und Sebastian Taube für die Unterstützung und Zusammenarbeit.

Christian Haubelt und Tobias Schubert danke ich für die wissenschaftliche Zusammenarbeit während und auch nach ihrer Zeit in Potsdam.

Besonderen Dank möchte ich Jasper Nöllner für die Korrektur und intensive Beschäftigung mit einem ihm fremden Forschungsgebiet aussprechen.

Schlussendlich danke ich meiner Familie, meiner Freundin, meinen Freunden und der Musik für die ununterbrochene Unterstützung.

Kurzzusammenfassung

Die Leistungsfähigkeit rekonfigurierbarer Rechensysteme steigt kontinuierlich und ermöglicht damit die parallele Ausführung von immer mehr und immer größeren Anwendungen. Die Vielfalt an Anwendungen macht es allerdings unmöglich ein optimales Kommunikationsnetzwerk zu entwickeln, welches die Anforderung jeder denkbaren Anwendung berücksichtigt. Die Performanz des rekonfigurierbaren Rechensystems sinkt. Das Kommunikationsnetzwerk ist jedoch nicht der einzige Teil des Systems, der Einfluss auf die Kommunikationsperformanz nimmt. Die Ressourcenverwaltung des Systems beeinflusst durch die Platzierung der Anwendungen die Latenz zwischen Kommunikationspartnern und die Kommunikationslast im Netzwerk. Kommunikationsprotokolle beeinträchtigen die Performanz der Kommunikation durch Daten und Rechenoverhead, die ebenso zu erhöhter Netzwerklast sowie Ressourcenanforderungen führen.

In einem ganzheitlichen Kommunikationsansatz wird nicht nur das Kommunikationsnetzwerk berücksichtigt, sondern außerdem Ressourcenverwaltung, Kommunikationsprotokolle und die anderweitige Verwendung vorhandener, temporär ungenutzter Kommunikationsressourcen. Durch Einbeziehung dieser Aspekte während Entwurfs- und Laufzeit und durch Optimierung unter Berücksichtigung der Kommunikationsanforderungen, wird eine ressourceneneffizientere Kommunikation erreicht. Ausführliche Evaluationen zeigen, dass eine ganzheitliche Betrachtung von Kommunikationsfaktoren, Verbesserungen von Performanz und Flexibilität erzielt.

Abstract

The constantly growing capacity of reconfigurable devices allows simultaneous execution of complex applications on those devices. The mere diversity of applications deems it impossible to design an interconnection network matching the requirements of every possible application perfectly, leading to suboptimal performance in many cases. However, the architecture of the interconnection network is not the only aspect affecting performance of communication. The resource manager places applications on the device and therefore influences latency between communicating partners and overall network load. Communication protocols affect performance by introducing data and processing overhead putting higher load on the network and increasing resource demand.

Approaching communication holistically not only considers the architecture of the interconnect, but communication-aware resource management, communication protocols and resource usage just as well. Incorporation of different parts of a reconfigurable system during design- and runtime and optimizing them with respect to communication demand results in more resource efficient communication. Extensive evaluation shows enhanced performance and flexibility, if communication on reconfigurable devices is regarded in a holistic fashion.

List of abbreviations

ASIP	Application Specific Processor
ASMO	Average Size Module Only
ASP	Answer Set Programming
CFB	Configurable Function Blocks
CLB	Configurable Logic Blocks
CMC	Configurable Memory Controller
CPLD	Complex Programmable Logic Devices
DFT	Discrete Fourier Transformation
DSP	Domain Specific Processor
DyNoC	Dynamic Network-on-Chip
EDF	Earliest Deadline First
FFT	Fast Fourier Transformation
FPGA	Field-Programmable Gate Arrays
FSL	Fast Simplex Link
GPP	General Purpose Processor
LMF	Largest Module First
LMO	Largest Module Only
LTF	Largest Task First
LUT	Look-Up Table
MAC	Multiply-Accumulate
MPI	Message Passing Interface
NoC	Network-on-Chip
PAL	Programmable Array Logics
PE	Processing Elements
PLA	Programmable Logic Arrays
PLD	Programmable Logic Devices
RD	Reconfigurable Device
RCU	Reconfigurable Unit
SMF	Smallest Module First
SMO	Smallest Module Only
SVD	Singular Value Decomposition

Contents

1 Introduction

In 1971 Intel introduced the first commercial microprocessor with 2300 transistors, the Intel 4004. Since this historic release, transistor count on processors increased rapidly and doubled every two years, leading to today's high-end microprocessors like the Intel Xeon E7-8870 [37] with 2600 million transistors. The increasing amount of transistors integrated on one die led to the emergence of several processor classes providing different ways to process applications. Two main means to characterize different processors classes are *flexibility* and *performance* as shown for four common processor types in Figure 1 [33, 9].

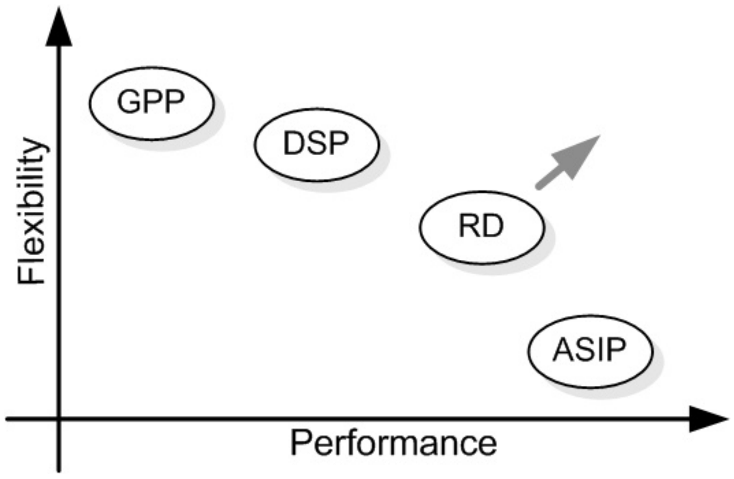

Figure 1: Performance vs. Flexibility of processor classes

General Purpose Processors: General Purpose Processor (GPP) are able to compute any kind of task. The concept of a GPP was presented by John von Neumann [69] in 1945 and became the foundation of modern microprocessors. Computations are executed by a fixed (general purpose) data-path which carries out a stream of instructions to perform calculations sequentially (temporal computing). Because GPPs can execute any kind of algorithm, they trade flexibility for performance.

Domain Specific Processors: Domain Specific Processor (DSP) are tailored to a specific class of algorithms. Like GPPs they execute a stream of instructions but their data-path is optimized to increased performance for common operations of the algorithm class. A common DSP is the digital signal processor which is used for applications involved in image processing, multimedia or telecommunication. The specialization of DSPs increases performance but does not allow the implementation of algorithms other than those for which it was optimized and therefore is less flexible.

Application Specific Processors: Application Specific Processor (ASIP) are even further tailored to specific applications compared to DSPs. The processor is designed for a particular application by implementing the application completely in hardware. No stream of instructions are present as the operations of the application are implemented in hardware directly, allowing for optimized (parallel) execution compared to GPPs and DSPs. While ASIPs do use spatial computing and therefore offer high performance they are only usable for the one application they are designed for, thus having very limited flexibility.

Reconfigurable Devices: A Reconfigurable Device (RD) allows adaption to the application during runtime by modifying its spatial structure. The RD is customized to a particular application containing only the needed operations, therefore reaching performance close to ASIPs. Unlike a static ASIP modifying the spatial structure of the RD by loading new configurations during runtime is similar to loading new software modules onto a GPP and enables RDs having flexibility close to GPPs.

While GPPs offer a high degree of flexibility due to their ability to compute any kind of task, they do not offer much performance in contrast to the other processor classes. Flexibility is achieved by adapting the application to the hardware which subsequently runs through fixed stages like Instruction Fetch, Instruction Decode or Execution. In contrast, ASIPs offer high performance because they are optimized for a particular application and the hardware is adapted to the application. Due to the dynamic of loading new configurations on the RD during runtime and the adaption of the hardware to the application, reconfigurable devices can offer both flexibility and performance. A simplified view at the architecture of a reconfigurable device (see Figure 2) shows computational resources connected to an interconnection network (communication resources). While the computational resources needs to adapt to the applications' computational requirements, the communication resources need to adapt to the applications' communication requirements.

A general goal when designing a network for communication is to design it at minimum costs while still fully satisfying performance requirements [13]. So, in the design of interconnection networks the applications' communication requirements like the number of

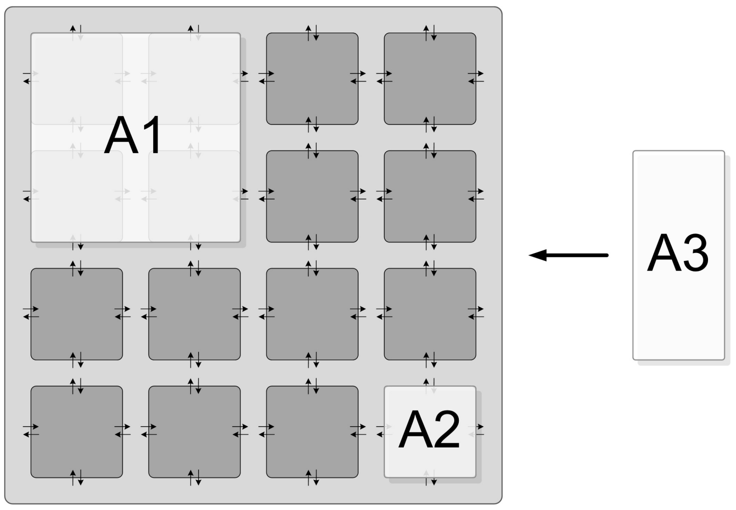

Figure 2: Reconfigurable device with computational resources (dark gray) and communication resources (light gray) with two loaded applications A1 and A2 and a not yet loaded application A3.

communication partners, bandwidth or latency are of major interest in order to design a network which achieves high performance [17]. However, the mere diversity of applications deems it impossible to design an interconnection network matching the requirements of each and every application perfectly hence leading to suboptimal performance from the start.

On RDs efficient communication cannot be achieved when relying solely on the design of the interconnection network architecture. Communication protocols affect performance by introducing data and processing overhead producing higher network load and resource demand, but offering increased flexibility on the other hand. Applications can be positioned freely on the RD. The location were an application is placed on the RD in respect to its communication partners (like peripherals) has influence on the performance of both the interconnection network and the application. Longer communication paths lead to longer latencies and an (overall) higher network load [1]. So, communication at runtime need to be considered in order to achieve efficient data exchange between communication partners. Furthermore, the high resource and area demand of flexible interconnection networks on RDs limits resources available to speed-up computations thus impacting performance of applications.

Approaching communication holistically considers not only the architecture of the interconnect, but also (i) communication-aware resource management, (ii) communication protocols and (iii) resource usage just as well. By incorporating different parts of the reconfigurable system during design- and runtime with respect to communication requirements, a *resource efficient communication* can be achieved.

1.1 Assumption and scientific challenges

The following thesis states the goal of my work.

> *By optimizing resource management and usage of communication protocols and network resources, and by incorporating these into the applications' communication requirements, resource efficient communication can be achieved. It can be shown, that this holistic communication approach leads to higher performance and flexibility of network-based reconfigurable systems compared to approaches focusing solely on the architecture of the interconnection network.*

Confirming this thesis is done in three parts considering different aspects of resource efficient communication. Extensive evaluation will show that performance and flexibility enhancements can be reached with the help of the concepts and techniques developed here.

Communication-aware resource management
Resource management of the RD at runtime is an important aspect of the reconfigurable systems and is similar to the parts of an operating system managing the underlying system resources. On a RD, the resource manager determines when (temporal planning) and where (spatial planning) to execute an application, which directly influences system performance. A simulator of an on-line runtime manager is presented and several optimizations are evaluated with regard to device utilization and communication distance (Chapter 3).

Dynamic Network-on-Chip architecture
It is not possible to design a runtime manager without a deeper understanding of the targeted reconfigurable architecture. In this part, the implementation of a Dynamic

Network-on-Chip is presented. The reconfiguration capabilities of the architecture are evaluated and methods to enhance flexibility and performance by using the available communication (and computational) resources are given. An FPGA-based prototype implementation allows detailed evaluation of the proposed methods (Chapter 4).

Communication protocol
Up until now primarily hardware-based implementations of applications for the used reconfigurable architecture were considered. Finally, the view on the system is broadened by extending the Dynamic Network-on-Chip architecture to support GPPs allowing applications to be constructed from software and/or hardware. For efficient communication between applications running on GPPs a high-level communication protocol is adjusted to the requirements of reconfigurable network-based multiprocessor systems on-chip (Chapter 5).

All these partial aspects show, that enhancements in flexibility and performance can be achieved through a resource efficient communication approach. The above aspects face several challenges to be solved.

Problems and challenges in communication-aware resource management
The dynamic behavior of loading new configurations on the RD during runtime has influence on the devices' free space, utilization, network load and fragmentation. With the overall goals being the maximization of acceptance of applications and the minimization of communication distance, methods for optimizing on-line temporal and spatial planning need to be found and evaluated.

Communication-aware temporal and spatial planning uses methods to approximate the path between sender and receiver on network-based RDs. Depending on the configuration of the device, the concrete communication path from sender to receiver changes during runtime, e. g. due to another application loaded in between sender and receiver. The precise path need to be considered to get information about realistic communication demands and to evaluate the quality of dynamic routing.

Spatial resource planning uses approximation algorithms to find solutions, because the planning problem is NP-hard. In order to evaluate the quality of the approximation algorithm, the optimal solution need to be known for comparison. Due to the complexity of the problem it is not possible to calculate the optimal solution in most cases. The challenge is therefore to find the best possible solution in a feasible amount of time.

Problems and challenges in Dynamic Network-on-Chip architectures
In Dynamic Network-on-Chip architectures, efficient resource usage is of major interest. Configurations covering both computation and network resources result in suboptimal resource usage as network resources remain unused when covered by a configuration. The challenge here is to find methods to utilize the network resources inside a configuration and to achieve an optimal cooperation in terms of computation and communication. These methods have to be implemented in order to evaluate their operability.

Problems and challenges in Communication protocol
For multiprocessor systems, like super computers, many widely accepted communication protocols exist. An adaption to on-chip multiprocessor systems is obvious, as the communication requirements are similar. However, the adaption of communication protocols to on-chip multiprocessor systems is not straight forward as system abilities differ. Compared to multiprocessor systems, resources of on-chip systems are limited. Especially on-chip memory are sparsely available and the performance of on-chip processors is lower compared to the performance of processors in a multiprocessor system. The task at hand is to develop a communication protocol with low memory footprint and computational overhead.

1.2 Organization of thesis

The rest of this thesis is organized as follows. Chapter 2 introduces the abstract resource model for adaptive on-chip multiprocessor systems used in this paper. In order to understand reconfigurability a brief introduction to relevant state-of-the-art reconfigurable devices, interconnection networks and resource managers is given. After this introductory chapter, the following chapters are devoted to resource efficient communication. First of all, in chapter 3 the implementation of an on-line resource manager is presented and methods to enhance performance and flexibility, specifically integrated module selection and relocation of tasks are evaluated. Furthermore, the results obtained by the on-line spatial planner are compared with an optimal (off-line) spatial planner. In chapter 4 the perspective is shifted from management of reconfigurable devices to the actual device itself. A resource efficient reconfigurable router is presented being usable as a network router as well as an additional resource for computation. Furthermore, an adaptive processing element for the calculation of the fast fourier transform algorithm is described. Processing elements of reconfigurable devices are not limited to pure hardware implementations like the processing element for the calculation of the fast fourier transform, but can also be implementations of general purpose processors. A communication protocol based on the message passing interface is presented in chapter 5, allowing flexible communication between programmable processors. Finally chapter 6 concludes this work and gives an outlook on follow-up research.

2 Reconfigurable computing systems

In this chapter the state of the art relevant for this work is given. Beginning with the description of an abstract hardware architecture for adaptive on-chip multiprocessor systems, different aspects of reconfigurable computing systems are introduced. Basics of resource management, reconfigurable logic devices, interconnection networks are covered and corresponding research work by others is presented.

2.1 Abstract hardware architecture of a reconfigurable computing systems

In this section the abstract resource model for a reconfigurable computing systems is described. Such a system needs to be able to (i) adapt to the application and (ii) execute several different applications simultaneously. Thus, the underlying hardware architecture needs to be able to change depending on the computational and communication demands and constraints of the applications. RDs offer the option to adapt the underlying architecture to the applications' needs. Applications or tasks are implemented by modules on a reconfigurable system.

Modules consist of processors (Pr), memory (Mem), local interconnects and/or hardware accelerators (Hw). Each module has access to further global resources like peripherals (I/O) and external memory (Ex. Mem) like RAM, ROM or HDDs. Peripherals make data exchange between modules and the outside world available. UART, USB, Ethernet, WI-FI, Video I/Os, Audio I/Os or Digital Analog Converters are common examples of peripherals. Global resources are located around the RD to allow separate data storage and/or access to peripherals from independent modules.

Figure 3 shows several modules m_i on a RD. External memory and peripherals are arranged at the borders of the RD. Modules are interconnected using a global interconnection network and have access to all global resources. Note that it is possible to include I/O (I/O Ctrlr.) or memory controllers (Mem. Ctrlr.) inside a module to set up privileged access. In this case these resources are not available to other modules.

This abstract resource model is regarded as the base model of the reconfigurable computing architecture used in this thesis. The model is refined in following chapters and an actual architecture is presented.

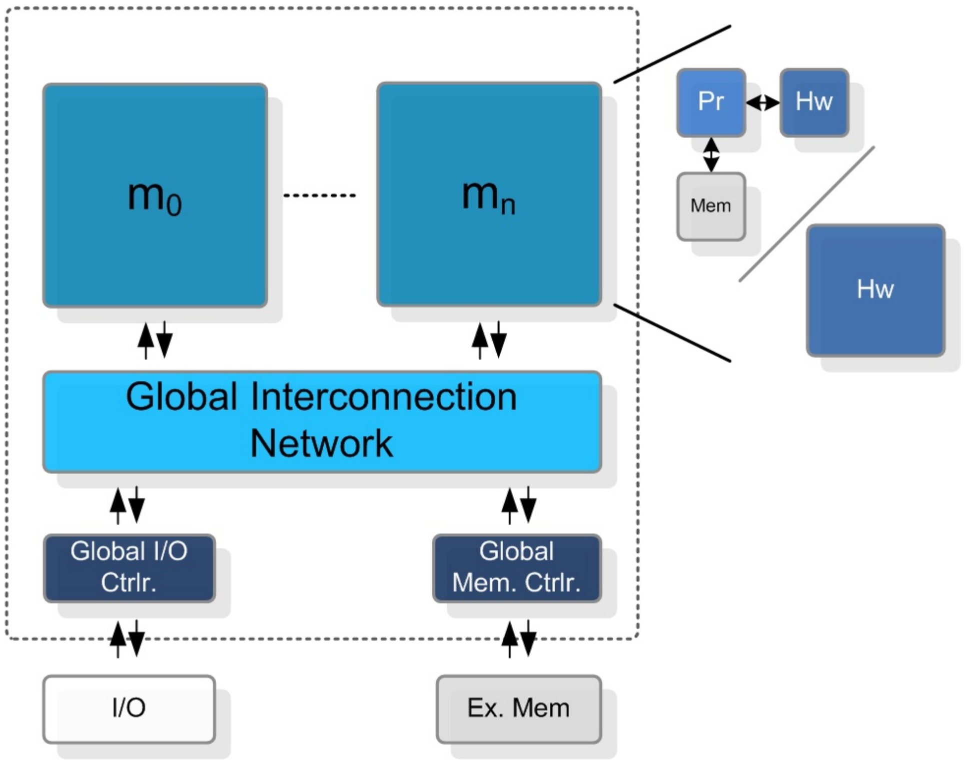

Figure 3: Hardware infrastructure of a reconfigurable computing system with $n+1$ modules

The following section gives an overview on reconfigurable logic devices in order to understand how such devices enforce reconfiguration on hardware level and how different approaches used by different reconfigurable devices deal with the involved issues.

2.2 Reconfigurable logic devices

The concept of reconfigurable computing was introduced by Gerald Estrin in the late 50s. The Fixed Plus Variable Structure Computer [24, 25] was the practical realization of this concept. It used a *fixed part* holding a general purpose processor, a *variable part* and a *supervisory control* part to coordinate operations between the fixed and the variable part. Configuration was done manually by replacing basic building blocks on the variable part. Two different types of building blocks were available: one for signal inversion, amplification or high-speed storage and the other one for combinatorics. Up to 36 building blocks could be inserted into the variable part and connection between them was done by a wiring harness allowing the configuration of the Fixed Plus Variable Structure Computer. The technology at that time made the use of the Fixed Plus Variable Structure Computer difficult. A lot of effort was required to implement applications in the first place and to reconfigure the device manually later [9].

Over the last two decades, progress in the field of reconfigurable devices was amazing and reconfigurable computing is widely used. Numerous workshops and conferences deal with this topic [9]. Basically, reconfigurable logic devices belong in one of two classes — *fine-grained* and *coarse-grained* devices [33, 30].

Fine-grained reconfigurable devices like Field-Programmable Gate Arrays (FPGA) use single-bit Configurable Logic Blocks (CLB). Coarse-grain devices provide reconfiguration on function level using word-level Configurable Function Blocks (CFB). While fine-grain reconfigurable hardware has the benefit of high flexibility, coarse-grain reconfigurable hardware trades-off some flexibility for a potentially higher degree of optimization in terms of area and power. Coarse-grain architectures need significantly less time to perform reconfigurations and the amount of reconfiguration data is much lower compared to fine-grained architectures [33, 68].

2.2.1 From simple PLDs to FPGAs

Programmable Logic Devices (PLD) like Programmable Logic Arrays (PLA)s, Programmable Array Logics (PAL)s, Complex Programmable Logic Devices (CPLD)s and FPGAs are commonly used devices when a flexible hardware structure is asked for [9]. All these devices belong to the fine-grained RD class.

PAL/PLA PALs and PLAs are the simplest programmable devices and are used to implement combinatorial logic circuits. They can implement any boolean functions in a disjunctive form by connecting an AND-plane to an OR-plane (see figure 4).

PLAs were introduced by Texas Instruments in 1970 [5] and allow programming of both the AND- and OR-plane. In contrast, PALs (introduced in 1978 by Monolithic Memories) only feature programability of the AND-plane with the OR-plane being fixed, so making PALs a subclass of PLAs [9]. Both PLAs and PALs use fuses to cut connections according to the desired boolean function. To program the device once (and for all), fuses are blown after fabrication by using high currents (antifuse technology). The main drawback of PALs

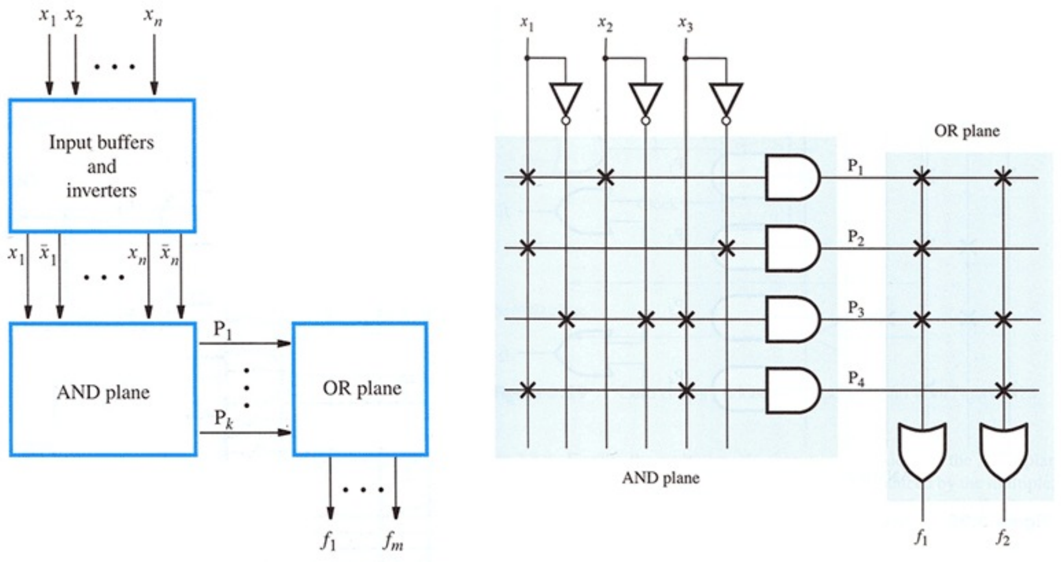

Figure 4: Programmable Logic Array [38]

and PLAs is their limited capacity due to the quick growth of the AND- and OR-planes, depending on the amount of inputs to the planes.

CPLD PALs and PLAs are only available in small sizes, equivalent to a few hundred gates. In order to create bigger circuits CPLDs can be used. CPLDs extend the concept of programmable AND/OR-planes by using additional I/O blocks and a programmable interconnect.

Figure 5 shows the structure of a CPLD using PAL-like blocks with additional registers, a programmable interconnect and I/O blocks. The registers, PAL-like blocks and I/Os can be interconnected freely. Areas, where CPLDs find common usage are power-up sequencing, system configuration, I/O expansion or interface bridging (glue logic). Many CPLDs have a non-volatile configuration memory, thus the CPLD can start processing immediately on start-up. However, these devices are still too small for applications requiring large gate counts.

FPGA FPGAs were introduced in 1985 by Xilinx and can be used to implement applications requiring large gate counts. Similar to CPLDs FPGAs have I/O blocks and a programmable interconnect, but use CLBs [1] instead of PAL-like blocks (see Figure 6). The

[1]CLB is a term used by Xilinx

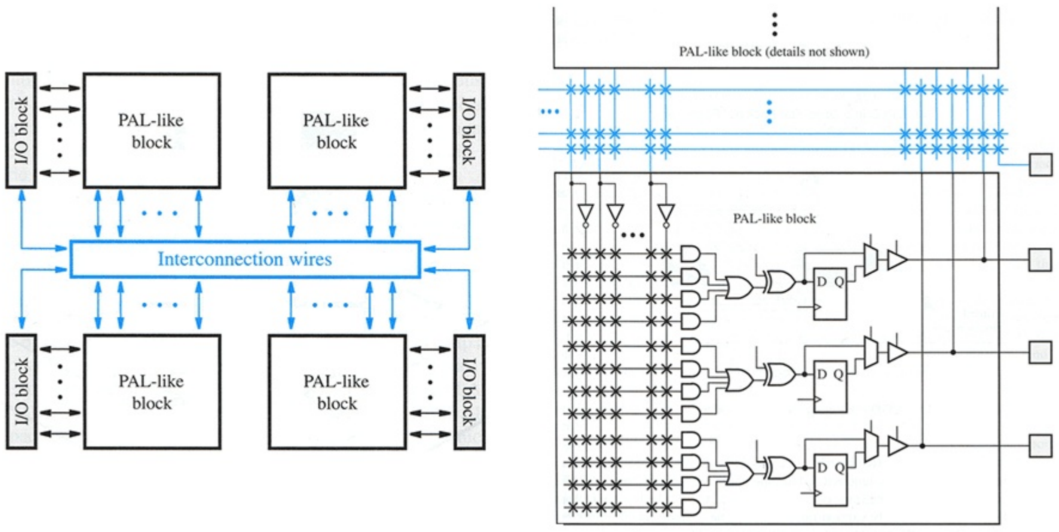

Figure 5: CPLD structure [38]

configurable logic blocks consists of a set of logic cells which typically consists of a Look-Up Table (LUT), a one-bit register and a full adder. In contrast to CPLDs the programmable interconnect of FPGAs is of finer granularity connecting individual logic blocks instead of the large PAL-like blocks. There are different process technologies used for manufacturing FPGAs resulting in different types of programmability: one-time programmability, in-system programmability, and reprogrammability. FPGAs using SRAM-based process technology offer in-system reprogrammability, an important ability in order to perform (partial) dynamic reconfiguration.

State-of-the-art FPGAs furthermore have specialized functional blocks like memories, digital signal processing elements, fast I/Os, microprocessors or on-chip memory controllers. These blocks normally provide word-level functions like 32-bit wide on-chip memory or 25×18 multipliers [75].

Large FPGAs, like the Altera Stratix-IV use more than 2 billion transistors [45] making them suitable to implement complex systems on chip.

2.2.2 Coarse-grained reconfigurable devices

With FPGAs any kind of function can be implemented. This is possible because of the fine-granularity of the logic blocks and the flexible programmable interconnect which connects these blocks. The interconnect is also one of the main limitations of FPGA per-

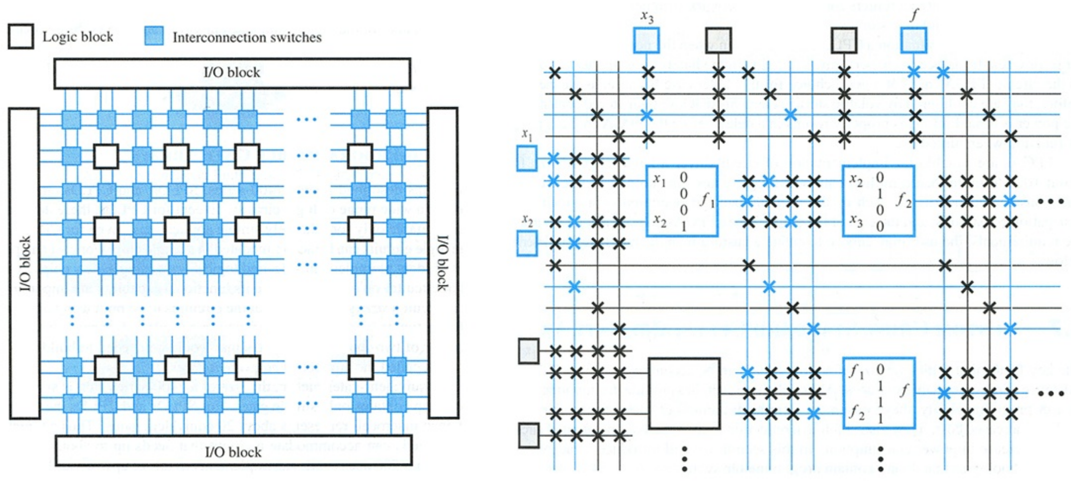

Figure 6: FPGA structure [38]

formance, as it uses up to 90 % of the chip area [43]. Coarse-grained reconfigurable devices use word-level CFBs and therefore need less interconnections. CFBs are able to execute few operations like multiplications, additions or subtractions and there interconnection is realized with switching matrices or dedicated buses [9].

Pact XPP-III The Pact XPP-III [54] architecture is an example of a commercial coarse-grained reconfigurable architecture (Figure 7).

The XPP-III (eXtreme Processing Platform) core consists of an array of different processing array elements (PAEs) and a communication network for data and events. Function PAEs (16-bit general purpose processor kernels) are used to execute control-flow dominated irregular program code, whereas regular streaming algorithms like matrix transformation or FIR-filters are executed on the dataflow array of the architecture using ALU- and RAM/IO-PAEs. ALU-PAEs contain a configurable unit for basic arithmetic and logical operations. RAM-PAEs are very similar to the ALU-PAEs, but replace the arithmetic logic unit with a dual-ported SRAM and an I/O interface. Horizontal and vertical routing connections for point-to-point connections between PAEs are used with different data-width for data and events. Besides the horizontal and vertical connections an additional configuration network exists, which allows runtime reconfiguration of the PAEs. Configuration words are send to its correct row and column addresses, where they are stored in the configuration register of a PAE. When configuring only one PAE, one

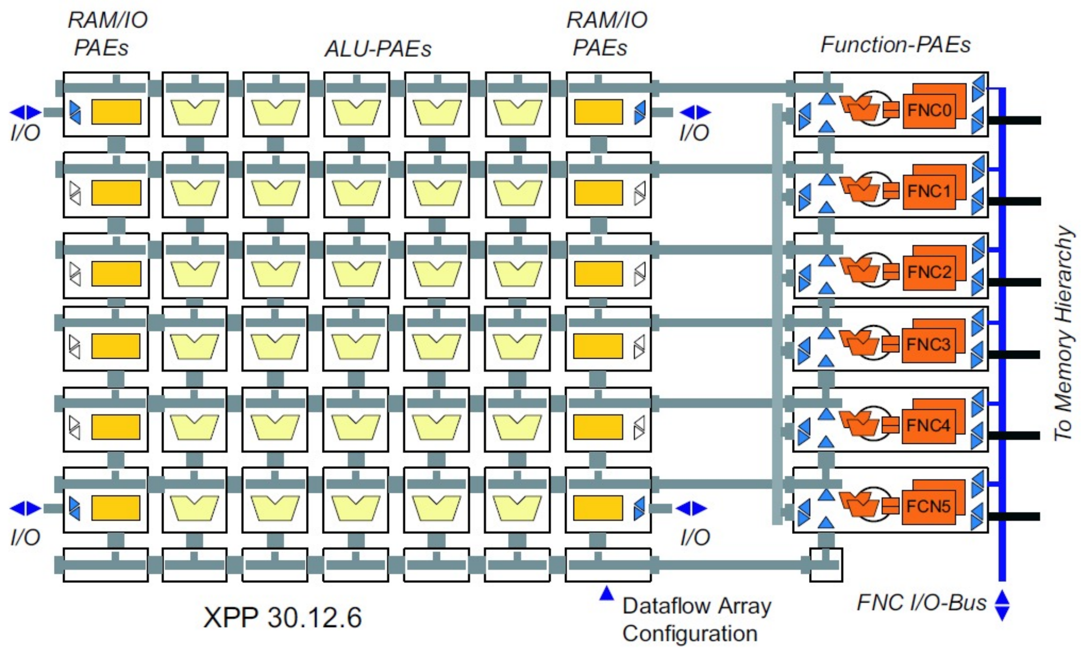

Figure 7: XPP-III core structure [54]

configuration word is needed, but with the configuration of many PAEs a larger configuration word is required. Thus, the configuration time is equivalent to the amount of configuration words [53].

Many more coarse-grained reconfigurable architectures exist (see for instance [67, 49, 62, 22, 44, 14]). In [33] an overview of several architectures can be found.

2.2.3 Reconfigurability

Some of the presented reconfigurable devices, especially SRAM-based FPGAs and coarse-grain architectures like the XPP-III, allow the implementation of tasks, like small video filters or complex video decoders, during runtime. The ability to reconfigure the chip during runtime enables the design of dynamically reconfigurable hardware systems that can adapt themselves to the current set of tasks for better performance. Further benefits of this capability are a reduced power consumption and a more efficient use of the available board space [72]. Tasks can be implemented using the whole RD (full reconfiguration)or only a part of the device (partial reconfiguration).

As an example for partial reconfiguration, consider an automotive infotainment system providing numerous services like audio and/or visual entertainment, video games, automotive navigation, location-based services or internet access. These tasks vary in computational and communication demand. Using a partial reconfigurable system allows execution of such tasks with high performance, while maintaining flexibility by implementing several different tasks in parallel. The system automatically adapts to the users' needs by reconfiguring the underlying hardware device with corresponding implementations of tasks [29].

In a partially reconfigurable device an Reconfigurable Unit (RCU) is the smallest fraction (or frame) available to be (re)configured. In the XPP-III RCUs are the ALU-PAEs and RAM-PAEs. This makes the XPP-III a heterogeneous RD consisting of different types of RCUs. Homogeneous devices on the other hand feature structurally identical RCUs. In FPGAs the size of a single RCU differs, depending on device family and size. In the Xilinx Virtex family FPGAs, a group of configurable logic blocks forms an RCU. The Virtex-2 FPGAs for instance allow partial reconfiguration of an array of 8 by 8 CLBs in XC2V40 devices and 112 by 104 CLBs in the XC2V8000. Meanwhile the newer Virtex-4 can be configured in 1 by 16 CLBs independent of the device's size [42].

In order to manage reconfigurations of the underlying resources a runtime resource manager is needed. Steiger et al. [64, 63] consider resource management the centerpiece of a reconfigurable operating system.

2.3 Resource management of reconfigurable devices

With the ability to execute several tasks simultaneously it is important to determine where the task should be placed on the reconfigurable device. This operation is called *placement* or allocation of resources. When considering timing constraints due to deadlines, either soft or hard, for the tasks has to be carried out in order to calculate the moment the tasks' execution is started *temporal planning*. However, the time at which tasks arrive is not known a priori. In Figure 8 the general architecture of a reconfigurable computing platform is shown.

The resource manager is needed to administrate the (re-)configurations of the RD according to incoming tasks and their requirements and consists of three parts: the *temporal planner*, the *placer*, and the *loader* implementing the task on the device.

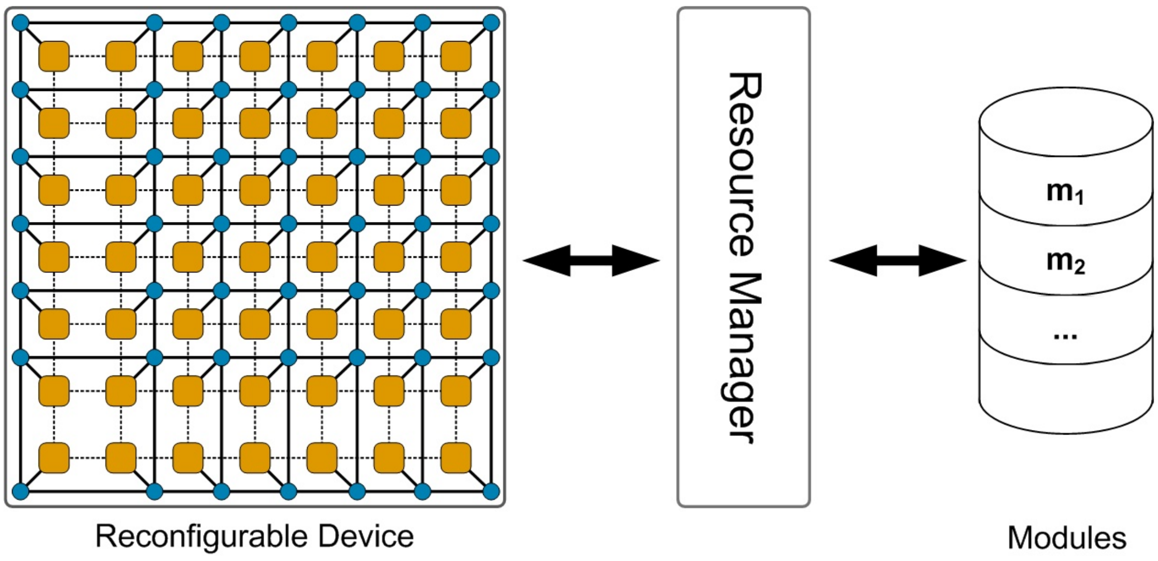

Figure 8: Architecture of reconfigurable computing platform

Tasks are implemented by pre-synthesized *modules* which are the actual hardware situated on the RD. A start time can only be assigned to a task, if its module can be placed on the device and the task can be executed prior to its deadline. Therefore, a strong nexus between temporal planning and placement of tasks exists. Together, both placement and temporal planning steps are called *scheduling* in the course of this thesis corresponding to scheduling in operating systems. Application placement is an important aspect of scheduling in certain multiprocessor systems, like large-scale, shared-memory non-uniform memory access multiprocessors [12]. Thus, an interrelationship between scheduling on multiprocessor systems and reconfigurable devices exist. Yet, in multiprocessor systems tasks are often executed on several different resources (processors), while in reconfigurable devices all tasks share the same adaptable surface.

Scheduling can be done on- or off-line. On-line scheduling handles the tasks arriving one-by-one and the schedule is calculated based on the list of known tasks one after another during runtime. Only a subset of the whole task set is known which might result in suboptimal scheduling decisions. Off-line scheduling on the other hand considers the whole set of tasks and is therefore able to find optimal solutions. However, the off-line placement problem is already NP-hard.

2.3.1 Placement

Most publications on managing the resources of RDs focus on temporal planning and/or placement. These papers can be divided into off- and on-line methods targeting homogeneous or heterogeneous reconfigurable devices. For the computation of task placements basically two steps are performed: (i) managing allocatable free space and (ii) finding a viable placement in this available space.

Bazargan et al. [6] take credit for being one of the first groups of authors to introduce on-line placement algorithms for two-dimensional homogeneous reconfigurable devices. Their algorithms partitions the free space and maintains a set of maximally free rectangles leading to a quadratic size of the set of free rectangles. A rectangle is chosen from the set of free rectangles by using a cost function considering the area of the free rectangles. The selected rectangle is then split into two smaller rectangles of free space which are added to the list of free rectangles. Alternatively, a method for partitioning the free-space into only $O(n)$ rectangles using heuristics is proposed which in comparison gives worse results in terms of task rejections, if considering the same task set. Steiger et al. [64] improved the Bazargan's $O(n)$ partitioner by delaying the split decision of the resulting free rectangles after a module has been placed. Handa and Vemuri [32] showed that free space in the form of maximal empty rectangles can be managed more efficiently by using the staircase data structure.

In [3] an algorithm considering routing-conscious placement is presented, which manages the allocatable space in $O(n \log n)$ using plane sweep methods from computational geometry.The decision where to optimally place a module in the allocatable free space is calculated using weighted communication costs instead of computing the size of the free rectangles. Another approach to placement was presented by Eiche et al. [23]. By using a discrete hopfield neuronal network, an on-line placer for heterogeneous devices was implemented. The neural network outperforms the older SUF fit placement algorithm presented in [40].

When placing modules on a reconfigurable device the available free space becomes fragmented as tasks finish, which can lead to a higher rejection rate of tasks, even though the total amount of allocatable space is higher then the spatial requirements of an incoming task. [26] presents a defragmentation approach for one-dimensional heterogeneous reconfigurable devices in a no-break fashion. By copying one module at a time and then relocating the running computation to the copied module. Results highlighted that the presented approach leads to allocatable space up to 50 % larger. In [61] Tabero et al. present a metric to estimate the degree of fragmentation of a reconfigurable device by analyzing the free space. Heuristics to decide when and how to perform defragmentation of the device are presented. In [66] this work is extended and the metric is used to implement a fragmentation-based heuristic to find a viable location in the available free space.

Danne and Stühmeier [19] introduced an off-line placement algorithm considering geometrical task variants. Tasks are modeled as three dimensional boxes constituted by their width, height and execution time. Modified heuristic methods from floorplanning are applied to select the task variants leading to better solutions for the placement problem. Belaid et al. [7] presented an off-line placer for heterogeneous devices formulating the placement problem as a constrained optimization problem taking task preemption and reconfiguration overhead into regard.

2.3.2 Temporal planning

With execution time and deadlines of tasks, temporal planning becomes a necessity in order to fulfill timing requirements. In [64] different strategies for temporal planning are explored. Normalized planning considering both spatial and temporal aspects of tasks are combined with weighting coefficients and the optimal values of these coefficients are determined. Overall, best results are obtained when considering early deadlines first. Also a non-linear combination of deadline and task size had been considered, but resulted in worse outcomes. Danne et al. [18] considered off-line temporal planning of periodic real-time tasks and adapted the global Earliest Deadline First (EDF) approach used in real-time operating systems to homogeneous reconfigurable hardware devices. Two preemptive algorithms, EDF-First-k-Fit and EDF-Next-Fit, are presented and the influence of reconfiguration overhead was evaluated. Results show that reconfiguration times of smaller scale than task computation time causes only minor losses in performance.

2.4 Interconnection networks and communication protocols of reconfigurable devices

With placement and temporal planning of tasks at runtime it is not known a priori where and when a task is placed. It is important, that ongoing interconnections e. g. between modules of tasks and peripherals or external memories are kept and new interconnections are established. The communication infrastructure and protocols of RDs therefore have to be adaptive.

In [9] communication approaches were categorized depending on the way the communication is realized. Direct communication, communication over third party, bus-based communication, circuit switching and Network-on-Chip (NoC)-based communication are all feasible communication schemes for RDs. However, bus-based communication and NoCs are the dominant schemes in research communities [8]. While buses offer low latency, they do not scale well and are therefore a good solution for a limited number of modules only. NoCs provide far better scalability and are regarded the most viable solution for large chips using many modules, but introduce high and variable latencies [59]. Next, the implementation of a bus-bused interconnection network and NoC architectures is presented.

2.4.1 Bus-based interconnection network

The RMBoC (Reconfigurable Multiple Bus-on-Chip) communication architecture targeting FPGAs is presented in [2]. Figure 9 shows the simplified architecture consisting of several switches locally attached to modules. Multiple segmented buses are used to connect the switches and allow each module to dynamically access the bus and establish communication with other modules (1-D circuit switching).

This slot-based communication approach is closely related to the reconfiguration capabilities of the Xilinx Virtex-2 architecture which allows 1-dimensional partial reconfiguration [56]. More recent FPGAs like Virtex-4 FPGAs allow tile-based partial reconfiguration and are therefore better suited for the implementation of 2-D communication approaches like with NoCs.

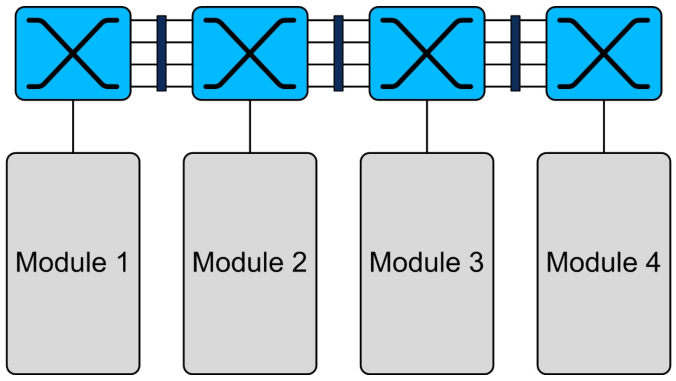

Figure 9: RMBoC architecture

2.4.2 Networks-on-Chip

Several reconfigurable architectures using Networks-on-Chip as the main communication paradigm exist [55, 20, 35, 39, 10, 70]. A NoC is basically constructed from a set of routers relaying data from source to destination.

Devaux et al. [20] implemented a fat-tree NoC using dynamically reconfigurable routers called R2NoC. Each router contains a partial reconfigurable region which implements the dynamic communication links. Depending on the configuration of the reconfigurable region, the in and outputs are (statically) connected and a dedicated connection is established between source and destination modules using circuit switching.

Yet another NoC relying on circuit switching, is the PNoC [35]. In this design, each router has its own subnet. A collection of processing elements in each of those subnets allows frequently communicating tasks implemented by modules to be placed within the same subnet. The routers are implemented in a static manner and only their routing tables are updated as modules are removed or added.

The configurable Network-on-Chip CoNoChi [55] uses virtual cut-through switching of packets. Routers have four equal full-duplex links connecting to the modules at the upper, lower, left and right side of the router. The RD is splitted into a fixed number of tiles which can be of one of four types. Figure 10 shows an exemplary configuration and the four tile types. The CoNoChi architecture makes placement of modules with different sizes possible. The communication infrastructure dynamically adapts and unused routers are deleted to decrease communication latency and to reuse computational and communication resources previously occupied by routers. In contrast, the router resources of R2NoC and PNoC are static and cannot be removed. Thus, CoNoChi offers a higher degree of modularity [56].

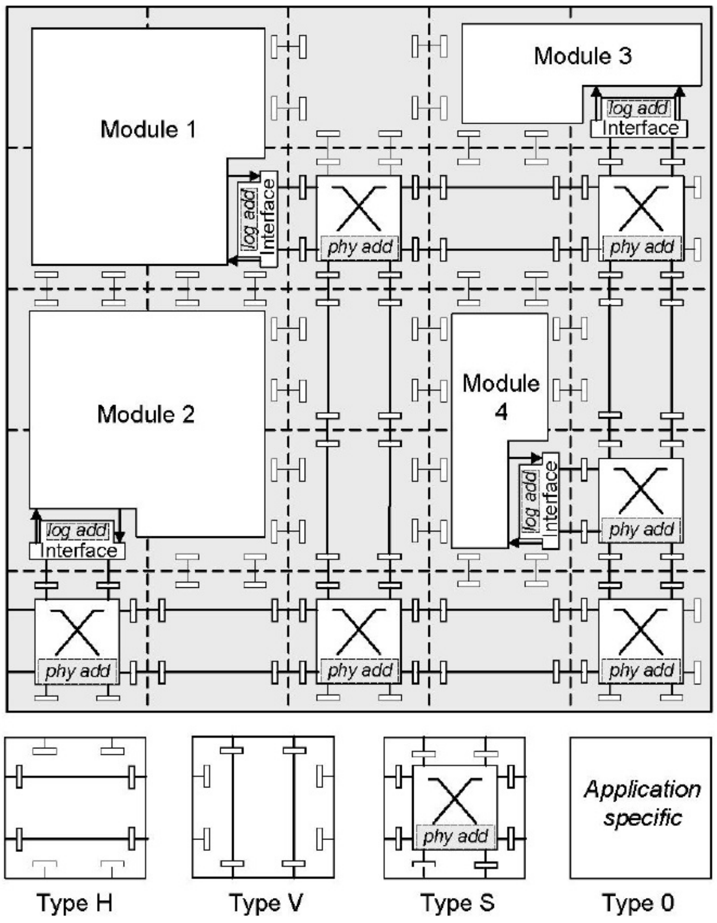

Figure 10: CoNoChi architecture [56]

The Dynamic Network-on-Chip (DyNoC) architecture was presented by Bobda et al. in [10, 11]. The basic state without any modules placed appears and behaves like a normal network-on-chip consisting of Processing Elements (PE)s and routers. Processing elements access the network via corresponding routers. In contrast to a normal NoC, the PEs of the DyNoC can also communicate with their nearest neighbors using direct links. This allows the aggregation of several PEs creating a rectangular module to compute a complex task. PEs inside a module do not need to use the communication resources of the network in between for communication, but use their direct links instead. The advantage of this hybrid communication scheme is a gain in flexibility. Figure 11 shows the DyNoC system architecture, whereas four modules from three different tasks are covering several PEs and routers.

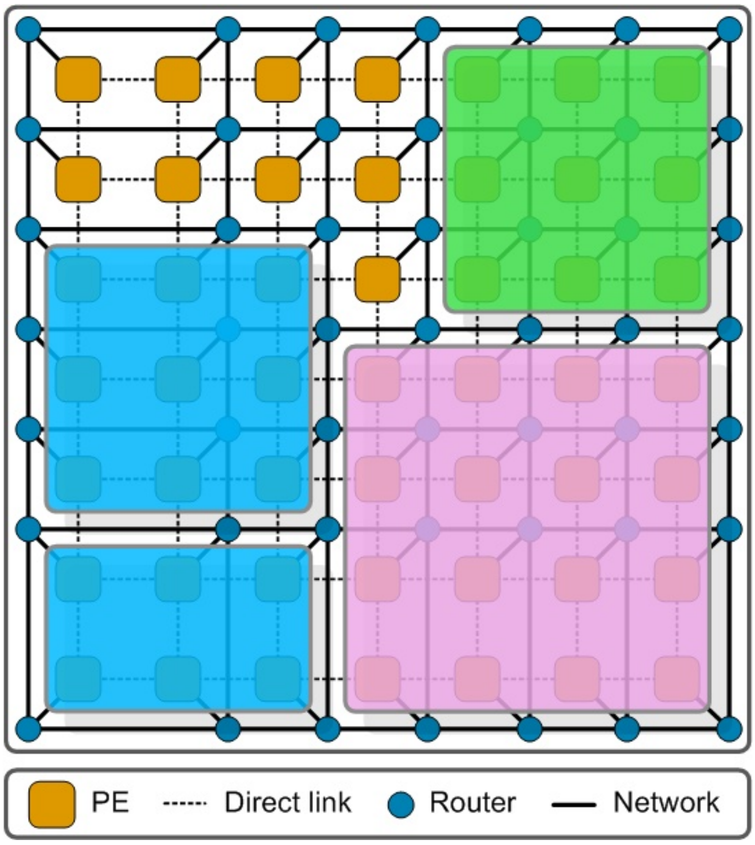

Figure 11: DyNoC system architecture

2.4.3 Communication protocols

Closely related to the architecture of the interconnection network are communication protocols, which define the formats and rules for exchanging messages over the network. Protocols are implemented in hardware and/or software and cover one or more layers of abstraction. Different layers provide different services, like routing, flow control or synchronized communication. Research in this area mainly differs in its focus on particular layers.

The network layer S-XY dynamic routing protocol [11] for the DyNoC adapts routes between communication partners in order to surround obstructing modules in the path of the packet. S-XY is implemented in the router of the DyNoC and operates in three different modes. In normal XY mode (N-XY), XY routing is applied. Packets are first send first send horizontally to the correct X-coordinate and then vertically to the Y-coordinate. Surround horizontal mode (SH-XY) is entered when an obstacle left or right of the packet needs to be surrounded. The last available mode is surround vertical (SV-XY) and is executed when an obstacle to the upper or lower direction is detected. Because modules are surrounded by a ring of router in a DyNoC it could be shown that with a very high probability S-XY routing is deadlock free by proving that there is always a path from the source of the packet to its destination and that the packet will reach its destination after a fixed number of steps [10]. Available work on high layer protocols often provides communication interfaces for applications [60, 46, 28]. Different concepts like synchronous, asynchronous or buffered communication, depending on the underlying architecture of the computational and communication resources, are offered and can be directly accessed in the applications' source code. Some projects consider high level protocol implementations in both software and hardware to realize uniform communication between hardware and software processing modules [71].

2.5 Discussion

As seen in the previous sections, published literature on reconfigurable systems is vast and reaches into multiple research fields. An important field of research is communication, especially as communication is a main factor limiting cost, area, power and performance of systems [17].

Many publications on communication in reconfigurable system focus on the architecture of the interconnection network and aim at developing an efficient communication infrastructure. The network resources of the RD need to provide a high degree of flexibility, in order to handle the changing requirements due to dynamic partial reconfigurations [55, 35, 50]. NoCs are most prominently considered as the interconnection network for reconfigurable devices and certain network aspects like quality of service [27] or buffer sizes [16, 51, 4] are taken into account during design. However, in many cases resource usage of the interconnection network is not optimal because communication resources do not adapt to the applications' communication and/or computational resource demand

[35, 20, 56, 11]. Therefore, methods for efficient resource usage have to be considered in the design of interconnection networks.

Protocols managing the communication on RDs generally target specific architectures, the S-XY routing algorithm of [11] is directly implemented in the hardware of the router, for example. Higher layer RD communication protocols enable flexible communication between applications. Most high layer protocols for on-chip systems can not be adapted to the applications' needs [60, 46, 28, 71] and therefore have a higher memory demand and reduced performance in comparison to adaptable protocols. Hence, flexible and adaptable high-level communication protocols have to be considered in reconfigurable systems.

Only few publications on communication consider resource management of reconfigurable devices during runtime. Most works on resource management focus on optimizing device utilization [64, 23, 26, 61, 18] of the RD, but do not consider communication as a performance factor. Communication-aware on-line temporal and spatial planning offer possibilities to improve performance, e. g. by reducing the network load and path length due to communication-aware scheduling [3, 65, 41]. However, this methods have to be further investigated and evaluated.

In this work a holistic approach on applications' communication with reconfigurable systems during design time and runtime is presented, which increases performance and flexibility through resource efficient management of communication demands.

2.6 Chapter conclusion

This chapter gave an overview of research aspects in the field of reconfigurable computing systems. An abstract hardware architecture of a reconfigurable multiprocessor systems was introduces as the basic resource model used in this work. Several actual reconfigurable devices with different complexities, performance characteristics, granularity and reconfiguration capabilities were described. A brief overview of state-of-the-art interconnection networks and communication protocols for reconfigurable devices has been given. Finally, a resource manager for the management of reconfigurable devices was introduced with the main job being temporal planning and placement of tasks.

The placement of tasks directly influences the volume of communication traffic, so task scheduling should be communication aware to efficiently use the communication resources of the reconfigurable device. Shorter communication path lead to an overall higher system performance, as latency between sender and receiver as well as the network load are reduced. Yet, most scheduling algorithms consider the calculation of viable allocations only and do not comprise finding an optimal placement in regard to communication.

The next chapter presents methods to improve performance of a communication-aware on-line scheduling algorithm. Furthermore, a communication-aware off-line placement algorithm using declarative problem solving is introduced and compared to the on-line variant.

3 Scheduling on NoC-based reconfigurable architectures

This chapter describes communication-aware scheduling of tasks on reconfigurable devices using a NoC-based communication infrastructure. While this paper focuses on the communication between tasks and/or I/Os at the devices' border, scheduling on RDs has not only to provide minimized communication distance as an optimization goal, but aims for maximal device utilization, as well. The approach of this work is to determine time and area, i. e. when and where to place a task on the RD in a fashion to meet the following objectives:

1. The amount of placed tasks out of a task set is maximized and in turn minimizes rejection rates.
2. Communication distances (path length) between communication partners are shortened to a minimum.

The first objective ensures, that the maximum number of tasks out of a task set is executed, while the second objective minimizes path length between sender and receiver. Both goals are strongly related to placement as well as temporal planning. The research presented in this chapter firstly extends a basic on-line scheduler to increase the amount of placed tasks and to lower communication distances between tasks and secondly compares the results of the on-line placer with the optimal results calculated by an off-line placer. Rejection rate of a task set and path length (hop count) are the main metrics used for evaluation. Placement and temporal planning on reconfigurable devices are dependent on the architecture of the device and the used task model. Knowledge on the communication and configuration capabilities of the device is crucial: The size of the reconfigurable regions, as well as the time needed to reconfigure a region and the interconnection network have to be considered to fully exploit the capabilities of the reconfigurable device by a scheduling algorithm. For the thesis at hand, the DyNoC architecture for the dynamic interconnection of reconfigurable modules in a mesh network was used (see Chapter 2.4.2).

Tasks are built up out of one or more components communicating with each other or with I/Os at the devices' edge. An exemplary task is the MPEG-4 part 10 video compression algorithm. It performs several consecutive steps like discrete cosine transformation (DCT), entropy coding or quantization. Each of these steps is dependent on at least one other step or an external data source or sink. The steps are modeled as components of the MPEG-4 task with intra-task dependencies.

3.1 Basic scheduler

This section is an extended version of [MCHB11]. Tasks have a set of components each consisting of at least one implementation in the form of a rectangular module covering a fixed amount of reconfigurable units. The relations between tasks, components and modules are defined as follows:

A task consists of at least one component.

Definition 1 (Task-Component-Relation) *For each task $t_i \in T$, a non empty set C_i of components $c_j \in C_i$ exist.*

Each component has at least one input and output communication link.

Definition 2 (Component-Component-Relation) *For each component $c_j \in C_i$, a set of communication partners or points $cp_k \in CP_j$ exist.*

Each communication partner or point $cp_k \in CP_j$ is associated with the communication requirements average bandwidth ba_k, the communication distance cd_k and the attribute sr_k which marks the communication partner as a sender or receiver. Besides components, also memories or I/Os positioned around the RD can be communication partners, which are called interfaces.

Definition 3 (Component-Module-Relation) *For each component $c_j \in C_i$, a non empty set M_j of modules $m_k \in M_j$ exist.*

With each module $m_k \in M_j$ of a component $c_j \in C_i$ of a task t_i a width w_k and height h_k and a worst-case execution time e_{jk} is associated. Thus, when considering several modules differing in area requirements and execution time, the selection of a modules need to be made prior to placement and temporal planning.

Definition 4 (Feasible Module Selection) *A feasible module selection assigns a module $m_k \in M_j$ to each component $c_j \in C_i$ of task t_i, so that a feasible schedule exist.*

A feasible schedule has to consider both spatial and temporal aspects meaning a placement on the device as well as a start time (temporal planning) for a task have to be found.

Definition 5 (Feasible Schedule) *A feasible schedule exist, when placement and temporal planning for the task exist.*

The issues of placement and temporal planning are firmly related; a feasible start time cannot be assigned without considering the placement of the tasks' modules.

Definition 6 (Feasible Placement) *Presuming, a set M_p of already placed modules $m_l \in M_p$ on a device with size (A_x, A_y) is given. With all modules $m_l \in M_p$ an origin x_l, y_l and the required space w_l, h_l of the modules is associated. A feasible spatial planning for a newly arriving task t_i at time a_i with the components $c_j \in C_i$, the selected modules m_k and the required space w_k, h_k of each module m_k exist, if for each m_k a position x_k, y_k on the reconfigurable device can be found satisfying the following conditions:*

I) $x_k + w_k \leq A_x \wedge y_k + h_k \leq A_y$

II) $\forall m_l \in M_p$:

$$[\ (x_l + w_l) \leq x_k \vee (x_k + w_k) \leq x_l\ ] \wedge$$
$$[\ (y_l + h_l) \leq y_k \vee (y_k + h_k) \leq y_l\ ]$$

As soon as a feasible placement for the selected module $m_k \in M_j$ has been determined, a start time $s_i \leq a_i$ can be assigned to the task t_i. Modules of tasks having finished execution are removed from the RD, i. e., $M'_p = M_p \setminus m_k$. Only when the task finishes prior its deadline d_i a feasible temporal planning exist.

Definition 7 (Feasible Temporal Planing) *A feasible temporal planning assigns a task t_i a start time s_i, so that $s_i + e_i + o_i < d_i$ holds true.*

e_i is the expected execution time of a task t_i and o_i contains the amount of configuration overhead.

Up next, the implementation of a basic scheduler follows.

3.1.1 Placement

Placement generally includes two steps, management of free space (partitioner) and fitting the application in this space (fitter). The partitioner keeps track of available free space on the device, while the fitter selects an area inside this free space depending on selection strategy. The research at hand is based on the routing-conscious, dynamic placement algorithm by Ahmadinia et al. [3]. This algorithm can be computed in $\theta(n \log n)$ time and considers manhattan distance between modules to select optimal placement. In contrast to [6, 63, 66], the algorithm by [3] manages free space by storing the location of occupied spaces and finding a placement for a single point instead of searching completely through all free rectangles. This is done by shrinking the reconfigurable device of an area $A = (0, 0, A_x, A_y)$ and simultaneously blowing up the already placed modules $m_p \in M_p$ by half of the width $w_j/2$ and half of the height $h_j/2$ of the module yet to be placed m_j. Equation 1 shrinks the area of the reconfigurable device A and Equation 2 blows the modules up.

$$A' = (\frac{w_j}{2}, \frac{h_j}{2}, A_x - \frac{w_j}{2}, A_y - \frac{h_j}{2}) \tag{1}$$

$$m'_p = (x'_p, y'_p, w'_p, h'_p) \tag{2}$$

Equation 3 and 4 calculates the new x and y position x'_p and y'_p of a placed module on the shrunken reconfigurable device A'.

$$x'_p = max(x_p - \frac{w_j}{2}, \frac{w_j}{2}) \tag{3}$$

$$y'_p = max(y_p - \frac{h_j}{2}, \frac{h_j}{2}) \tag{4}$$

Equations 5 and 6 determines the updated width w'_p and height h'_p of the already placed module.

$$w'_p = min(w_p + w_j, W - w_j) \tag{5}$$

$$h'_p = min(h_p + h_j, H - h_j) \tag{6}$$

In the end, all possible positions for m_j are reduced to points instead of rectangles. The fitter then selects a point location from the available free space. The points immediately on the edge of the chip are of particular interest, as they preserve the structure of free space in good shape. The contours of the free space are computed using the CUR algorithm (contour of union of rectangles) [31] known from computational geometry. It performs plane sweeps and uses segmentation trees as data structures. The optimal point for the module in the available free space is found by minimizing the manhattan distance between communication points.

3.1.2 Temporal planning

If a feasible placement for a task has been calculated and the task t_i meets its deadline d_i (feasible temporal planning) the task is accepted and placed on the reconfigurable device.

When considering multiple tasks with different area requirements, execution times, deadlines and arrival times, one has to order these tasks in a priority queue according to their specific requirements. Placement of the task of highest priority is then calculated first and, if a feasible placement exists, the task is loaded on the RD. Next the task of second highest priority is considered for placement, and so on. Steiger et al. [64] have shown that sorting the tasks according to their deadline leads to better results compared to ordering according to size. This sorting process is an adaption of the well known EDF dynamic scheduling algorithm for real-time systems. Furthermore, an currently unplaceble task is put on hold and the computation continues with the next task in queue. Danne [18] called this the EDF-NF (Next-Fit) algorithm and presented a schedulability test for periodic task sets. The Next-Fit strategy can place and execute tasks out-of-order, thus allowing a better utilization of the reconfigurable device. The work at hand uses the EDF-NF algorithm.

The following sections extends and optimizes the basic scheduler trying to maximize device utilization as well as minimizing communication distance between communication partners. Module selection is considered the first optimization necessary, focusing on device utilization.

3.2 Integrated module selection

The selection of a module from a set of modules implementing the same functionality but having different constraints poses the first issue to turn to. Research on the matter was published in [MCHB11] and this section is a derivation of this paper.

Tasks can be computed by different implementations (modules) of a component varying in size, execution time and communication overhead: Computing an Fast Fourier Transformation (FFT), for instance, involves the butterfly operation. A N-point Discrete Fourier Transformation (DFT) needs $log_2(N)$ stages with $N/2$ radix-2-butterflies per stage. Viable FFT implementations need to calculate one radix-2-butterfly operation at minimum, covering the area a, or at most $N/2$ butterflies covering an area of roughly $a \cdot N/2$. This does not consider the area needed for communication. Thus the execution time of a N-point DFT varies when done on different modules. The largest implementation computes $N/2$ butterflies in parallel and is about $N/2$ times faster then the smallest one. Other tasks, as present in image processing, sort algorithms or applications using the Monte Carlo method show a similar behavior. The availability of several modules of different characteristic implementing the very same component of a task, does not only expand the design space for static designs, but also for dynamic implementations like Dynamic Networks-on-Chip. In particular, selecting the best fitted module has impact on both schedulability and optimality of an implementation.

In Figure 12 the tasks t_1, t_2, t_3 and t_4 with one component per task are placed on a DyNoC. Task t_5 has one component implemented by two modules m_1 and m_2. The modules differ in size and execution time with m_1 having the shorter execution time but covering a larger area of the RD.

A feasible placement of t_5 is only possible with module m_2, as m_1 excels the available space. Still, the execution time of m_2 must not excess the deadline of t_5. Module selection is an extension to temporal planning and placement steps and is integrated in the scheduler.

This section only considers tasks consisting of one component in order to evaluate the achievable performance through integrated temporal planning, module selection and placement. The problem definition for integrated temporal planning, module selection and placement reads as follows:

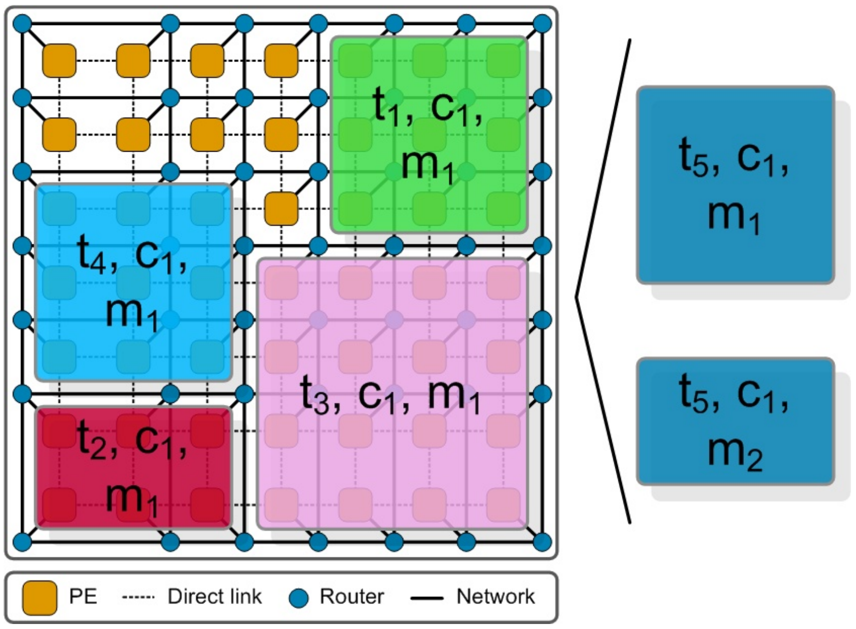

Figure 12: Module selection of task t_5

Definition 8 (Integrated temporal planning, module selection and placement) *Given a set T of tasks $t_i \in T$ with arrival times a_i, components C_i and associated modules $m_j \in M_j \subset M_i$. For each component $c_j \in C_i$ selects a module $m_j \in M_j$ in such a way that a feasible schedule for the maximal number of tasks t_i exists.*

3.2.1 Module selection strategies

State-of-the-art on-line schedulers only consider one module per task. With the availability of several modules able to execute a particular task, a fitting module has to be selected during runtime. Selecting the right module is crucial when considering device utilization and rejection rates. A solid module selection strategy should be dynamic in order to allow adaption to the current utilization of the device and timing requirements of the task. Another advantage of having multiple modules per task is the possibility to use other modules, if the initially chosen one cannot be scheduled.

Dynamic Module Selection The *Largest Module First (LMF)* heuristic picks the module with the largest area requirement first. When the largest module can not be placed, because free space is not big enough, the second largest module is chosen. The LMF is invoked over and over again until a feasible placement for the task has been found or the execution time of the chosen module exceeds the deadline. This strategy tries to compute a task as fast as possible.

Smallest Module First (SMF) is the counter part to LMF and considers the module with the smallest area requirements first. When the runtime of this module conflicts with the given deadline of the task, the next largest module is chosen until the smallest possible module is found which meets the deadline. This strategy tries to maximize free space by using only the smallest operable module.

Static Module Selection While dynamic selection heuristics allow the adaption to the device state or timing by selecting smaller (slower) or larger (faster) modules, static heuristics preselect a module for placement at compile time and do not take other modules for a task into account. The following static module selection heuristics are discussed as a comparison to dynamic heuristics.

Largest Module Only (LMO) considers the module with the largest area requirement only.

Smallest Module Only (SMO) heuristic chooses the module with the smallest area requirement which meets the deadline d_i.

Average Size Module Only (ASMO) just picks the module of overall average size.

Module selection is integrated in the basic scheduler as an extension. Pseudo code for this extended scheduler is given next considering only tasks with one component.

3.2.2 Extended scheduling algorithm

The scheduler handles tasks by managing them in different lists according to their current state. When a task arrives it is put in the $T_{FLOATING}$ list and the scheduler is called. $T_{FLOATING}$ holds all plannable tasks and sorts them by means of the EDF-Next-Fit temporal planning algorithm. The first task t_i of the list is checked for feasibility. A task is considered feasible, if there is at least one module for each component meeting the deadline $t_i.deadline$ of the task. If the task is not feasible, it is rejected and put into the $T_{REJECTED}$ list. A module m_j of a feasible task is selected according to the module selection strategy and the placer (spatial planning) is invoked. If placement is successful, the task is loaded on the device and put onto the $T_{RUNNING}$ list, otherwise the next module of the task is chosen depending on the module selection strategy. If no feasible module can be placed, the next task is considered (next fit) and t_i is put back in $T_{FLOATING}$ again. It might be possible to schedule the task at later point.

Algorithm 1 Scheduling with module selection

```
$success \leftarrow FALSE$
$k \leftarrow 0$
$Sort(T_{FLOATING})$
**while** ( $k <= next\_fit$ ) **do**
  $t_i \leftarrow Pop(T_{FLOATING})$
  $c_j \leftarrow t_i.C_i$
  $M_s \leftarrow c_j.M_j$
  **if** ( $time() + M_s.MinRuntime() < t_i.deadline$ ) **then**
    **while** ( **not** $success$ **and** $M_s \neq \emptyset$ ) **do**
      $m_k \leftarrow SelectModule(M_s)$
      $success \leftarrow Placement(t_i, c_j, m_k)$
    **end while**
    **if** ( $success$ ) **then**
      $t_i.start\_time \leftarrow time$
      $t_i.state \leftarrow RUNNING$
      $T_{RUNNING} \leftarrow t_i \cup T_{RUNNING}$
    **else**
      $t_i.state \leftarrow FLOATING$
      $T_{FLOATING} \leftarrow t_i \cup T_{FLOATING}$
    **end if**
  **else**
    $t_i.state \leftarrow REJECTED$
    $T_{REJECTED} \leftarrow t_i \cup T_{REJECTED}$
  **end if**
  $k \leftarrow k + 1$
**end while**
```

Algorithm 1 shows the pseudo code of one run of the scheduler. The algorithm has an overall complexity of $O(n \log n)$. In [3], Ahmadinia et al. showed the placement algorithm to be of a complexity of $O(n \log n)$. All other steps of the scheduling algorithm, like the ordering of tasks in a priority queue, are simply accessing and sorting lists taking $O(n)$ or $O(n \log n)$ steps respectively. Thus, the overall complexity of the scheduling algorithm remains in the $O(n \log n)$ domain. The scheduler is integrated in a simulation environment and is implemented in the Python programming language.

3.2.3 Evaluation

The performance of the module selection strategies is evaluated using the percentage of rejected tasks and the communication distance. All task sets are evaluated using EDF-Next-Fit scheduling with next-fit values of $k \in \{0, 1, 3, 10, \infty\}$ and a device size of 100×100. No precedence of tasks is considered, as only the influence of dynamically selecting modules is evaluated.

Several task sets of different degrees of device utilization were benchmarked and three representative task sets featuring high utilization were chosen. Each task set TS1, TS2 and TS3 contains 500 tasks with one component per task. All modules were created by generating one rectangular base module with random dimensions between 4 and 32 for width w and height h and a random runtime between 16 and 48 time steps. This module serves as the derivative source for all other modules of the task by changing the size and calculating the runtime depending on the change of size, e. g. a module with half the area demand runs for double the time. The deadline of the task was randomly chosen, the only constraint being all modules of the task were able to meet the deadline. Arrival times of tasks were randomly chosen between 0 and 500 time steps and each task uses randomly chosen input and output interfaces at the border of the device. The main difference between the task sets TS1, TS2 and TS3 is the amount of modules per task. A task in TS1 has seven modules, while a task in TS3 only has three modules.

Table 1 shows the rejection rate of the three task sets. TS1 has seven modules per task ranging from 0.04 % to 54.8 % of the device size. Dynamic module selection using LMF leads to no task rejections in TS1, while SMF, the runner-up, comes out at a rejection rate of 3.8 % in comparison. If enabling out-of-order placement by increasing k, the difference between all strategies shrinks, but still LMF gives the best results. For TS2 and TS3 less modules per task were available, limiting the selection flexibility and resulting in an overall worse rejection rate of the module selection strategies.

Both LMF and SMF heuristics outperform the static strategies in almost all cases. While the LMF strategy has no tasks rejected in TS1, with TS2 and TS3 the reconfigurable hardware becomes saturated and the situation changes. LMF basically behaves as follows. First larger modules are placed and the free space between the large modules is filled-up with smaller modules. When a large module finishes, a module with at most the size of the finished module can be placed. Generally, the newly placed module will be smaller

Table 1: Rejection rate for three task sets and varied next-fit value k

Task Set	k	LMF	LMO	SMF	SMO	ASMO
	0	0.0 %	20.6 %	3.8 %	11.4 %	12.6 %
TS1	1	0.0 %	14.8 %	2.0 %	9.4 %	10.4 %
0.04 % -	3	0.2 %	11.8 %	0.2 %	8.6 %	8.6 %
54.8 %	5	0.4 %	10.4 %	1.0 %	9.0 %	9.0 %
7 modules	10	1.4 %	7.0 %	2.0 %	9.0 %	9.0 %
	∞	1.8 %	7.2 %	4.2 %	9.0 %	9.0 %
	0	0.8 %	17.4 %	7.6 %	11.2 %	13.8 %
TS2	1	0.2 %	11.8 %	4.8 %	10.2 %	10.2 %
0.04 % -	3	0.4 %	6.6 %	3.6 %	8.8 %	8.8 %
36.0 %	5	0.6 %	5.4 %	2.0 %	8.8 %	8.8 %
5 modules	10	2.4 %	4.0 %	3.8 %	8.8 %	8.8 %
	∞	3.4 %	7.6 %	6.8 %	8.8 %	8.8 %
	0	12.8 %	24.0 %	17.6 %	17.6 %	18.6 %
TS3	1	9.4 %	18.0 %	13.0 %	15.2 %	15.2 %
0.09 % -	3	6.6 %	12.8 %	10.6 %	14.8 %	14.8 %
16.0 %	5	5.8 %	10.2 %	9.2 %	14.8 %	14.8 %
3 modules	10	7.8 %	9.8 %	10.8 %	14.8 %	14.8 %
	∞	7.8 %	10.2 %	11.2 %	14.8 %	14.8 %

creating a small area of free space which in turn can be filled by smaller modules, if available. SMF running in saturated state has finishing modules create small areas of free space allowing the placement of only small and smaller modules and thus leading to a higher device fragmentation, which leads to a higher rejection rate. Additionally, the available time frame between task arrival and latest execution start time is significant shorter compared to LMF, limiting possible start times.

Dynamic module selection offer a chance to lower rejection rate compared to static strategies. The influence of these strategies on communication distance is evaluated next by using the manhattan distance as metric.

The influence of module selection on average manhattan distance per task differs dependent on strategy, illustrated in Figure 2. Basically speaking, the lower the rejection rate (higher device utilization), the longer the communication distance will be. More tasks are placed on the device minimizing the chance of having free space with a short

Table 2: Rejection rate and average manhattan distance for module selection heuristics for $k = 0$

Task Set	Heuristic	Rejection Rate	Manhatten Distance [hops]
TS1 0.04 % - 54.8 % 7 modules	LMF	0.0 %	136.0
	LMO	20.6 %	104.4
	SMF	3.8 %	113.8
	SMO	11.4 %	112.2
	ASMO	12.6 %	111.1
TS2 0.04 % - 36.0 % 5 modules	LMF	0.8 %	134.7
	LMO	17.4 %	110.7
	SMF	7.6 %	116.2
	SMO	11.2 %	110.7
	ASMO	13.8 %	109.4
TS3 0.09 % - 16.0 % 3 modules	LMF	12.8 %	126.1
	LMO	24.0 %	114.0
	SMF	17.6 %	111.2
	SMO	17.6 %	113.8
	ASMO	18.0 %	113.8

manhattan distances. Results from LMF suggests the same, as it has the largest average manhattan distance and the lowest rejection rate for all task sets. SMF on the other hand showed a low average manhattan distance while providing a good device utilization, only surpassed by LMF. Modules-to-place are significantly smaller with SMF and free space is higher fragmented making it overall easier to place tasks closer to their communication partners compared to LMF.

Module selection allows the resource manager to adapt the tasks to the device state by considering several implementations differing in runtime and size. The next section provides a strategy to improve the on-line scheduler with respect to communication distance and rejection rate by allowing task preemption and enabling relocation of tasks during runtime.

3.3 Relocation of tasks

This section is a modified version of [MBed]. The actual device state with a set of placed tasks can lead to suboptimal placements for new tasks in terms of communication distance. Tasks cannot be located at their optimal position due to other task occupying the area or the lack of free space.

Figure 13 (left) shows the placement of four tasks t_1 to t_4 with one component per task implemented by a module. Tasks t_1 and t_3 are located optimally with minimal communication distances regarding their communication partners at the border of the device (marked by colored arrows), whereas tasks t_2 and t_4 are not placed optimally. The right side of Figure 13 shows a placement later in time with task t_1 removed and t_2 and t_4 relocated, leading to optimal placements. for t_2 and t_4. The communication distance between communication partners has been reduced to a minimum.

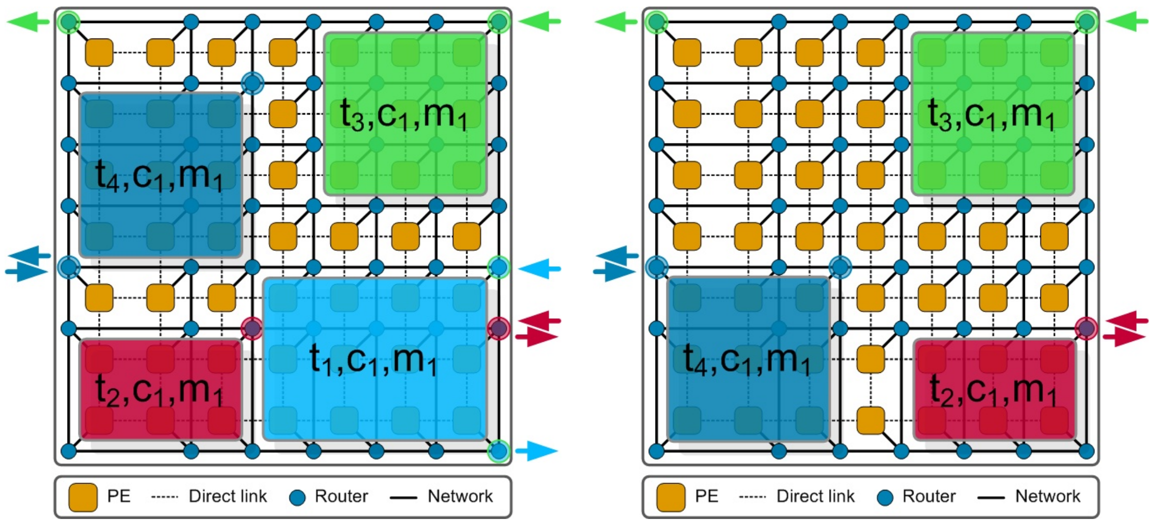

Figure 13: Relocation of modules m_2 and m_4 after m_1 was removed on a Dynamic Network-on-Chip

Reducing communication distance in a network-based reconfigurable device means lowered latency and network load and therefore increased performance for both computation and communication. Relocation of tasks during runtime allows the optimization of placements by reducing communication distance in regard to the communication partners of placed tasks. The definition of relocation used in this work is as follows:

Definition 9 (Relocation) *Select a feasible set of tasks T_r from the set of placed tasks T_p and relocate these tasks, so that the communication distance cd_r of the tasks in T_r is minimized.*

The several consecutive steps necessary to relocate tasks are topic of the next section.

3.3.1 Relocation strategies

In order to enforce relocation of tasks some consecutive steps have to be executed at the time of relocation. First, the set of placed tasks is searched for a potential subset of tasks. Secondly, the selected tasks are sorted following an ordering strategy and finally the ordered tasks are placed one after another. The final placement step is a simple call of the placement algorithm introduced in section since the goal of minimizing communication distance remains the same. Thus, this step is not described here and only the moment of relocation, the selection of tasks and the sorting of task are discussed.

Relocation timing Two different strategies when to relocated tasks seem applicable, triggered relocation and periodic relocation. The former triggers the execution of relocation, whenever an event like rejection of a task, completion of a (certain) task (see Figure 13), or crossing of a communication bound occurs. The latter runs relocation whenever a fixed number of time steps ts has passed.

Relocation selection Selecting the tasks for relocation is of major interest for the cause and several approaches exist. An important criterion is the existence of a feasible schedule for each selected task so that only tasks are selected for which an feasible temporal and spatial planning exists. The temporal overhead o_i introduced by relocation can deny tasks to finish execution before its deadline d_i, because $s_i + e_i + o_i > d_i$. Definition 10 ensures the existence of a feasible temporal planning and that the task can be resumed when placeable. This definition does not guarantee that feasible placement (spatial planning) exists for each task in T_r.

Definition 10 (Weak Feasible Relocation Selection) *Given a set $T_p \subset T$ of placed tasks. A feasible relocation selection assigns tasks $t_r \in T_p$ to $T_r \subseteq T_p$, so that a feasible temporal planning for all $t_r \in T_r$ exist with $o_i \geq 0$.*

In contrast Definition 11 demands a feasible schedule to exist.

Definition 11 (Strong Feasible Relocation Selection) *Given a set $T_p \subset T$ of placed tasks. A feasible relocation selection assigns tasks $t_r \in T_p$ to $T_r \subseteq T_p$, so that a feasible schedule for all $t_r \in T_r$ exist with $o_i \geq 0$.*

Because the placer does only consider tasks one after another even if several tasks are selected at once for relocation, feasible placement for all tasks cannot be guaranteed. During relocation the placement of tasks (and the free space) is changed compared to the original placement, this can result in device states with feasible temporal planning for a task, but spatial planning failing. In Figure 14 three tasks t_1, t_2 and t_3 with corresponding modules m_1, m_2 and m_3 are placed on a DyNoC. The arrival times of the tasks are $a_1 < a_2 < a_3$ and the deadlines are $d_3 < d_2 < d_1$. The tasks t_i are placed at their time of arrival a_i and all tasks are selected for relocation $\{t_1, t_2, t_3\} \in T_r$

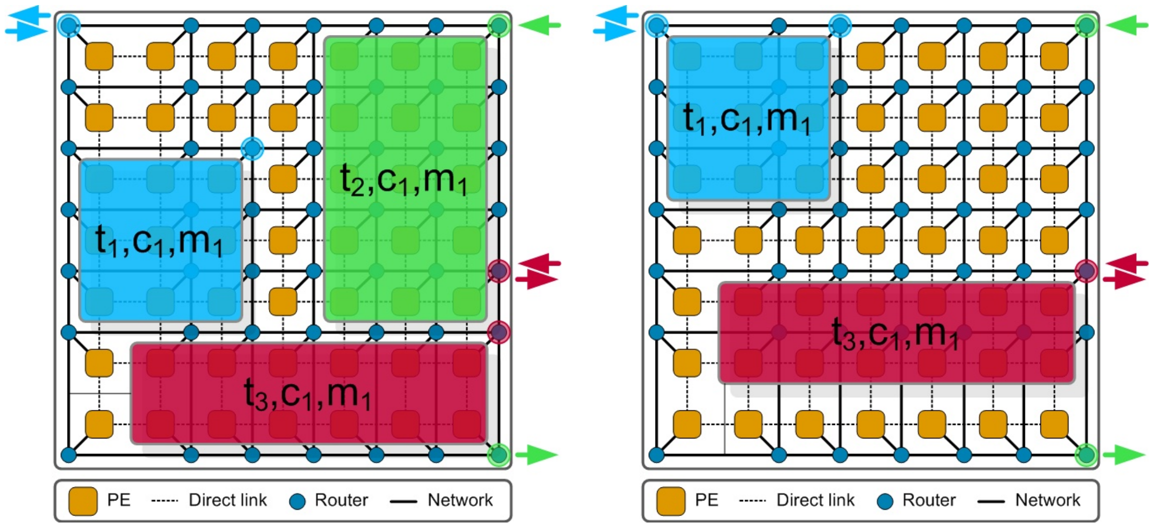

Figure 14: Weak feasible relocation selection of tasks

The ordering of tasks has influence on the extend of performing successful task relocations. For instance, consider sorting of the relocation selection T_r by their deadlines d_i. Task t_3 will be selected first and located at its optimal position. However, due to the fragmentation of free space, task t_2 can not be successfully placed anymore. There is no guarantee for a feasible schedule for all selected tasks. If sorting the tasks by their size $w_i \cdot h_i$ though, task t_2 is placed first and allows successful placement of all three tasks. So, sorting of the relocation selection directly influences relocation results.

Relocation ordering Two sorting schemes focused on deadlines and task sizes are considered here to sort the relocation selection. The *Largest Task First (LTF)* heuristic sorts tasks by their area requirements and considers the largest task first. This strategy attempts to optimize placement by placing larger tasks first in order to keep the allocatable free space as defragmented as possible. Additionally, sorting by deadlines with the EDF strategy is considered.

3.3.2 Extended scheduling algorithm

The pseudo code of the scheduling algorithm with module selection given in Chapter 3.2.2 is extended to support periodic relocation (see Algorithm 2). Also tasks with multiple components are considered here. Therefore the auxiliary variable *successComponent* is introduced, which stores the success of components' placements of a task. When and only if all components can be placed successfully, the placement is regarded executable for the task, the task is started and put into the $T_{RUNNING}$ list.

3.3.3 Evaluation

Performance of the relocation strategies is evaluated using the percentage of rejected tasks and the average path length of a task by varying periodicity of relocation as well as overhead. Results are interpreted considering the average task runtime and slack time[2]

A synthetic task set TS4 containing 200 tasks with one component and one module each was created for evaluation. Runtime e_i of the tasks is between 16 and 32 time steps ts with an average of 24 ts. The average slack time of a task $t_i \in TS4$ is $2.7 \cdot e_i$ which is about 65 ts. The deadlines are generated depending on the randomly generated arrival times and runtimes and each task is able to meet its deadline. Device size is 100×100 and module size ranges between 0.064 % and 6.25 % of the device size. Each component has one input and one output communication partner at the border of the device. These communication partners are randomly chosen out of all points at the edge. TS4 is evaluated using EDF-Next-Fit scheduling with a fixed next-fit value of $k = 3$. Only periodic relocation is considered.

[2]Slack time is the temporal difference between deadline, time when the task is ready to be executed and runtime. Slack time is also known as laxity

Algorithm 2 Scheduling with periodic relocation

```
$k \leftarrow 0$
$Sort(T_{FLOATING})$
$PeriodRelocateTasks(T_{RUNNING}, period, time())$
while ( $k <= next\_fit$ ) do
  $successTask \leftarrow TRUE$
  $t_i \leftarrow Pop(T_{FLOATING})$
  $C_h \leftarrow t_i.C_i$
  while ( $C_h \neq \emptyset$ ) do
    $c_j \leftarrow Pop(C_h)$
    $M_s \leftarrow c_j.M_j$
    if ( $time() + M_s.MinRuntime() < t_i.deadline$ ) then
      $successComponent \leftarrow FALSE$
      while ( not $successComponent$ and $M_s \neq \emptyset$ ) do
        $m_k \leftarrow SelectModule(M_s)$
        $successComponent \leftarrow Placement(t_i, c_j, m_k)$
      end while
      $successTask \leftarrow successTask$ and $successComponent$
    else
      $t_i.state \leftarrow REJECTED$
      $T_{REJECTED} \leftarrow t_i \cup T_{REJECTED}$
      $EXIT$
    end if
  end while
  if ( $successTask$ ) then
    $t_i.start\_time \leftarrow time$
    $t_i.state \leftarrow RUNNING$
    $T_{RUNNING} \leftarrow t_i \cup T_{RUNNING}$
  else
    $t_i.state \leftarrow FLOATING$
    $T_{FLOATING} \leftarrow t_i \cup T_{FLOATING}$
  end if
  $k \leftarrow k + 1$
end while
```

Relocation overhead and periodicity of relocation are varied and for each combination a data point consisting of manhattan distance and rejection rate is generated. The static overhead is modified from 0 to 7 time steps which corresponds to 0 % to almost 30 % of the average task runtime. Periodicity is varied by starting relocation every ts, and then up to every 49 time steps which is about two times of the average runtime of a task. Without relocation TS4 saturates the device so that 5 % of the tasks are rejected. The average manhattan communication path in this case has a hop count of 128.

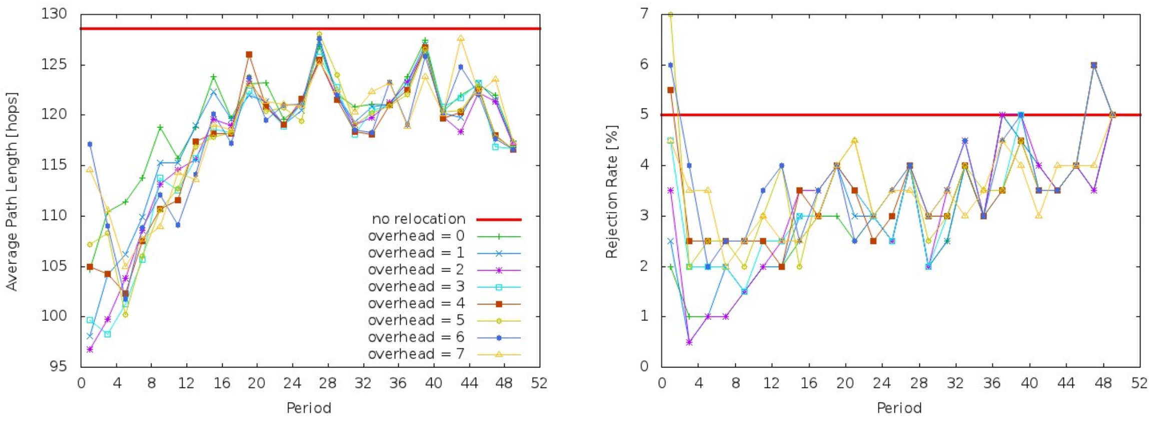

Figure 15: Periodic relocation with LTF

For LTF heuristic Figure 15 and for EDF Figure 16 show the rejection rate and path length of TS4 under variation of relocation overhead and periodicity.

Periodic relocation under LTF lowers the average path length of TS4 for all simulated parameters. Carrying out relocation every time step ts leads to an increased rejection rate when considering overheads larger then 16 % of the average task runtime ($o_i \geq 4$) because task variation of consecutive task selections is lower for small period values. Each time a task is relocated, the overhead o_i is added to the tasks finishing time. So, the relocation selection T_r is dependent on the periodicity. For a period between 3 and 13 time steps which corresponds to 12.5–54 % of the average runtime of a task both the path length and rejection rate show better results compared to a simulation of TS4 with relocation disabled. For a relocation overhead of $o_i = 3$ and a period of 3 the average path length is reduced by 23 % and the rejection rate is lowered from 5 % to 2 %.

The EDF sorting heuristic behaves similar in regard to results obtained by LTF as shown in Figure 16. Again a minimum between period 3 and 5 for both rejection rate

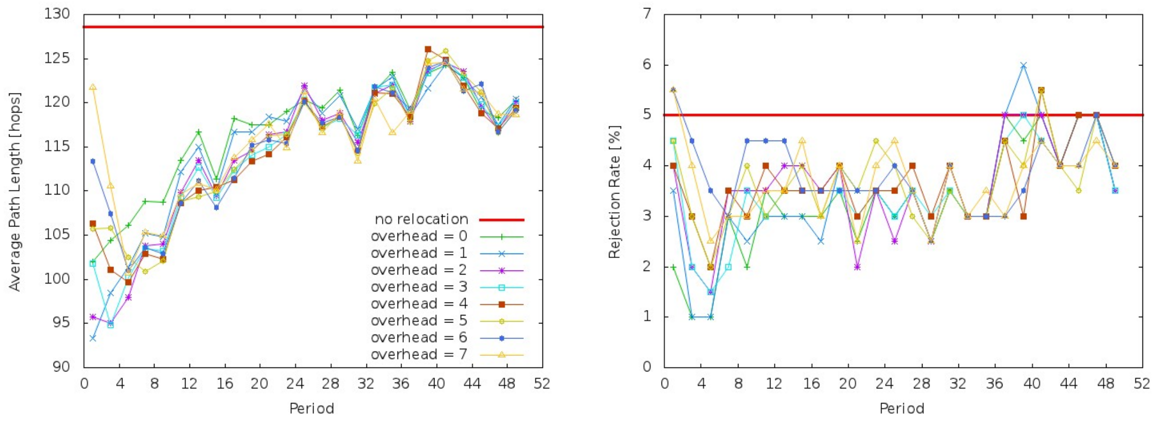

Figure 16: Periodic relocation with EDF

and path length exist. The path length is reduced by 25 % and the rejection rate changes from 5 % to 1,5 % when considering a small relocation overhead.

Both LTF and EDF relocation strategies improve the scheduler, but overall EDF shows slightly better results for TS4.

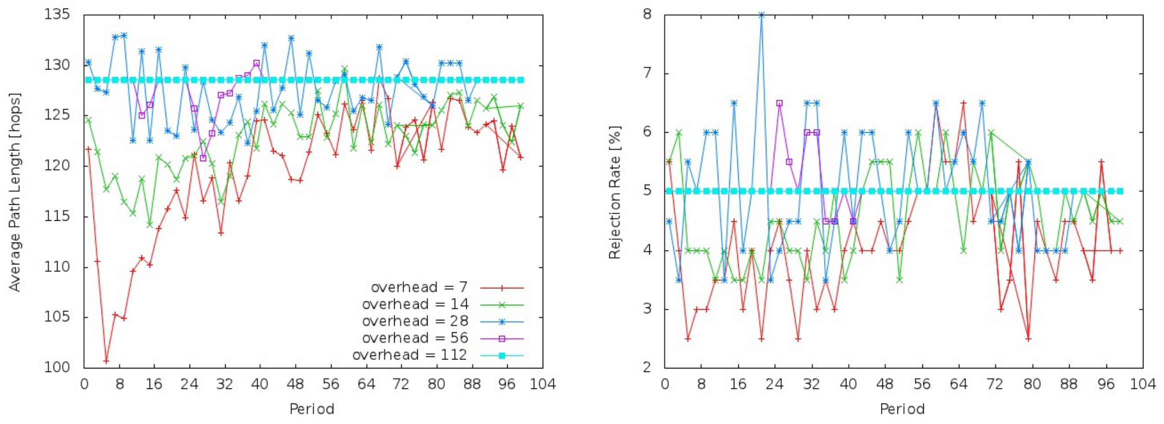

Figure 17: Periodic relocation with large overheads

As relocation overhead is dependent on different properties like hardware architecture, task preemption strategy and network load, the influence of higher overheads should be examined in order to determine overhead bounds. In Figure 17, overhead is increased so that the relocation of a task exceeds the deadline (overhead=112). In this case no relocation is executed and results are identical to the results without relocation, as given in Figures 16 and 15. A relocation overhead of about 58 % of the average task runtime or 20 % of the slack time (overhead=14) still leads to a lowered average path length for periods of up to 100 ts.

Relocation offers the possibility to increase system performance by decreasing rejection rate and communication distance.

3.4 Influence of dynamic placement on path length

Up to now, communication distance was calculated using manhattan distance; the distance between two points is calculated by the sum of the absolute differences of their coordinates. Manhattan distance approximates the actual path length. Because of the dynamic placement and removal of tasks, communication paths can be blocked by modules so that the absolute differences of coordinates cannot be applied to determine the actual path length. In [10, 11, 9] the Surrounding-XY routing algorithm (S-XY) for Dynamic Networks-on-Chips was described. It is used here to determine the correct path. The communication overhead in hop counts is calculated by comparing the manhatten distance with the weighted average S-XY distance. By using the weighted average, the influence of dynamic placement on communication distance is taken into account. Task placement on the device changes over time and so the ideal routes from sender to receiver, as calculated by the S-XY routing algorithm, vary.

In Figure 18 an example for S-XY routing is depicted. While t_1 is placed so that each communication partner of t_1 is on the optimal path, t_2 is not placed optimally. Additionally, t_1 presents an obstacle for the communication of t_2 as a path around t_1, c_2, m_1 has to be taken. The hop count from source to sink of t_2 is 10 using manhattan distance and increases to 14 when S-XY routing is used to route packets.

Table 3 shows the difference between path lengths using manhattan distance and S-XY routing. The actual hop count for task sets TS1, TS2, TS3 and TS4 is determined with S-XY routing. For task set TS4, periodic relocation was used (with EDF sorting, overhead $o_i = 3$ and a period of 3) and for TS1-TS3 the LMF heuristic was used and the next-fit value k was set to 0.

The influence of module selection strategies on the actual distance is given in Table 4.

Both dynamic module selection strategies, SMF and LMF, increase path length due to the usage of dynamic S-XY routing. The increase again is about 16 %. LMO has the

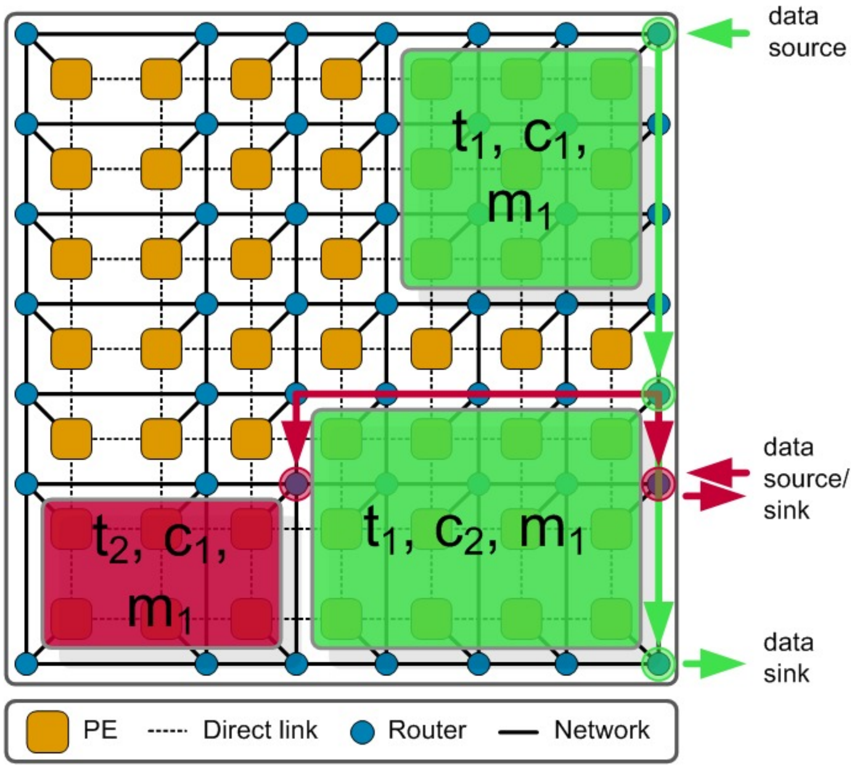

Figure 18: Path length using S-XY routing

lowest increase and overall path length, but also the highest rejection rate (see Table 2). At lower device utilization, tasks can be placed closer to their optimal location leading to shorter communication paths and a lower dynamic routing overhead. Overall, S-XY routing introduces an increase in path length of 7.85 % to 17.58 % for the given task sets at an average of about 16 % compared to manhattan distance.

3.5 Off-line spatial planning with ASP

In the previous sections communication-aware on-line scheduling algorithms have been considered and results for several task sets were given. With the on-line approach a feasible placement can be found in $\theta(n \log n)$ time for each task dealt with one after the other.

In Figure 19 an overall optimal placement for tasks t_1 and t_2 is given. Compared to the placement seen in Figure 18, where only t_1 is placed optimal, the total path length is

Table 3: Comparison of path length calculated with manhattan distance and average S-XY routing distance for TS1, TS2, TS3 and TS4

Tasks Set	Manhatten Distance [hops]	Avg. S-XY Distance [hops]	Increase [%]
TS1	136.00	159.75	15.98
TS2	134.66	156.86	16.49
TS3	126.06	146.99	16.60
TS4	119.57	138.03	15.43

Table 4: Influence of module selection strategy on path length for TS1

Heuristic	Manhatten Distance [hops]	Avg. S-XY Distance [hops]	Increase [%]
LMF	136.00	159.75	15.98
SMF	113.80	131.51	15.55
LM0	104.41	112.61	7.85
SM0	112.24	131.98	17.58
ASM0	111.10	130.54	17.49

reduced by 4 hops when considering communication distance. t_2 now has a path length of 2 hops while the path length of t_1 is increased by 5 hops to 14 hops. This means, the total path length is reduced by 16 %. While an optimal placement for one task can be achieved by the on-line scheduler, an optimal allocation for a set of tasks needs to consider all tasks before actual placement. Unfortunately, finding the optimal placement for a set of tasks poses an NP-hard problem [57].

This chapter introduces an off-line placer for homogeneous and heterogeneous reconfigurable devices in order to determine the optimal placement. Arrival times a_i, start times s_i, execution times e_i and deadlines d_i of tasks are not considered here.

3.5.1 Problem solving

The declarative problem solving paradigm Answer Set Programming (ASP) is used to find optimal placements. ASP, similar to boolean satisfiability, takes advantage of highly effective boolean constraint technology and offers a simple modeling language. Basically,

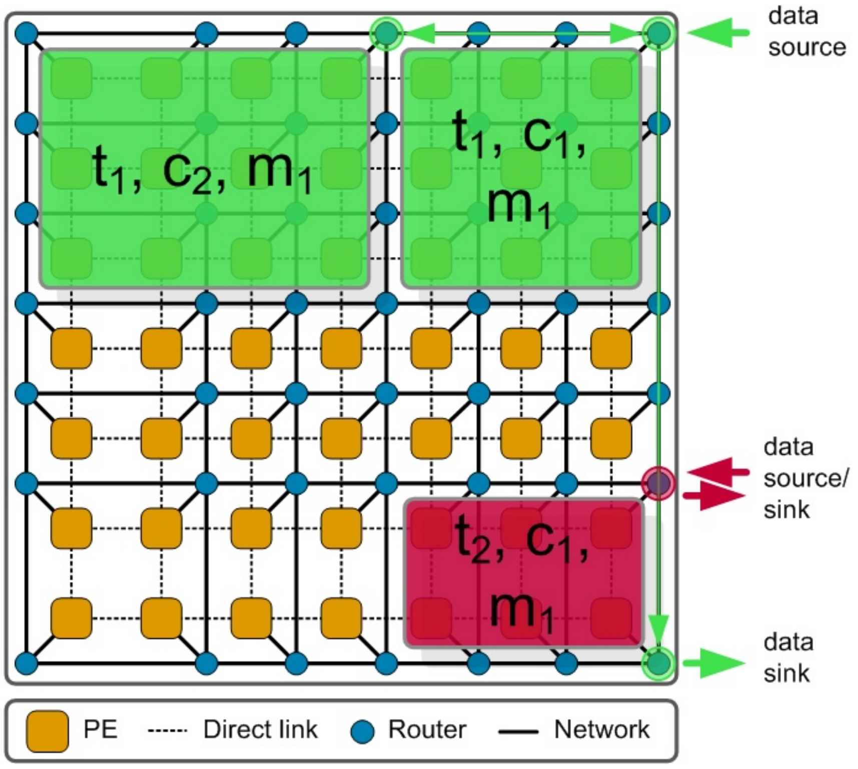

Figure 19: Optimal placement

ASP encodes the problem as a logic program such that its set of answers represents the solutions from which the optimal solution can be picked. The Potsdam Answer Set Solving Collection (Potassco) [58] is used to model, ground and solve the problem. Similar to the on-line placer, the goal is to place a task such that the path length, the manhattan distance between communication points of a task, is minimized and the maximum number of tasks are placed. The modeling of the goal is shown in the following five rules.

$$\#maximize\{place(Task, Component)@2\}.$$

With the *maximize* rule the number of placement of tasks considering all components is maximized. The *minimize* rules minimize the communication distance between the components of a task ($distx$ and $disty$) and between components and interfaces at the border of the device ($distinterfacex$ and $distinterfacey$).

$$\#minimize\{distx(Task, Component, Component2, X)@1\}.$$
$$\#minimize\{disty(Task, Component, Component2, Y)@1\}.$$
$$\#minimize\{distinterfacex(Task, Component, X)@1\}.$$
$$\#minimize\{distinterfacey(Task, Component, Y)@1\}.$$

The *maximize* rule has a priority of 2 and is prioritized higher.

3.5.2 Evaluation

Runtime In this section the influence of device utilization and the amount of communication points per task on the runtime of the off-line placer is evaluated. All tests were performed on a PC using an Intel Core2Duo E8500 with 2 GByte of RAM. Table 5 shows the results for the placement of a varying number of tasks on a 50×50 homogeneous device. Each task consisted of one component with one module of size 7×13. The amount of tasks to be placed was increased one by one for every test case and the time to calculate the optimal answer is given. The amount of interfaces (IF) per task was varied from 0 to 2 in order to evaluate the influence of communication between tasks and communication points at the border of the device. The communication points were identical for all tasks and positioned at 0/25 for IF1 and 49/25 for IF2 respectively.

In each test case answers were found, but only when a time is given, the ASP solver was able to find the optimal solution. In all other cases, a feasible solution was found. With IF0 no communication points were considered thus only the *maximize* (placement) rule is used as the objective function. Up to 21 tasks can be placed on the device at most. Thus, there is no solution for placing 22 tasks and the solver timed out after running more than 8 hours. However, the last answer found actually was the optimal answer, but the solver was not able to conclude that no better answer exists.

In all test cases for IF1 and IF2, at least a feasible solution was found by the ASP solver. With one or two interfaces in the mix, the problem becomes harder, as the *minimize* (communication) rules are additionally considered as objective functions. However, in all cases for IF0, IF1 and IF2 the maximal possible amount of tasks were placed on the device, so that the *maximize* (placement) rule was met. It remained open, if the found feasible answers were in fact the optimal ones, as eventually the solver always ran out of memory.

Table 5: Off-line placer runtime

#Task	Time [s]			Utilization [%]
	IF0	IF1	IF2	
1	0.015	0.350	1.850	3.64
2	0.230	91.950	76.890	7.28
3	0.330	75.750	1927.230	10.09
4	0.015	1521.910	solution found	14.56
5	0.430	940.780	solution found	18.20
6	0.560	2555.550	solution found	21.84
7	0.680	solution found	solution found	25.48
8	0.790	solution found	solution found	29.12
9	1.330	solution found	solution found	32.76
10	1.130	solution found	solution found	36.40
11	1.430	solution found	solution found	40.04
12	2.060	solution found	solution found	43.68
13	2.080	solution found	solution found	47.32
14	3.290	solution found	solution found	50.96
15	6.010	solution found	solution found	54.60
16	21.250	solution found	solution found	58.24
17	45.150	solution found	solution found	61.88
18	232.730	solution found	solution found	65.52
19	504.440	solution found	solution found	69.16
20	525.590	solution found	solution found	72.80
21	328.280	solution found	solution found	76.44
22	time-out	solution found	solution found	80.08

Comparison with on-line placer After showing that the ASP off-line placer is able to determine answers (even if they are not the optimal ones), it appears interesting to compare these answers with the on-line placement algorithm and to evaluate the quality of the answers. A task set was derived from real world hardware components using synthesis results for a Virtex-4 FX100 FPGA. In Table 6 the tasks t_i of task set TS5 with their components c_i can be referenced. Each component has one module with width and height w and h as well as communication partners In and Out. The device size was limited to 50×50 and Ipc1 to Ipc3 describe different basic image processing cores.

Table 6: Task set TS5 with six tasks each consisting of up to four components

Task t_i	Component c_i	Name	#Slices	Size w/h	In	Out
t_0	c_0	Bayer2RGB	734	8/10	0/25	c_1
	c_1	MJPEG_Encoder	4182	20/22	c_0	c_2
	c_2	Ethernet	650	8/10	c_1	49/25
t_1	c_0	Ipc1	489	6/8	25/0	c_1
	c_1	Ipc2	2721	16/18	c_0	c_2
	c_2	MJPEG_Encoder	4182	20/22	c_1	c_3
	c_3	Ethernet	650	8/10	c_2	25/49
t_2	c_0	Bayer2RGB	734	8/10	0/25	c_1
	c_1	Integral_Sum	532	8/8	c_0	c_2
	c_2	Ipc2	2721	16/18	c_1	c_3
	c_3	VGA_Out	85	2/4	c_2	0/0
t_3	c_0	MJPEG_Decoder	8825	30/30	0/20	c_1
	c_1	Ipc3	44	2/4	c_0	c_2
	c_2	VGA_Out	85	2/4	c_1	0/0
t_4	c_0	AES128	7639	28/28	49/0	49/0
t_5	c_0	AC97_Cntrlr	467	6/8	49/49	49/49

Because the on-line placer allocates tasks one after the other, the results vary depending on the order of the tasks. Hence, the order was randomized and results of ten placer runs were generated. The next-fit value was fixed to $k = 3$. Table 7 shows the results.

Dependent on task ordering, between two and four tasks could be placed leading to a device utilization of 26 % to 68 %. On an average, 2.8 tasks could be placed at a manhattan distance of 121.3 hops and a device utilization of 55 %. The influence of the task order on the results especially emerges from the first three rows of Table 7 where tasks T0, T3 and T5 had been placed in different succession. The average manhattan distance per task in row two 7 is 47 % higher than the corresponding distance in row one yielding an increase of about 38 hops per task, whereas device utilization is the same.

The off-line placer on the other hand found a feasible solution, but was not able to determine the optimal solution as it ran out of memory. However, the provided solution still surpasses the best results of the on-line placer. Table 8 shows the results in comparison

Table 7: Results of on-line placer for TS5

#Task	Tasks	Manhatten Distance [hops]	Utilization [%]
3	T3, T0, T5	81.0	56
3	T3, T5, T0	119.7	56
3	T5, T3, T0	123.7	56
2	T5, T0	136.5	26
3	T1, T5, T0	125.7	60
3	T5, T3, T1	126.3	66
3	T3, T1, T5	85.0	66
3	T4, T0, T5	113.7	57
3	T1, T4, T5	126.7	68
2	T5, T1	174.5	36

to the on-line placer considering the number of placed tasks, manhattan distance per task and device utilization.

Table 8: Comparison of on-line and off-line placer for TS5

Placer	Tasks	Manhatten distance [hops]	Utilization [%]
Off-line	T1, T2, T3, T5	99	83
On-line (best)	T3, T1, T5	85	66
On-line (worst)	T5, T0	136.5	26
On-line (average)	-	121.3	55

While the off-line placer is able to place an additional task and therefore has an increased device utilization of about 25 %, the average manhattan distance per task is increased by 16 %. Because the tasks consists of one or more components, the manhattan distance has to be considered per component or per connection and not per task in order to get realistic results. The manhattan distance for tasks with many components can be significantly higher compared to tasks with only one component, even if the distance per connection is smaller. So, when looking at the average communication distance per connection (communication with interfaces and communication between components) the off-line placer is able to place an additional task t_2 with four components and eight connections. The manhattan distance per connections is only increased by 4 %.

Comparison of the off-line and on-line placer emphasizes the large potential for improvement of the on-line placer. Integrated module selection and relocation discussed in this chapter are two methods to increase performance of the on-line scheduler.

3.6 Chapter conclusion

In this chapter communication-aware scheduling of tasks on a reconfigurable device has been presented. A basic on-line scheduler was extended by integrated module selection and task relocation during runtime. Reduced rejection rates and communication distances of tasks resulted in increased performance. Furthermore, the influence of dynamic placement and removal of tasks on the actual path length was evaluated and the on-line placement step was compared with an off-line placer in order to evaluate the quality of the on-line method. The problem of calculating the optimal placement for a given task set is NP-hard so instead of calculating the optimal placement, in most test cases only a feasible solution was calculated. However, these solutions still surpass the on-line placer.

This chapter has taken a first step towards more resource efficient communication by looking at concepts to increase performance of the reconfigurable system at runtime. In the following chapter the architecture of the reconfigurable device used in this work and the supportive usage of the computational and communication resources is described.

4 Resource efficient DyNoC architecture

The previous chapter covered improvements to resource management of a network-based reconfigurable system. However, the actual architecture of such a system was not described in detail. In this chapter the DyNoC architecture is explained and example implementations focusing on communication and resource usage of both router and processing element are presented.

4.1 System architecture

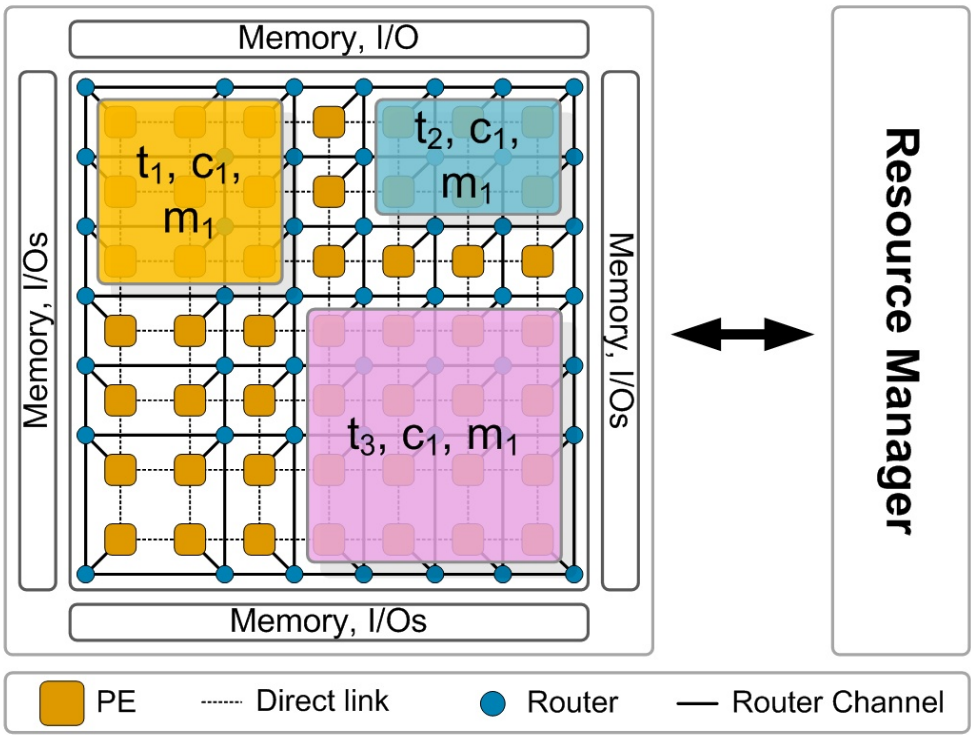

Figure 20: DyNoC system overview with three tasks implemented by three modules and a resource manager

In Figure 20 the system overview of the DyNoC is depicted showing reconfigurable device and resource manager. The DyNoC is a Network-on-Chip whose structure can be changed dynamically at runtime. It is build up of processing elements and corresponding routers. All routers are connected to neighboring routers in north, south, east and west direction and are used to establish a packet-based communication network between PEs, I/Os or memory. Tasks are implemented in the form of modules covering at least one processing element, but can also cover several PEs and routers, as visualized in Figure 20. Modules can always access the packet-based network using the top right router. However, communication inside a module uses horizontal and vertical direct-links between PEs instead, resulting in a hybrid communication scheme and the disadvantage of routers inside modules being unused resources.

Because communication between modules is established at runtime and the configuration of the DyNoC architecture is not known a priori, all-time reachability of all modules and pins of the network has to be ensured. This is achieved by having a ring of routers surrounding each placed module [9]. All modules need to be synthesized in this manner. Additional information about the DyNoC can be found in [9, 10, 11].

With this chapter focusing on resource efficient communication, the data exchange between PEs inside a module, between modules, as well as resource usage is examined in detail and implementations of a processing element and a router are presented.

4.2 Router

This chapter is a modified version of [MB10]. As mentioned before, the modules of a DyNoC can cover several routers and PEs. Inside a module, direct wiring between PEs is the main communication scheme. Only one router is used to access the packet-based network while the other routers inside the module remain inactive. Obviously, communication is not resource efficient in this regard. This chapter prospects how to use these available router resources inside modules and briefly describes data exchange over packet-based networks.

Routers generally feature three main components. A crossbar-switch which connects multiple inputs to multiple outputs, a set of buffers to temporarily store packets or parts of packets and a controller managing the forwarding of packets according to their destination. In Figure 21 a generic router and these components are shown.

Several options to utilize these components as additional resources in a DyNoC come to mind: First, the crossbar-switch could be used as an additional communication resource between PEs. Second, buffers could serve as random accessible memories, e. g. implementing lookup-tables. Third, the controller could be replaced by a small processor running all control functionality as software thus enhancing the router capabilities to

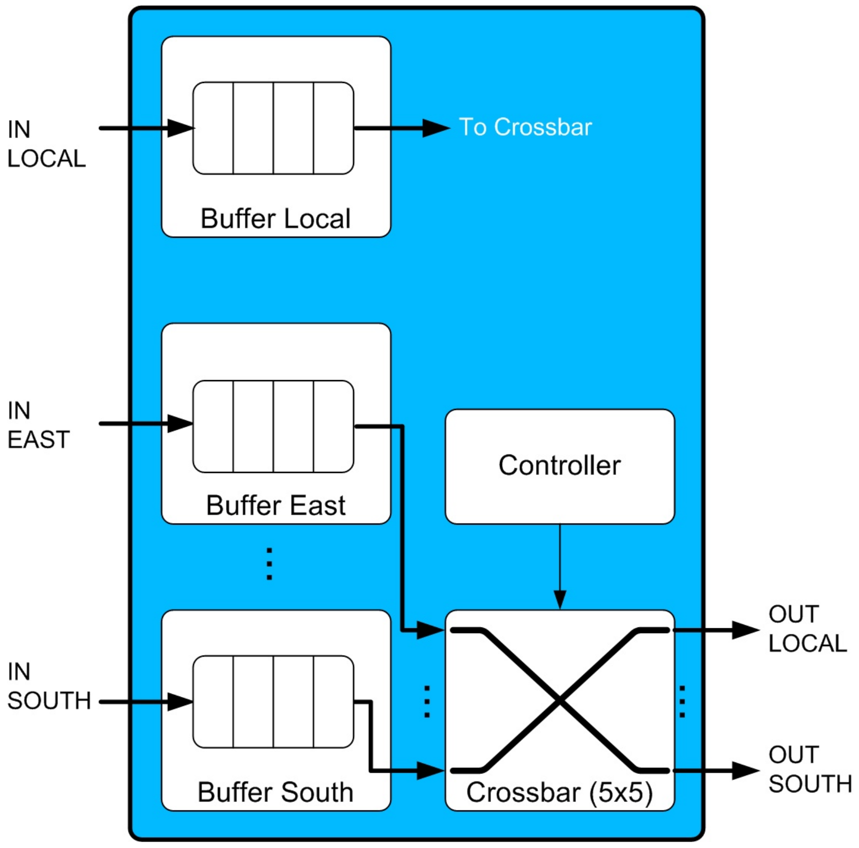

Figure 21: Router architecture with five inputs and outputs

those of a co-processor. While a co-processor would provide great flexibility, it also has some major drawbacks. Besides the resources lost for implementation of the processor, additional memory to store instructions is needed (when in router mode) yielding higher demand of area. Reconfiguration in this case would be performed by exchanging software making the process more complex, while increasing the amount of reconfiguration data and time significantly. Due to these considerations, the co-processor approach is not further explored in this work.

The following sections present a flexible and reconfigurable router exploiting crossbar-switch and buffers as additional resources to increase performance in DyNoC systems.

4.2.1 Router Architecture

In Figure 22 a simplified view of the proposed reconfigurable router architecture is given. The input channels for the local, east and south port and the output channels for the local and south port can be seen. Compared to a generic router (see Figure 21), the main differences of the reconfigurable design are the extended control logic to manage the different configurations (configuration controller) and the capability to randomly access input buffers (memory management). Additionally, the routing protocol has been enhanced to adapt to the changing application placement of DyNoCs by implementing the S-XY routing algorithm (see Section 4.2.1) with router guiding. Router guiding further improves the performance of S-XY routing by guiding packets around modules.

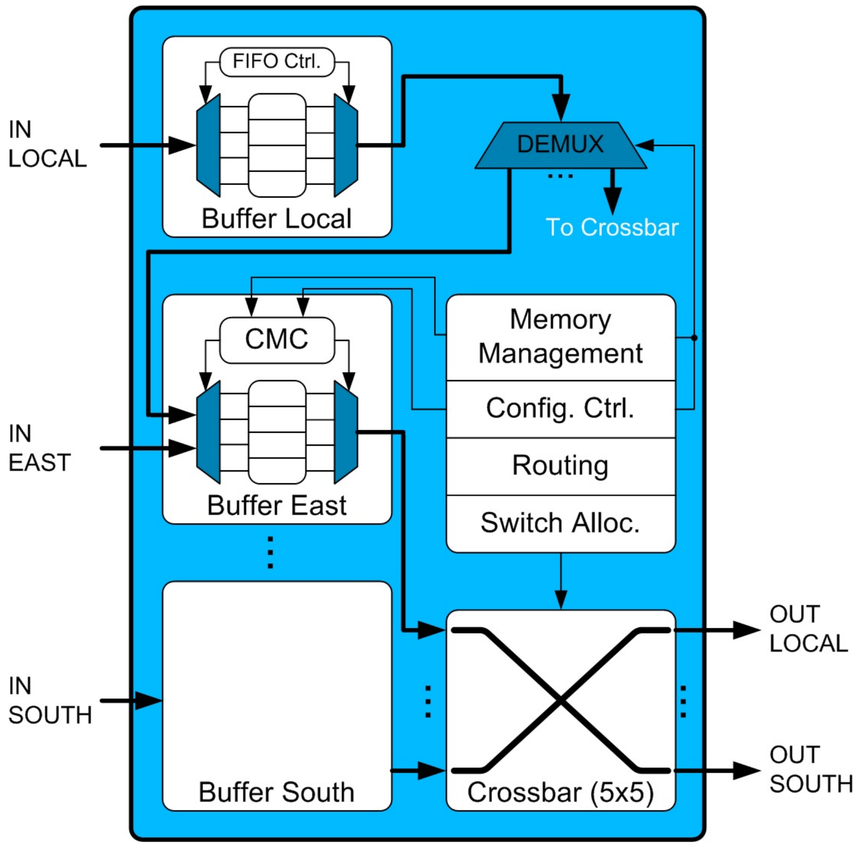

Figure 22: Simplified reconfigurable router architecture

The router can either provide router functionality or can be used as a resource inside a module running the following configuration modes: the buffers serve as a lookup table or as write- and readable memory or the switching matrix can be used for intra module communication by enabling a fixed path between PEs.

The buffers assigned to the (input) ports east, west, south and north are accessed like dual-ported RAM (DPRAM). The input port of DPRAM is designated as a writing port and is connected to the local port on the one hand and to the neighboring routers on the other. So these buffers can be written to by the local PE, if used as a resource inside a module and by neighboring routers, if in basic network mode. The second port of the DPRAM is used for reading buffers and is connected to the crossbar switch. The buffers are controlled by a Configurable Memory Controller (CMC) which manages the different access modes.

The different buffer access modes are explained in the following. In LUT mode, a 8-, 16- or 32-bit word can be looked up, so that a computation is replaced by a simple read during runtime. To fully utilize the limited amount of available memory four 8-bit words, respectively two 16-bit words can be stored in one 32-bit memory column. All the words to lookup have to be written to memory during configuration. The LUT mode supports read bursts. When in RAM mode, the buffers of the router are used as write- and readable memory — an address followed by a 32-bit data word is sent form the local PE for writing, while only an address is sent for reading.

The east, west, south and north buffers of the router are provided as contiguous memory (virtual memory) to the attached PE. As a read/write in LUT or RAM mode is initiated, a memory management unit (MMU) is used to calculate the actual physical addresses from the virtual ones and then addresses the corresponding CMC of the buffer. Up to four physically fragmented buffers of one router can be used as one contiguous memory resource by the local PE to simplify external access to the buffers. Because configuration is initiated over the local port using configuration packets the path from PE to router is fixed and the local buffer of the router can not be used as a resource inside a module, but provides FIFO implementation only. All five ports (local, east, west, south and north) of the router are 32-bit wide and all five buffers allow simultaneous reads and writes. The corresponding controllers operate independently and in parallel, both critical for low latency.

A dedicated path between routers inside a module (intra module communication) can be established by fixing the in and outputs of the switching matrix and corresponding ports, so that no routing and switch allocation is required. Data can be send directly between two non-neighboring PEs inside a module increasing complexity of modules. This feature is called tunneling, as a fixed path between two PEs is created.

In basic operation mode the router is used for packet routing through the network and operates by the concept of wormhole switching (or wormhole flow control). Wormhole switching offers some advantages over other switching techniques like cut-through, store-and-forward or circuit switching. While cut-through and store-and-forward switching allocate buffers and channel bandwidth to packets, wormhole switching allocates both of these resources to smaller pieces of a packets called flits (flow control digits) resulting in an overall smaller footprint of the router. Circuit switching on the other hand is a bufferless switching technique. Circuits between two elements are established before actual communication happens. Circuit switching raises some drawbacks in reconfigurable designs though, like long communication delay or the exclusive use of chip space [9, 21].

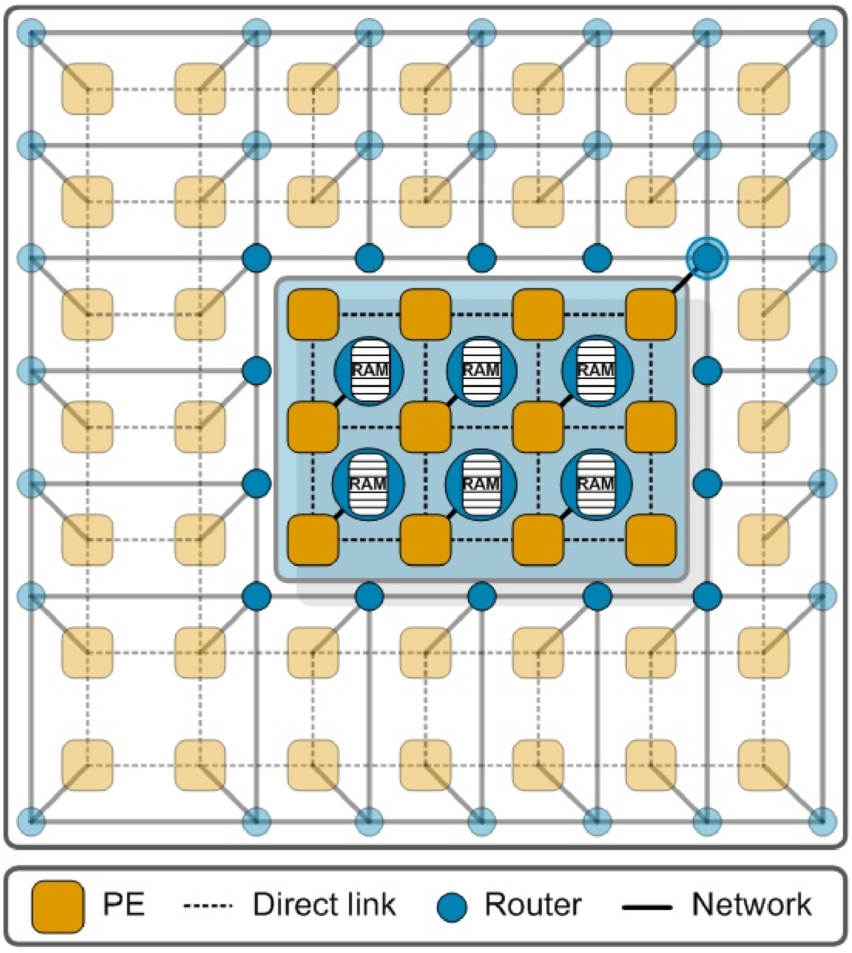

Figure 23: Module using reconfigurable routers as memory resource

Figure 23 points out an exemplary mapping of a module using reconfigurable routers as resources. Only six of these routers can be used as resources inside the module, the other six routers are part of the all-time reachability ring around the module and are dedicated to packet-based communication. In the state shown, the routers provide additional memory for the local PEs.

Reconfiguration Reconfiguration is triggered by configuration packets send over the local channel to the router. Reconfiguring the router takes less than 32-bits, as this is enough to distinguish between several coarse-grain configurations. In LUT mode however, the lookup results need to be written to memory during configuration, significantly increasing the amount of configuration data. Accessing the router resources always involves three kinds of packets. They are either configuration packets, memory access packets (LUT, RAM) or (basic) communication packets. An adaptive header is used which consists of a static part and a configuration-dependent part. The static part is used to differ between configuration packets, memory access packets and communication packets, while the configuration-dependent part of the header holds information about packet size, address, configuration type or tunneling source and destination port. The header of communication packets holds information about size and destination. The information of the configuration packets is stored in configuration control registers inside the router.

When a configuration is triggered, it has to be guaranteed that no packets present in the buffers of the router are erased. Packets in memory need to be processed before the memory can be configured. Simultaneously, router guiding is enabled to lock the port to neighboring routers which then send packets around the busy router.

4.2.2 Evaluation

Evaluation of the configurable router is done in terms of area, size and latencies. A case study shows the feasibility of this resource usage by means of a real life example.

Synthesis Results For purpose of comparison two routers have been implemented: a generic five port router (Generic Router) using wormhole switching and XY routing and a reconfigurable router (Reconfig. Router) using wormhole switching and S-XY routing. Both routers were implemented using Handel-C, whereas special coding techniques and a self-made compiler introduced by Middendorf et al. [48] came to use. In table 9 differences between the routers are shown. Synthesis results are given for a Xilinx Virtex 5 LX110 FPGA using Xilinx XST synthesis.

Table 9: Synthesis results of routers for Xilinx Virtex 5 LX110 FPGA

	Slice LUTs	Slice Registers	Frequency [MHz]
Reconfig. Router	3994 (5%)	2246 (3%)	113.1
Generic Router	2701 (3%)	1542 (2%)	176.7

Each of the five buffers of the router has 1024 bits (32 × 32-bit) of memory. Compared to the generic router the reconfigurable router occupies about 48 % more LUTs and 45 % more registers, which leads to a total increase of about 47 %. This increase in size compared to the generic router implementation is caused by the extended control logic for configuration and random memory access, as well as the implementation of the S-XY routing algorithm which is more costly than XY routing. Also, a 32-bit 1-to-4 demultiplexer is needed to access all input port buffers over the local port.

Timings The time needed to configure the router is low, one of the advantages of coarse-grain reconfiguration. In Table 10 all timings in number of cycles are given including timings for processing and reconfiguration.

Equation 7 describes the amount of time T_{CONF} needed for every configuration. No packet is supposed to be destroyed during reconfiguration, so packets in memory need to be routed before the buffer can be configured; this is indicated by T_{RESET}. This value is strongly related to packet size and current network load. An amount of time T_{SETUP} is required to read the first header flit, extract all necessary information and update the configuration control registers. T_{GUIDE} is the time used for activating or deactivating router guiding.

$$T_{CONF} = T_{RESET} + T_{SETUP} + T_{GUIDE} \tag{7}$$

T_{CONF} specifies the basic time for every configuration. For the different modes configuration time varies indicated by T_{TTC} (see Equation 8). T_{LUT} is the time needed to fill the memory with i 32-bit words in LUT mode and T_{RAM} the time to configure the router for RAM mode. Apparently switching to RAM mode does not introduce an additional overhead. T_{COMM} is the number of cycles taken to set up the router for communication by enabling tunneling mode or basic router mode.

Table 10: Router Timings

	Min. # Cycles	Max. # Cycles
T_{LUT}	3	5
T_{RAM}	0	
T_{COMM}	1	
T_{GUIDE}	1	
T_{RESET}	4	traffic and size dependent
T_{SETUP}	2	
T_{LOOKUP}	$5 \cdot i$	$6 \cdot i$
T_{WRITE}	3	5
T_{READ}	4	6
$T_{PAYLOAD}$	i	
$T_{ROUTING}$	1	3 (obstacle)

$$T_{TTC} = T_{CONF} + \begin{cases} T_{LUT} \\ T_{RAM} \\ T_{COMM} \end{cases} \tag{8}$$

In equation 9 the time needed by the local PE to access the router T_{TTA} is given. Again, a time T_{SETUP} is needed to read the first header flit and to check if reconfiguration needs to be performed. Additionally all necessary information is extracted from the header flit (packet size and destination). Timings for looking up words T_{LOOKUP} and for random access of the memory T_{WRITE} and T_{READ} are listed, whereas a burst mode is currently only available for look-ups (see Table 10). When in basic communication mode, the time for calculating the route and switching of data is given by $T_{ROUTING} + T_{PAYLOAD}$, the latter is dependent on the amount of user data i. For tunneling $T_{ROUTING}$ is zero, as the switching matrix is fixed during configuration time and no route has to be calculated.

$$T_{TTA} = T_{SETUP} + \begin{cases} T_{LOOKUP} \\ T_{WRITE} \\ T_{READ} \\ T_{ROUTING} + T_{PAYLOAD} \end{cases} \tag{9}$$

Case Study A 3×3 DyNoC was implemented on a Xilinx Virtex 5 LX110 FPGA using configurable routers and MicroBlaze [73] soft-core processors as processing elements. The processors are clocked at a frequency of 125 MHz and were connected to the routers running at 50 MHz using an asynchronous Fast Simplex Link (FSL) direct link. However, in this case study only one router and MicroBlaze of the 3×3 DyNoC is considered and the LUT mode of the router is evaluated by comparing the time to calculate the value of a sine using the MicroBlaze processor, to the time to lookup the result using the buffers of the router as a LUT. Taylor series are used to approximate the values of the sine in the domain $[0, \pi/4]$. To calculate the sine for other values, periodicity and symmetry of the sine can be used. The series is calculated for the first four terms and is accuracy to about five decimals.

Table 11: Comparison of lookup and computation of the sine signal

	Cycles (@125MHz)	Speedup
MicroBlaze	17954	1
Lookup + Conf.	1115	16,1
MicroBlaze + FPU	583	30,8
Lookup	57	315

The calculation time necessary with a MicroBlaze processor is compared to the lookup of the sine value with a router by using a hardware timer to measure the amounts of cycles. The hardware timer was attached to the MicroBlaze using FSL. Results are given in table 11 showing the speedup in relation to the computation by a MicroBlaze with and without additional floating point unit (FPU). Also a lookup including prior configuration of the router with 64×32-bit words is given. Looking up results is significantly faster than calculating the results in this case. Using buffers as lookup tables is efficiently feasible and incorporating routers inside a module as resources provides a way to increase performance of the system.

4.3 Processing element

In chapter 2 the abstract resource model for a reconfigurable computing systems was described, whereas modules were constructed from processors, memories, local interconnects and/or hardware accelerators. In a DyNoC each module is based on processing elements (and routers). Processing elements can be of different types, among them general purpose processors, domain specific processors and hardware accelerators being the most common in DyNoCs. In the previous case study the MicroBlaze GPP was used as a PE. In contrast, the implementation of a hardware accelerator for Multiply-Accumulate (MAC) operations is illustrated in this section. MAC is a common operation used by matrix multiplication, FFT and finite impulse response filters (FIR) and is shown in Equation 10. In this work only the Fast Fourier Transformation algorithm is implemented.

$$a = b + c \cdot d \tag{10}$$

The FFT was published by Cooley and Tukey in 1965 [15] and is an algorithm to calculate the discrete fourier transformation which transforms values of a function in the time domain to the corresponding values in the frequency domain using the divide and conquer paradigm. A radix-2 FFT calculates n output words from n input words in $log_2(n)$ stages while at each stage $\frac{n}{2}$ butterfly operations are executed. The butterfly operation is shown in Equation 11 with the trigonometric constants ω being the so called twiddle factors (Equation 12).

$$y_0 = x_0 + x_1\omega^k, y_1 = x_0 - x_1\omega^k \tag{11}$$

$$\omega = \exp\left(-\frac{2\pi i}{n}\right) \tag{12}$$

4.3.1 Architecture

Figure 24 depicts a simplified representation of a scalable reconfigurable processing element. The PE's four main components are MAC-element, local memory, crossbar switch and control logic (configuration and data-flow controller).

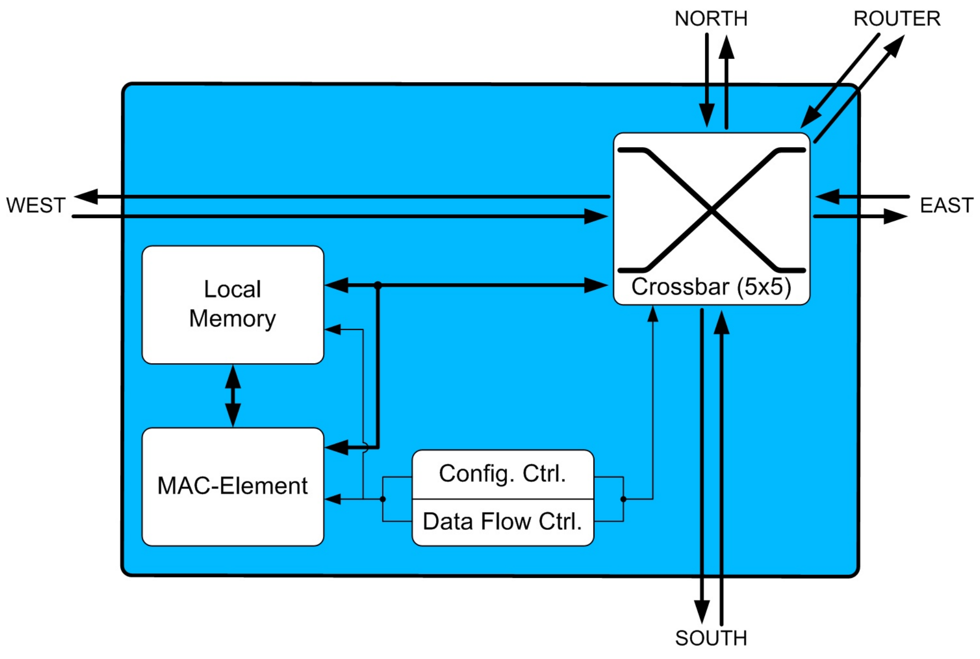

Figure 24: Simplified processing element architecture with five inputs and outputs for the calculation of the FFT algorithm

Input data for the calculation of the FFT is sent over the packet-based network to the processing element. After processing of the input data the results are sent back using the packet-based network. Besides input values for the FFT, twiddle factors are needed for the calculation. A radix-2 FFT with 1024 input words uses 512 twiddle factors. These complex input and ω values need to be stored in local memory of the PE prior to the calculation. When both input data and twiddle factors are available, the PE calculates each stage of the FFT one after another.

When m PEs are joined to calculate the FFT in cooperation, a direct connection between neighboring PEs has to be established by means of the crossbar switches. In this setting the twiddle factors are send to each PE of the module and the input data is splitted equally between the PEs. Each processing element calculates $\frac{n}{2m}$ butterfly operations per stage and forwards the results at stage $log_2(n) - log_2(m)$ and onwards to other PEs of the module for calculation of subsequent stages. After the successful calculation of the final stage, the results are sent out by the network router of the module.

Figure 25 shows the data flow from input to output for the calculation of a 16-point-FFT with four PEs.

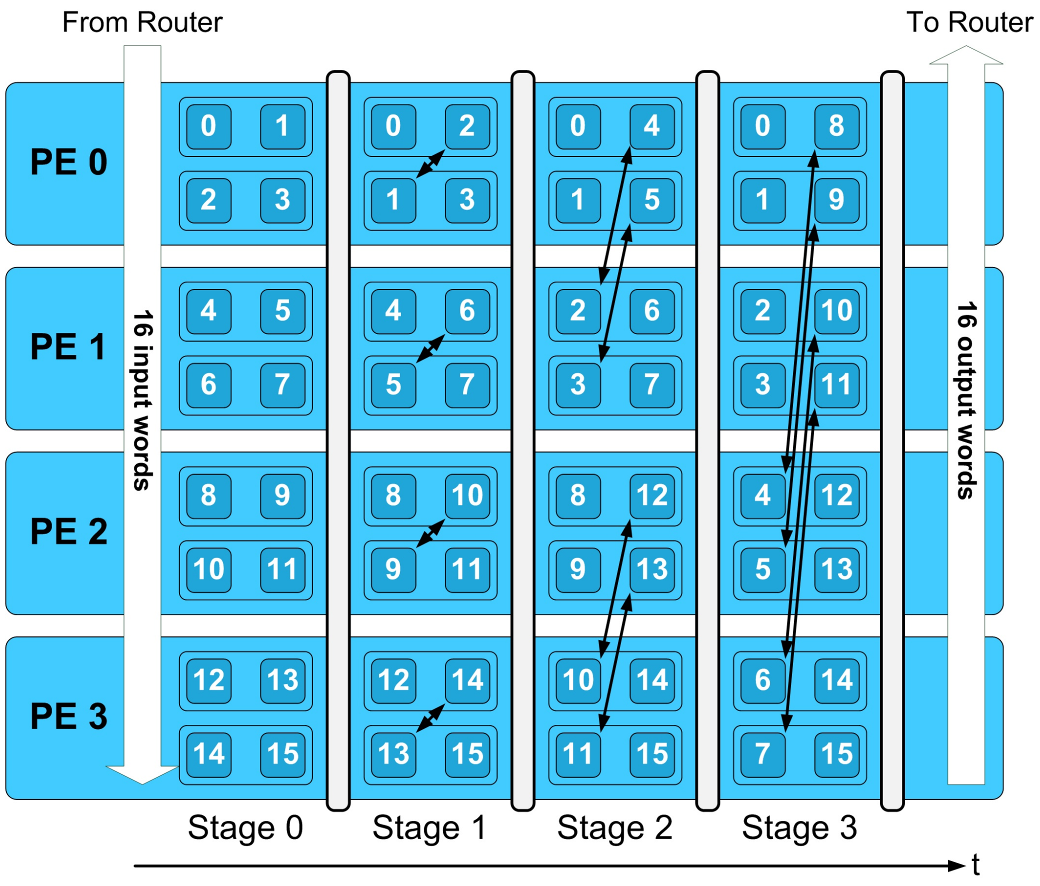

Figure 25: FFT data flow on a module consisting of four PEs for 16 input words

Figure 26 shows module shapes created by interconnecting several PEs in different ways. Note that only the router of the packet-based network interacting with the module is present in the figure.

Usage of router resources The previous chapter proposed to use routers inside a module as additional resource. The implementation of the processing element discussed here does not incorporate this concept. However, several possibilities exist to support the calculation of the FFT by using routers inside of modules. The buffers of the router offer a

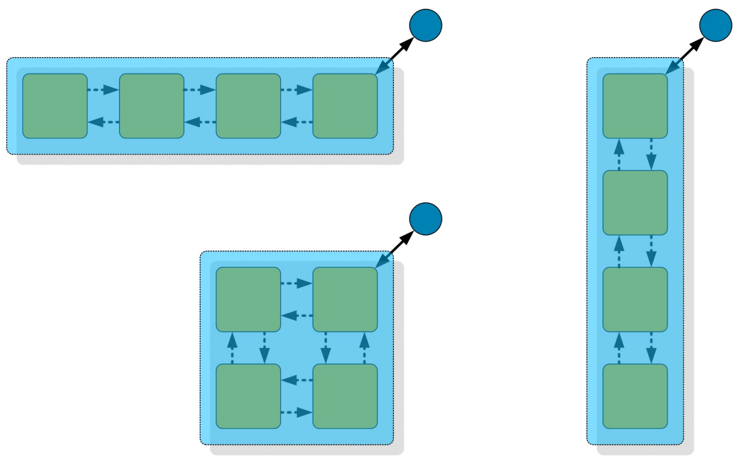
Figure 26: Different shapes of modules build-up of four PEs

way to extend the local memories of PEs. For once, this can lead to PEs being able to compute FFTs with more input words and furthermore it allows PEs to be constructed with less local memory, reducing the resource demand of the PE. The switching matrix of the router inside a module can be used to exchange intermediate values between (non-neighboring) PEs. During the calculation of each FFT stage the communication path between PEs can be adapted, leading to faster intra module communication.

Reconfiguration Similar to the configuration of the router, the processing element is configured by a configuration word with less than 32 bits. Only information about the amount of input words, the designated position of the PE inside a module and the connection to the neighboring PEs are relevant and required for configuration. So, local memory, MAC element, data flow controller and crossbar switch are configured according to the configuration word.

4.3.2 Evaluation

This section evaluates the performance of the scalable processing element at the calculation of the FFT algorithm by checking synthesis results and case study results comparing the PE to a PC-based system.

Synthesis Results Synthesis results of a Virtex 5 LX110T FPGA are listed in Table 12. Each PE needs to store the twiddle factors ω and input values for each butterfly operation. Six 32-bit wide dual-ported RAMs were connected together to build three 64-bit wide dual-port memories to store the complex twiddle factors, as well as the complex input and output values of the butterfly operations.

Table 12: Synthesis results of PE for Xilinx Virtex 5 LX110 FPGA

Slice LUTs	Slice Registers	DSP48E	BRAM (36kBit)	Frequency [MHz]
8952 (12%)	4058 (6%)	8	6	89

In addition, 128-bit wide direct connections between the PEs were used to communicate with neighboring PEs in order to transfer two complex values in parallel. This leads to large crossbar switches due to the high resource demand of multiplexers but also offers high bandwidth leading to enhanced performance of the PE.

Case Study The performance of the processing element was evaluated by calculating the FFT for varying amounts of input data and PEs. Up to four processing elements were used to calculate the FFT for 16 to 1024 input values. A MicroBlaze processor was used to configure the PEs and a hardware counter came to use for precise measurement of timings. Table 13 shows the results of this setup.

Table 13: Computation time and speed-up of the FFT algorithm with several processing elements and varying amount of input data

N-point FFT	1 PE [ns]	2 PEs [ns]	4 PEs [ns]	GPP [ns]
16	4660 (1)	4760 (0.98)	5020(0.93)	22178 (0.21)
32	6300 (1)	6006(1.05)	6020 (1.05)	21629 (0.29)
64	9220 (1)	7960 (1.16)	7420 (1.24)	25077 (0.36)
128	15020 (1)	11520 (1.30)	9700 (1.54)	25534 (0.58)
256	27200 (1)	18600 (1.46)	13900 (1.96)	29410 (0.92)
512	53500 (1)	33360 (1.60)	22260 (2.40)	33833 (1.58)
1024	110500 (1)	64760 (1.71)	39580 (2.79)	49207 (2.24)

For an FFT with 1024 points a speed-up of 2.79 was achieved when using four PEs instead of one. Because communication between processing elements is a limiting factor, communication overhead reduces the theoretically possible speed-up. Compared to the results of an AMD Phenom II X6 processor clocked at 3.2 GHz the PEs provide a solid speed-up. However, larger amounts of input values shrink the advantage of the PEs due to the resulting communication overhead. Note that the Phenom was running Linux and the GNU scientific library (GSL) for the calculation of the FFT.

4.4 Chapter conclusion

In this chapter an implementation of the DyNoC architecture was discussed and the communication and computational resources in form of routers and processing elements were introduced.

The flexible DyNoC router can be configured to reuse the buffers or switching matrix of the router for computation and communication inside a module. Evaluation of the router in terms of area and timings was given and a case study encouraged the usage of the router to increase performance.

With the implementation of a scalable and configurable processing element for the FFT algorithm modules can be build-up from neighboring processing elements. Evaluation showed that a speed-up of 2.79 for the calculation of a 1024-FFT can be achieved when letting four processing elements cooperate, compared to the calculation using only one PE.

The chapter focused on flexibility and performance enhancements of reconfigurable systems by efficient usage of the communication and computational resources. The following chapter explores efficient high-level communication. A high-level communication protocol for general purpose processors is presented allowing flexible and resource efficient communication on network-based reconfigurable on-chip systems.

5 High-level communication in reconfigurable on-chip systems

Up until now this paper focused on communication between applications or tasks implemented as hardware-only processing elements. Programmable general purpose processors running software were only considered marginally. However, irregular control-flow dominated applications are best implemented in software, while data-flow dominated algorithms, like the fast-fourier-transformation presented in the previous chapter, can be efficiently implemented as dedicated hardware. The Pact XPP-III architecture for instance combines coarse-grain reconfigurable logic blocks with some programmable processors [54]. In a DyNoC design, processing elements can also be programmable processors. In order for programmable processors to communicate, a protocol, which on the hand provides an interface for applications and on the other hand provides access to the interconnection network, is vital. The software protocol must be lightweight to fit in the limited amount of memory available on-chip and should not introduce significant computing overhead.

Message Passing Interface (MPI) [47] is a well known interface suited for communication in multiprocessor systems. A programmer basically calls *send* and *receive* functions to communicate with other processors among the system. MPI is commonly used in super computers or computer clusters and supports different communication operations like point-to-point, broadcast, or scatter and gather. The use of many MPI functions leads to the necessity of a large library. This can become a severe problem in on-chip multiprocessor systems, due to the limited amount of available on-chip memory — especially so in FPGA-based systems. Ensuring that most data accesses are satisfied from on-chip memory is a critical problem, as the cost of an off-chip access can be very high [52, 34]. A configurable library for reconfigurable on-chip systems tackles this problem by including only low-level functions for the currently required interconnection networks and the needed MPI functionality. Such a library was implemented and named SoC-MPI.

The following sections are a modified version of [MLIB08] and structure and features of the SoC-MPI library as well as results are given. The SoC-MPI library is not limited to a certain interconnection network, like the one of the Dynamic Network on-Chip is, but supports different interconnection networks like buses, stars and rings. It provides a generic approach to high-level communication in on-chip multiprocessor systems.

5.1 Structure and Functionality

In order to achieve flexibility the library allows the adaption to the applications' needs and to the interconnection network. For this purpose it is split into a Network Independent Layer (NInL) and a Network Dependent Layer (NDeL). Figure 27 displays the structure of the SoC-MPI library.

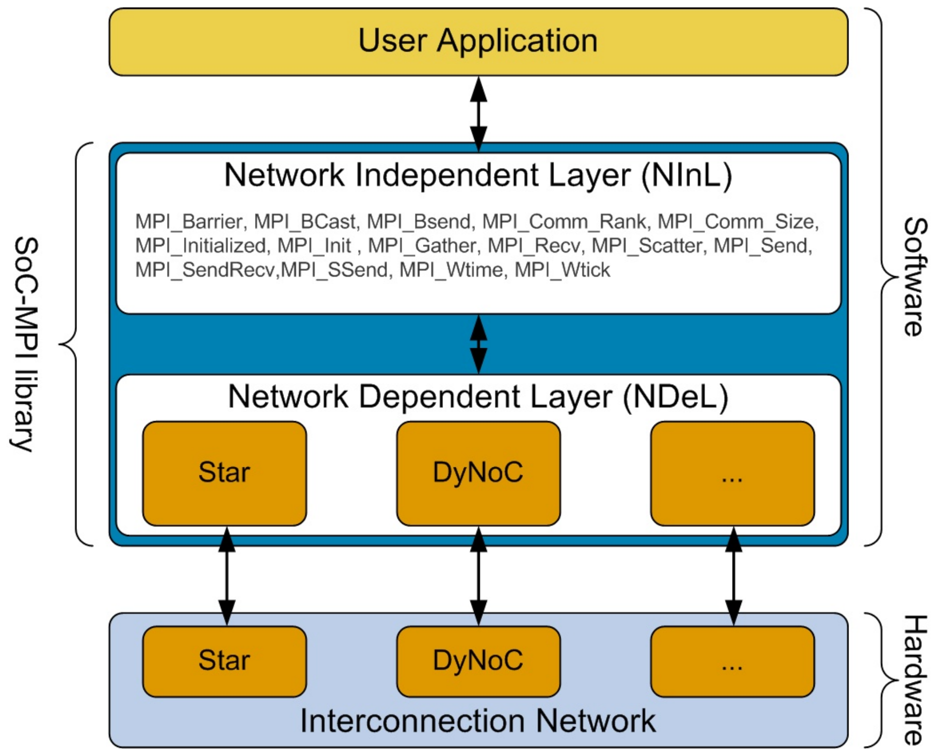
User Application
Network Independent Layer (NInL)
MPI_Barrier, MPI_BCast, MPI_Bsend, MPI_Comm_Rank, MPI_Comm_Size, MPI_Initialized, MPI_Init , MPI_Gather, MPI_Recv, MPI_Scatter, MPI_Send, MPI_SendRecv,MPI_SSend, MPI_Wtime, MPI_Wtick
Network Dependent Layer (NDeL)
Star
DyNoC
...
Star
DyNoC
...
Interconnection Network
SoC-MPI library
Software
Hardware

Figure 27: SoC-MPI Layer Structure

The MPI functions are implemented in the Network Independent Layer. The Network Dependent Layer holds the basic communication functions and implements the drivers for the aspired interconnection network. Also conversion of MPI specific ranks to hardware addresses and vice versa is done in this layer.

Due to the separation of the library into network dependent and independent parts, it is possible to port the library to different platforms within a minimal amount of time. The SoC-MPI library supports a subset of the MPI-1 standard. All available MPI functions are listed in Table 14.

It is possible to have several interconnection networks and different communication schemes in one MPI session. The programmable processing elements do not need to

Table 14: Supported MPI functions

`Function`	Description
`MPI_Init`	Initializes SoC-MPI
`MPI_Finalize`	Cleans up SoC-MPI
`MPI_Comm_rank`	Gets the own MPI_RANK
`MPI_Comm_size`	Gets the size of the set of nodes
`MPI_Send`	Sends a message
`MPI_RSend`	Sends in ready mode
`MPI_BSend`	Sends in buffered mode
`MPI_SSend`	Sends in synchronous mode
`MPI_Recv`	Receives a message
`MPI_SendRecv`	Combines a Send and a Recv call in one call
`MPI_Gather`	Collect multiple message segments into one
`MPI_Scatter`	Delivers message segments to multiple receiver
`MPI_Barrier`	For synchronizing a set of nodes
`MPI_Bcast`	Broadcasts a message to a set of nodes
`MPI_Wtime`	Provides a timer
`MPI_Wtick`	Gets resolution of the timer
`MPI_Pack`	Packs multiple messages in one
`MPI_Unpack`	Unpacks one message to multiple ones

support every interconnection network and specific message passing operation like buffered or non-blocking send (Figure 28). The library is configurable to keep it small and precisely tailored according to requirements. The configuration phase is splitted in a global and a local part. The global configuration is done for each interconnect network in the system. The hardware addresses of each processing element in the network is defined and further requirements and features like the maximum packet size of the network or hardware supported features are specified. The local configuration is done for each processing element of the interconnect; during the local configuration the MPI ranks of the processing nodes are set and the connected network addresses and required parameters like memory addresses are determined.

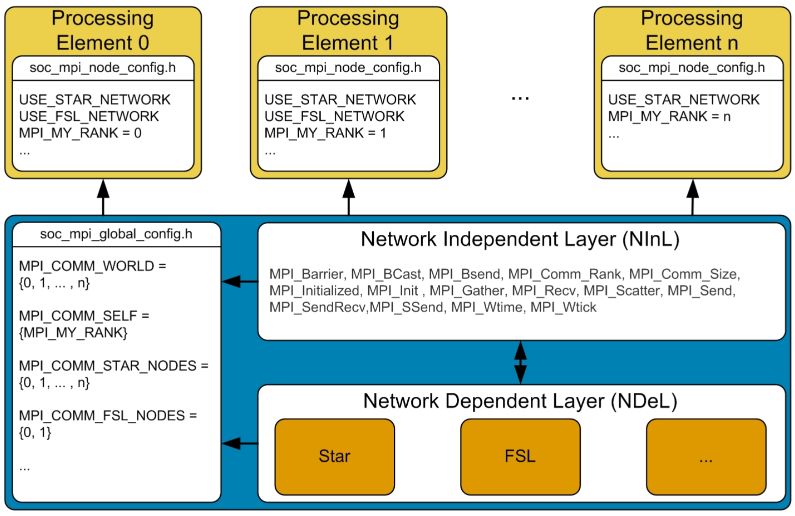
Processing Element 0
soc_mpi_node_config.h
USE_STAR_NETWORK
USE_FSL_NETWORK
MPI_MY_RANK = 0
...
Processing Element 1
soc_mpi_node_config.h
USE_STAR_NETWORK
USE_FSL_NETWORK
MPI_MY_RANK = 1
...
...
Processing Element n
soc_mpi_node_config.h
USE_STAR_NETWORK
MPI_MY_RANK = n
...
soc_mpi_global_config.h
MPI_COMM_WORLD = {0, 1, ... , n}
MPI_COMM_SELF = {MPI_MY_RANK}
MPI_COMM_STAR_NODES = {0, 1, ... , n}
MPI_COMM_FSL_NODES = {0, 1}
...
Network Independent Layer (NInL)
MPI_Barrier, MPI_BCast, MPI_Bsend, MPI_Comm_Rank, MPI_Comm_Size, MPI_Initialized, MPI_Init , MPI_Gather, MPI_Recv, MPI_Scatter, MPI_Send, MPI_SendRecv,MPI_SSend, MPI_Wtime, MPI_Wtick
Network Dependent Layer (NDeL)
Star
FSL
...

Figure 28: Simplified SoC-MPI Configuration Scheme

5.2 Evaluation

For the evaluation of the SoC-MPI library several benchmarks and a case study were carried out. The library currently supports four interconnection networks: FSL, PLB bus, a star network and a ring. The smallest version of the library uses 11056 bytes when implementing a star topology as the communication network and the minimal set of needed MPI functions. The current full version has a size of 16124 bytes. This version supports four different network types and the MPI functions as seen in table 14. In both cases compiler optimization was deactivated.

5.2.1 Benchmarks

The SoC-MPI library is evaluated using Intel MPI Benchmarks 3.1 [36], which is the successor of the Pallas MPI Benchmarks. The PingPong, SendRecv, Gather, Scatter and Bcast micro benchmarks were carried out on a Xilinx ML-403 evaluation platform [74] equipped with a Virtex 4 FPGA. Three MicroBlaze soft-core processors [73] with 2 kBytes cache each are connected together via a star network using FSL direct links. Due to the limited amount of on-chip memory, the program data for each MicroBlaze is located in local on-chip Block RAM, while instruction data is stored in off-chip memory. The system is running at 100 MHz.

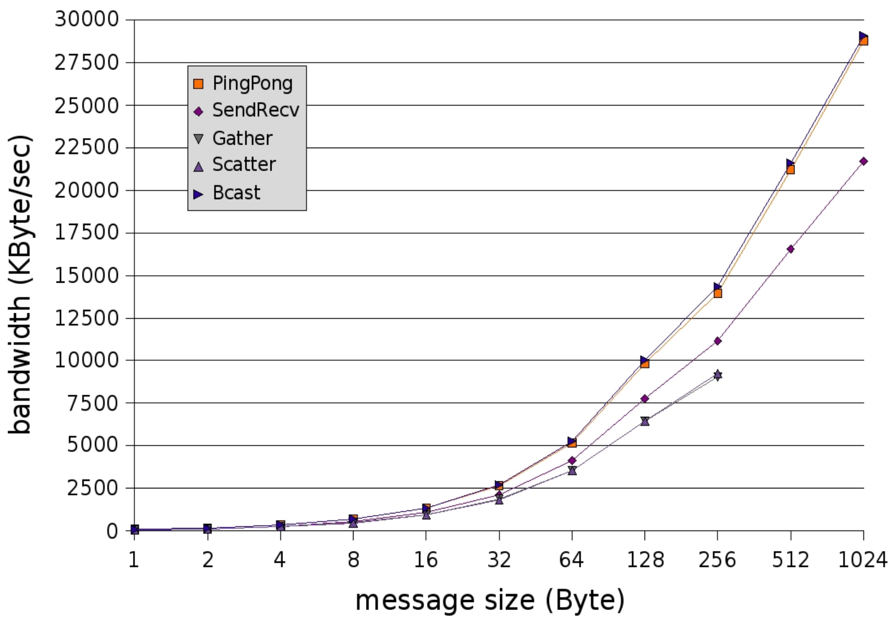

Figure 29: Benchmark Results of SoC-MPI using Intel MPI Benchmarks 3.1

In Figure 29 the results of the five micro benchmarks are shown. MPI_Wtime was used to measure the time of each benchmark. A packet consists of 5 byte header data and the actual MPI message, at a maximum length of 251 bytes here. MPI messages larger than 251 bytes need to be split into multiple messages introducing additional overhead. This is apparent from the small bandwidth drop between 128 and 256 bytes MPI message length. Not all benchmarks could be performed completely, again due to limited on-chip memory.

5.2.2 Case study

In order to test the SoC-MPI library working with a real life application, Singular Value Decomposition (SVD) was implemented. SVD executes the factorization of a $m \times n$ matrix A into a product $A = U \times \Sigma \times V^T$, where U is an $m \times n$ real unitary matrix, Σ is an $m \times n$ rectangular diagonal matrix with non negative real numbers on the diagonal, and V^T is the conjugate transpose of an $n \times m$ real unitary matrix. The positive diagonal entries of Σ are the singular values of A. SVD has many important scientific and engineering applications. However, for very large matrices, as it is the case in information retrieval, computation is very time consuming.

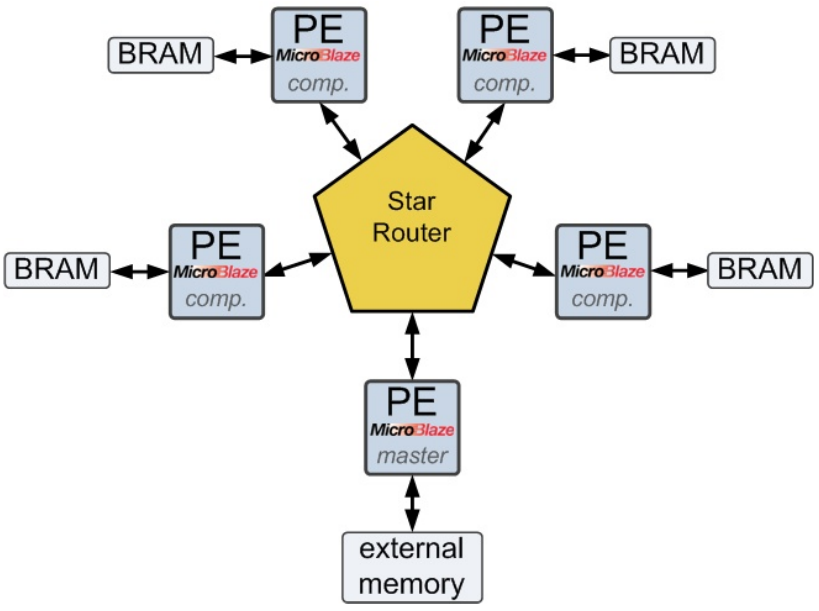

Figure 30: Star Network for Singular Value Decomposition with five MicroBlaze processing elements

Table 15: Benchmark Results of the SVD Implementation in

Matrices	1 PE [ms]	2 PEs [ms]	4 PEs [ms]
16 x 200	25	26	32
32 x 200	93	75	76
64 x 200	-	239	199

The singular value decomposition for real matrices was implemented on a ML-310 evaluation platform [76] with number of MicroBlaze soft-core processors varying from two to five. One master processor (PE master) distributes the data to the computing nodes (PE comp.). The master node therefore has direct access to an external memory. All processors are connected together via a star router, each computing node has additional cache for storing data. In Figure 30 the system used for the case study is depicted. The size of the SoC-MPI library in this case study is 13584 bytes using the *MPI_Init*, *MPI_Finalize*, *MPI_Wtime*, *MPI_Send*, *MPI_Receive* and *MPI_Barrier* functions. The total amount of slices needed for the system with five MicroBlazes was 12343 Slice LUTs (90 %) and almost all BRAM (94%) of the FPGA.

The results of the case study are shown in Table 15 for n × m matrices varying from (16 × 200) to (128 × 200) and computing nodes varying from one to four. In each case a master node is present and the maximum number of columns assigned to each processor is limited to 32 in order to fit block matrices to the local memory. The performance increases by use of multiple processors for large matrices. However, approximately 9000 cycles for sending a 128 byte packet with a MicroBlaze processor was measured, leading to the conclusion that communication with MicroBlaze limits performance.

5.3 Chapter conclusion

This chapter introduced the SoC-MPI communication library to handle high-level communication between programmable processors. The library implements a subset of the MPI standard and is configurable in terms of the used underlying interconnection networks and actually necessary MPI functionality allowing adaption to the applications' needs. This flexibility in usage allows more efficient communication between programmable processors due to low computational overhead and memory footprint.

The chapter concludes the considerations on the main aspects of holistic resource efficient communication in reconfigurable on-chip systems and is followed by summary and outlook next.

6 Conclusion and outlook

This thesis explored a holistic approach to communication in network-based reconfigurable on-chip systems. By incorporating and optimizing different parts of the reconfigurable system during design and runtime a resource efficient communication is achieved resulting in increased flexibility and performance. The resource manager, the architecture of the interconnection network, communication protocols as well as communication resource usage were pointed out as the most important communication-related parts of a reconfigurable system.

Communication-aware resource management
Starting from a basic communication-aware on-line scheduler, the resource manager of a reconfigurable device was extended to support module selection and relocation of tasks. Module selection allows the resource manager adaption of the device usage to the application by considering different implementations of the application. Several module selection strategies considering size and execution time of modules have been implemented and evaluated leading to a reduced rejection rate and increased system performance. The second extension, relocation of tasks at runtime, provides a way to optimize the placement of tasks to reduce the actual communication distance between communication partners. Selection of tasks for relocation, reordering of the selected tasks and subsequent replacement have been taken into regard. Evaluation showed up to 25 % reduction in communication distance and a decrease from 5 % to 1,5 % in rejection rate for the presented benchmarks, resulting in increased performance.

Manhattan distance is a metric used for approximating communication distance. In a Dynamic Network-on-Chip, large modules are build-up of several processing elements and routers and therefore create obstacles for packets traveling through the network. The real communication length was calculated using the S-XY dynamic routing algorithm. This resulted in an average 17 % increase of hops compared to manhattan distance for the run benchmarks.

Finally, the on-line placement step of the basic communication-aware scheduler was compared to a communication-aware off-line placer implemented in the declarative problem solving paradigm ASP. ASP enables the calculation of optimal communication-aware placements and raises device utilization by at least 25 % for the used task set.

Dynamic Network-on-Chip architecture
Next, an actual implementation of a DyNoC consisting mainly of router and processing elements was described. Resources inside the router can be used for computation and communication inside modules. The buffers and switching matrix allow the storage of data and extended interconnection inside a module by connecting non neighboring processing elements, resulting in flexible router usage and increased performance of modules. For the DyNoC different types of processing elements are possible and this thesis only explored an implementation of the FFT algorithm. Multiple PEs can be joined together, to form a module capable of parallel execution of several FFT calculations. For 1024 input words a speed-up factor of 2.79 when using four PEs compared to one could be achieved.

Communication protocol
The final section of the paper illustrated a high-level communication protocol which allows communication between software-programmable processing elements in a DyNoC system. The communication protocol uses the message passing communication paradigm. A subset of the MPI standard was implemented and is provided in the form of a configurable and lightweight library, called SoC-MPI. This flexible library offers low protocol overhead and therefore provides efficient high-level communication in network-based reconfigurable on-chip systems.

All these insights support the thesis of the work given in chapter 1. Overall, results show that performance and flexibility enhancements can be achieved by resource efficient communication. Of course, there is still potential to further increase flexibility and performance.

Future versions of the resource manager will include enhancements of communication-aware task relocation by optimized selection of the relocation task set for specific relocation of individual tasks. The relocation of a task also makes relocating data of that task necessary. This aspect is strongly related to task preemption strategy. Follow-up research will consider different kinds of preemption and strategies for relocation of data.

Furthermore, the on-line scheduler has to be extended to support heterogeneous architectures which would allow more precise modeling of reconfigurable devices. The ASP implementation of the off-line placer already support heterogeneous structures but has to be improved to find optimal solution in most cases. Therefore, on the one hand the solver parameters have to be tweaked and on the other hand the implementation has to be optimized towards a reduced set of answers.

Future developments of the DyNoC architecture will consider the integration of the router in modules and evaluation with real-world applications. New methods for reducing resource demand of the router and increase usability inside modules have to found. Finally, a DyNoC should consist of both hardware-based processing elements and programmable processors. For uniform communication between hardware and software PEs, both elements should support the message passing interface.

In summary, this thesis covers an approach to resource efficient communication in reconfigurable systems by incorporating the resource manager, the architecture of the interconnection network, the communication protocols and communication resource usage. Evaluation of the proposed methods led to increased performance and flexibility of the system. While this thesis presented a holistic approach to communication in reconfigurable systems, more research is required to improve the presented concepts and techniques. Hence, this work should motivate further research in the exciting area of communication in reconfigurable systems.

Author's Publications

[BMAI10] C. Bobda, P. Mahr, B. Andres, and H. Ishebabi. Application-driven architecture synthesis of on-chip Multiprocessor systems. In Waleed W. Smari and John P. McIntire, editors, *High Performance Computing and Simulation (HPCS)*, pages 591–598. IEEE, 2010.

[HMMB10] R. Hartmann, F. Al Machot, P. Mahr, and C. Bobda. Camera-based system for tracking and position estimation of humans. In *Design and Architectures for Signal and Image Processing (DASIP)*, pages 62–67. IEEE, 2010.

[IMB08a] H. Ishebabi, P. Mahr, and C. Bobda. Automatic Synthesis of Multiprocessor Systems From Parallel Programs under Preemptive Scheduling. In *International Conference on ReConFigurable Computing and FPGAs*, Cancun, Mexico, December 2008.

[IMB08b] H. Ishebabi, P. Mahr, and C. Bobda. Makespan Minimization in Automatic Synthesis of Multiprocessor Systems from Parallel Programs. In *IEEE International Conference on Field-Programmable Technology*, Taipei, Taiwan, December 2008.

[IMB$^+$09] H. Ishebabi, P. Mahr, C. Bobda, M. Gebser, and T. Schaub. Application of ASP for Automatic Synthesis of Flexible Multiprocessor Systems from Parallel Programs. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '09)*, volume 5753 of *Lecture Notes in Computer Science*, pages 598–603. Springer, 2009.

[MAIB09] P. Mahr, B. Andres, H. Ishebabi, and C. Bobda. A Design Methodology for Reconfigurable Heterogeneous Architectures. In *Many-Core and Reconfigurable Supercomputing Conference 2009 (MRSC'09)*, Berlin, Germany, March 2009.

[MB10] P. Mahr and C. Bobda. Reconfigurable Router for Dynamic Networks-on-Chip. In *RSP'2010: Proceedings of the 2010 IEEE/IFIP International Symposium on Rapid System Prototyping*, Fairfax, Virginia, USA, 2010.

[MBed] P. Mahr and C. Bobda. Reducing communication costs on Dynamic Networks-on-Chip through runtime relocation of tasks. In *IEEE International*

Symposium on Field-Programmable Custom Computing Machines (FCCM 2012), Toronto, Canada, 2012, submitted.

[MCHB11] P. Mahr, S. Christgau, C. Haubelt, and C. Bobda. Integrated Temporal Planning, Module Selection and Placement of Tasks for Dynamic Networks-on-Chip. In *Reconfigurable Architectures Workshop (RAW 2011)*, Ancorage, Alaska, USA, 2011.

[MHB09] P. Mahr, A. Heine, and C. Bobda. On-chip transactional memory system for FPGAs using TCC model. In *Proceedings of the 6th FPGAworld Conference*, FPGAworld '09, pages 39–43. ACM, 2009.

[MIL$^+$08] P. Mahr, H. Ishebabi, C. Loerchner, M. Metzner, and C. Bobda. Automated Design Approach for On-Chip Multiprocessor Systems. In *Proceedings of the 5th FPGAworld Conference*, FPGAworld '08, September 2008.

[MKZB10] P. Mahr, M. Krebs, C. Zemko, and C. Bobda. Transparent Energy Metering in Smart Living Environments. In *3. Deutscher AAL Kongress 2010*, Berlin, Germany, January 2010.

[MLIB08] P. Mahr, C. Lorchner, H. Ishebabi, and C. Bobda. SoC-MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips. In *International Conference on ReConFigurable Computing and FPGAs*, pages 187–192. IEEE Computer Society, 2008.

Bibliography

[1] A. Agarwal. Limits on interconnection network performance. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):398–412, 1991.

[2] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, and J. Teich. A practical approach for circuit routing on dynamic reconfigurable devices. In *In: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, pages 84–90, 2005.

[3] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, and J. C. van der Veen. Optimal Free-Space Management and Routing-Conscious Dynamic Placement for Reconfigurable Devices. *IEEE Trans. Comput.*, 56(5):673–680, 2007.

[4] A. Ahmadinia and A. Shahrabi. A Highly Adaptive and Efficient Router Architecture for Network-on-Chip. *Comput. J.*, 54:1295–1307, August 2011.

[5] K. Andres. A Texas Instruments Application Report: MOS programmable logic arrays, 1970.

[6] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. In *IEEE Design and Test of Computers*, pages 68–83, 2000.

[7] I. Belaid, F. Muller, and M. Benjemaa. New three-level resource management enhancing quality of offline hardware task placement on FPGA. *International Journal on Reconfigurable Compututing*, 2010:4:1–4:20, January 2010.

[8] L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, 35:70–78, 2002.

[9] C. Bobda. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications.* Springer Publishing Company, Incorporated, 2007.

[10] C. Bobda and A. Ahmadinia. Dynamic Interconnection of Reconfigurable Modules on Reconfigurable Devices. *IEEE Design and Test of Computers*, 22:443–451, 2005.

[11] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. P. Fekete, and J. van der Veen. DyNoC: A dynamic infrastructure for communication in dynamically reconfigurable devices. In *International Conference on Field Programmable Logic and Applications 2005 (FPL 2005)*, pages 153–158, 2005.

[12] T. Brecht. On the importance of parallel application placement in NUMA multiprocessors. In *In SEDMS IV. Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 1–18, 1993.

[13] W. Chou. Problems in the design of data communications networks. *SIGCOMM Comput. Commun. Rev.*, 4:1–6, April 1974.

[14] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen. PipeRench implementation of the instruction path coprocessor. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 147–158. ACM, 2000.

[15] J. Cooley and J. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[16] R. Dafali and J-Ph. Diguet. Self-Adaptive Network Interface (SANI): Local Component of a NoC Configuration Manager. In *International Conference on Reconfigurable Computing and FPGAs*, pages 296–301, 2009.

[17] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[18] K. Danne and M. Platzner. An EDF schedulability test for periodic tasks on reconfigurable hardware devices. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 93–102. ACM, 2006.

[19] K. Danne and S. Stühmeier. Off-Line Placement of Tasks onto Reconfigurable Hardware Considering Geometrical Task Variants. *IFIP International Federation for Information Processing*, 184, 2005.

[20] L. Devaux, S. Pillement, D. Chillet, and D. Demigny. R2NoC: Dynamically Reconfigurable Routers for Flexible Networks on Chip. In Viktor K. Prasanna, Jürgen Becker, and René Cumplido, editors, *ReConFig*, pages 376–381. IEEE Computer Society, 2010.

[21] J. Duato, S. Yalamanchili, and N. Lionel. *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann Publishers Inc., 2002.

[22] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135. Springer-Verlag, 1996.

[23] A. Eiche, D. Chillet, S. Pillement, and O. Sentieys. Task placement for dynamic and partial reconfigurable architecture. In *DASIP '2010: Proceedings of the 2010 Conference on Design & Architectures for Signal & Image Processing*, pages 642–649. IEEE Press, 2010.

[24] G. Estrin. Organization of computer systems: the fixed plus variable structure computer. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '60 (Western), pages 33–40. ACM, 1960.

[25] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. *Electronic Computers, IEEE Transactions on*, pages 747–755, December 2006.

[26] S. P. Fekete, T. Kamphans, N. Schweer, C. Tessars, J. C. van der Veen, J. Angermeier, D. Koch, and J. Teich. No-break dynamic defragmentation of reconfigurable devices. In *International Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 113–118, September 2008.

[27] A. Ferrer, S. Parkes, and P. Mendham. Quality of Service in NoC for Reconfigurable Space Applications. In *Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*, AHS '09, pages 482–487. IEEE Computer Society, 2009.

[28] P. Francesco, P. Antonio, and P. Marchal. Flexible Hardware/Software Support for Message Passing on a Distributed Shared Memory Architecture. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 736–741. IEEE Computer Society, 2005.

[29] B. Glas, A. Klimm, O. Sander, K. D. Müller-Glaser, and J. Becker. A self adaptive interfacing concept for consumer device integration into automotive entities. In *IPDPS*, pages 1–6. IEEE, 2008.

[30] Maya B. Gokhlae and Paul S. Graham. *Reconfigurable Computing: Accelerationg computation with Field-Programmable Gate Arrays.* Springer Publishing Company, Incorporated, 2005.

[31] R. H. Güting. An optimal contour algorithm for iso-oriented rectangles. *J. Algorithms*, 5:303–326, September 1984.

[32] M. Handa and R. Vemuri. An Efficient Algorithm for Finding Empty Space For Online FPGA Placement. In *DAC'04: Proceedings of the 41st Design Automation Conference*, 2004.

[33] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[34] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[35] C. Hilton and B. Nelson. PNoC: a flexible circuit-switched NoC for FPGA-based systems. *Computers and Digital Techniques, IEE Proceedings -*, 153(3):181–188, May 2006.

[36] Intel. Intel® MPI Benchmark. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`, 01/17/12.

[37] Intel. Intel® Xeon® Processor E7-8870. `http://ark.intel.com/products/53580/Intel-Xeon-Processor-E7-8870-%2830M-Cache-2_40-GHz-6_40-GTs-Intel-QPI%29`, 12/09/2011.

[38] ivcblog. Logic devices. `http://beta.ivc.no/blog/2011/03/30/logic-devices/`, 08/22/2011.

[39] S. Jovanović, C. Tanougast, C. Bobda, and S. Weber. CuNoC: A dynamic scalable communication structure for dynamically reconfigurable FPGAs. *Microprocess. Microsyst.*, 33(1):24–36, 2009.

[40] M. Koester, M. Porrmann, and H. Kalte. Task placement for heterogeneous reconfigurable architectures. In *IEEE International Conference on Field-Programmable Technology*, pages 43–50, 2005.

[41] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev. A Communication Aware Online Task Scheduling Algorithm for FPGA-Based Partially Reconfigurable Systems. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 65–68. IEEE Computer Society, 2010.

[42] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL 2006)*, pages 1–6. IEEE, 2006.

[43] Z. Marrakchi, H. Mrabet, U. Farooq, and H. Mehrez. FPGA interconnect topologies exploration. *Int. J. Reconfig. Comput.*, 2009:6:2–6:2, January 2009.

[44] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pages 135–143. ACM, 1999.

[45] C. Maxfield. Altera's new 40nm FPGAs - 2.5 billion transistors! `http://www.eetimes.com/electronics-products/fpga-pld-products/4104287/Altera-s-new-40nm-FPGAs--2-5-billion-transistors-`, 08/24/2011.

[46] T. P. McMahon and A. Skjellum. eMPI/eMPICH: Embedding MPI. In *MPIDC '96: Proceedings of the Second MPI Developers Conference*, page 180. IEEE Computer Society, 1996.

[47] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, 1994.

[48] L. Middendorf and C. Bobda. Declarative Programming with Handel-C. In *ERSA '10 International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2010.

[49] T. Miyamori and K. Olukotun. REMARC (abstract): reconfigurable multimedia array coprocessor. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, FPGA '98, pages 261–. ACM, 1998.

[50] M. Modarressi, H. Sarbazi-Azad, and A. Tavakkol. An efficient dynamically reconfigurable on-chip network architecture. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 166–169. ACM, 2010.

[51] C. Nicopoulos, V. Narayanan, and C. R. Das. *Network-on-Chip: A Holistic Design Exploration.* Springer Sceince + Buisness Mesia, 2009.

[52] O. Ozturk, M. Kandemir, G. Chen, M. J. Irwin, and M. Karakoy. Customized on-chip memories for embedded chip multiprocessors. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 743–748. ACM, 2005.

[53] Pact. Reconfiguration on XPP-III Processors: White Paper. `http://www.pactxpp.com/download/XPP-III_reconfiguration_WP.pdf`, 10/25/2011.

[54] Pact. XPP-iii processor overview: White paper. `http://www.pactxpp.com/download/XPP-III_overview_WP.pdf`, 08/29/2011.

[55] T. Pionteck, C. Albrecht, and R. Koch. A dynamically reconfigurable packet-switched Network-on-Chip. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 136–137. European Design and Automation Association, 2006.

[56] T. Pionteck, C. Albrecht, R. Koch, E. Maehle, M. Hübner, and J. Becker. Communication Architectures for Dynamically Reconfigurable FPGA Designs. In *IPDPS*, pages 1–8. IEEE, 2007.

[57] M. Platzner, J. Teich, and N. Wehn. *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[58] Potassco. A User's Guide to gringo, clasp, clingo and iclingo. `http://dfn.dl.sourceforge.net/project/potassco/potassco_guide/2010-10-04/guide.pdf`, 09/07/2011.

[59] T. D. Richardson, C. Nicopoulos, D. Park, V. Narayanan, J. Xie, C. Das, and V. Degalahal. A Hybrid SoC Interconnect with Dynamic TDMA-Based Transaction-Less Buses and On-Chip Networks. In *VLSID '06: Proceedings of the 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design*, pages 657–664. IEEE Computer Society, 2006.

[60] M. Saldaña and P. Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *International Conference on Field Programmable Logic and Applications 2006 (FPL 2006)*, pages 1–6, 2006.

[61] J. Septién, H. Mecha, D. Mozos, and J. Tabero. 2D defragmentation heuristics for hardware multitasking on reconfigurable devices. In *IEEE Reconfigurable Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium*, 2006.

[62] H. Singh, G. Lu, E. Filho, R. Maestre, M. Lee, F. Kurdahi, and N. Bagherzadeh. MorphoSys: case study of a reconfigurable computing system targeting multimedia applications. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 573–578. ACM, 2000.

[63] C. Steiger, H. Walder, and M. Platzner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE Trans. Comput.*, 53(11):1393–1407, 2004.

[64] C. Steiger, H. Walder, M. Platzner, and L. Thiele. Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices. In *In: Proceedings of the 24th International Real-Time Systems Symposium, Cancun*, pages 224–235, 2003.

[65] J. Suh, D. Kang, and S. P. Crago. A Communication Scheduling Algorithm for Multi-FPGA Systems. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '00, pages 299–. IEEE Computer Society, 2000.

[66] J. Tabero, J. Septién, H. Mecha, and D. Mozos. A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *Field Programmable Logic and Applications (FPL 2004)*, volume 3203 of *Lecture Notes in Computer Science*, pages 241–250. Springer, 2004.

[67] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04. IEEE Computer Society, 2004.

[68] S. Vassiliadis and D. Soudris. *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Publishing Company, Incorporated, 2007.

[69] J. von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15:27–75, October 1993.

[70] X. Wang and S. Thota. Design and Implementation of a Resource-Efficient Communication Architecture for Multiprocessors on FPGAs. In *Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*, pages 25–30. IEEE Computer Society, 2008.

[71] J. A. Williams, I. Syed, J. Wu, and N. W. Bergmann. A Reconfigurable Cluster-on-Chip Architecture with MPI Communication Layer. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 350–352. IEEE Computer Society, 2006.

[72] Xilinx. Difference-Based Partial Reconfiguration. `http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf`, 10/25/2011.

[73] Xilinx. Microblaze soft processor core. `http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm`, 01/17/12.

[74] Xilinx. ML403 Evaluation Platform Documentation. `www.xilinx.com/products/boards/ml403/docs.htm`, 01/17/12.

[75] Xilinx. Virtex-6 FPGA DSP48E1 Slice. `http://www.xilinx.com/support/documentation/user_guides/ug369.pdf`, 08/24/2011.

[76] Xilinx. Xilinx ML310 Documentation and Tutorials. `http://www.xilinx.com/products/boards/ml310/current/`, 01/17/12.

List of Figures

List of Tables