Architectures, Methods, and Tools for Distributed Run-time Reconfigurable FPGA-based Systems

Der Technischen Fakultät der Universität Erlangen-Nürnberg zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Dirk Koch

Erlangen 2009
## Contents

1 Introduction ................................................................. 1
  1.1 Basic Definitions on Reconfigurable Computing ..................... 3
  1.2 Overview ..................................................................... 5

2 Self-adaptive Reconfigurable Networks ................................ 9
  2.1 Basic Definitions and Terminology .................................. 12
    2.1.1 System Model ...................................................... 12
    2.1.2 Concepts for Self-adaptive Reconfigurable Systems .......... 13
  2.2 State Dependent Software Task Migration .......................... 16
    2.2.1 Related Work on Software Checkpointing .................... 16
    2.2.2 Migrating Software States Using Checkpointing ............. 20
  2.3 State Dependent Hardware Task Migration .......................... 21
    2.3.1 Related Work on Hardware Checkpointing .................... 23
    2.3.2 FSM-based Task Model ........................................... 24
    2.3.3 Checkpointing Input and Output States ....................... 25
  2.4 Hardware/Software Morphing ........................................ 27
    2.4.1 Related Work on Hardware/Software Morphing ............... 28
    2.4.2 Modeling Hardware/Software Morphing ....................... 29
    2.4.3 State Translation ................................................ 32
  2.5 Reconfigurable Networks for Self-adaptive Systems .............. 34
    2.5.1 Network Topology ................................................. 36
    2.5.2 Fast Link State Detection ...................................... 37
    2.5.3 Algorithms for Dynamic Rerouting ............................ 38
    2.5.4 Real-time Communication ....................................... 39
  2.6 Operating System Services for Self-adaptive Networks .......... 43
    2.6.1 Implementing Software Migratable Tasks ........................ 44
    2.6.2 Migrating Software Tasks in a ReCoNet ........................ 45
    2.6.3 Techniques for Saving and Restoring Hardware Checkpoints 48
    2.6.4 Automatic Hardware Transformation with StateAccess ....... 53
    2.6.5 Experimental Evaluation ....................................... 54
    2.6.6 Design Flow for Implementing Morphable Tasks .............. 60
    2.6.7 Morphing Case Study ............................................ 62
  2.7 Design, Implementation, and Evaluation of a ReCoNet Protocol ... 63
    2.7.1 Physical Layer .................................................... 64
2.7.2 Electrical Interface ........................................... 64
2.7.3 Frame Format and Node Addressing .................... 67
2.7.4 Task-to-Task Communication .............................. 69
2.7.5 Implementation of a ReCoNet Router .................... 72
2.7.6 Experimental Evaluation of a ReCoNet ................. 74
2.8 Chapter Summary .............................................. 77
3 Intra-FPGA Communication Architectures for Reconfigurable Systems 83
3.1 Preconsiderations on Flexibility and Module Placement ........ 85
3.2 Efficiency Preconsiderations ................................. 86
3.3 FPGA Architecture Model .................................. 89
3.4 Related Work on Communication for Reconfigurable Systems ... 93
  3.4.1 Circuit Switching ....................................... 94
  3.4.2 Networks on a Chip ..................................... 98
  3.4.3 Buses .................................................... 100
  3.4.4 Physical Implementation Techniques .................. 105
3.5 Bus-Based Communication for Reconfigurable Systems ......... 114
  3.5.1 Shared Write Signals .................................. 117
  3.5.2 Dedicated Write Signals ................................ 119
  3.5.3 Shared Read Signals .................................... 128
  3.5.4 Dedicated Read Signals ................................ 133
  3.5.5 Module-to-Module ReCoBus Communication ............ 141
  3.5.6 Buses for Two-dimensional Placement .................. 142
3.6 Point-to-Point Communication ................................ 146
  3.6.1 One-dimensional Online Circuit Switching .............. 149
  3.6.2 Two-dimensional Online Circuit Switching ............. 152
3.7 Experimental Results ....................................... 157
  3.7.1 ReCoBus Timing Characteristics ........................ 157
  3.7.2 Logic Cost of the ReCoBus Architecture ................ 167
  3.7.3 The Impact of Modularization on the Physical Implementation 172
  3.7.4 Glitch-free Reconfiguration ............................. 176
  3.7.5 Communication for Two-dimensional Module Placement ... 186
3.8 Chapter Contributions and Discussion ....................... 197
4 Building Partially Reconfigurable Systems – Methods and Tools 199
4.1 A Framework for Component-Based Reconfigurable System Design 200
  4.1.1 Simulating Run-time Reconfiguration .................. 204
  4.1.2 ReCoBus-Builder: a Builder for Reconfigurable Systems ... 207
  4.1.3 Bitstream Assembly and Hardware Linking ............. 215
  4.1.4 Core-Based System Design ............................. 216
## Contents

4.2 Run-time Reconfiguration ........................................... 217  
  4.2.1 Horizontal Bitstream Relocation ............................... 218  
  4.2.2 Vertical Bitstream Relocation ................................. 219  
  4.2.3 Small Bitstream Manipulations ................................. 222  
4.3 High Speed Reconfiguration ........................................ 223  
  4.3.1 Configuration Prefetching ..................................... 223  
  4.3.2 Hardware Accelerated Bitstream Decompression ............. 225  
  4.3.3 Algorithms for Bitstream Compression ....................... 228  
  4.3.4 Benchmarking Bitstream Decompression Accelerators ...... 238  
  4.3.5 ASIC Implementation of an LZSS Decompressor ........... 244  
  4.3.6 High-speed Defragmentation .................................. 244  
4.4 Chapter Summary and Contributions .............................. 249  
5 Concluding Remarks .................................................... 251  
  5.1 Future Work ......................................................... 253  
6 German Part .............................................................. 255

Acronyms ............................................................................. 285

Symbols ............................................................................... 287
1. Introduction

More than two decades in FPGA evolution have emerged in a multibillion US$ market and devices that are currently providing one thousand and more multipliers, megabytes of on chip memory, hundreds of thousands of logic cells, and clock speeds of a half gigahertz. With this progress, FPGAs went far beyond the glue logic stage from their early days and are now occupying central positions in highly complex systems. But not only the capabilities of the high-end FPGAs have expanded with the progress in silicon industry, also the cost per function has enormously decreased such that a one-US$ FPGA can nowadays host a microcontroller and a few peripherals. Hence, FPGAs can be found in many present systems and we all may probably have used FPGAs without being aware if this, like, for example in our home network equipment, flat screen televisions, or cars.

In addition to their commercial success, the introduction of reconfigurable devices has established a large area of research under the name reconfigurable computing which, in particular, comprises the investigation of self-adaptive hardware systems. These systems are capable to exchange, update, or extend hardware functionality after production at runtime. Instead of providing all hardware modules in parallel, a self-adaptive system may load or swap modules with respect to present requirements or environmental conditions. Consequently, such systems may be implemented more cost and power efficient on smaller devices or faster by allowing more area for exchangeable accelerators. This is highly relevant for FPGAs where the gap to application specific integrated circuits (ASICs) is monumental. For example, in the same 90-nm CMOS process technology, a standard cell ASIC requires roughly 18 times less area and 14 times less dynamic power, while being three to five times faster than an FPGA counterpart [KR07]. With the help of partial runtime reconfiguration, this gap can be materially narrowed. Despite these obvious advantages, runtime reconfiguration, as an integral part of a digital system, is still rarely exploited in commercial applications. The main reasons for this are open questions on the following topics:

1. Applications: What are the applications that can considerably benefit from runtime reconfiguration?

2. Methodologies: How can hardware modules be efficiently integrated into a system at runtime? How can this be implemented with present FPGA technology? And, how can runtime reconfigurable systems be managed at runtime?
1. Introduction

3. **Tools:** How to implement reconfigurable systems at a high level of abstraction for increasing design productivity?

This thesis aims to answer these fundamental questions. An example of the right application will be given by an automotive application where a reconfigurable distributed network provides fault tolerance with a minimum of structural redundancy by self-adapting and self-healing the system according to currently available resources. This includes migrating hardware and software functionality within the network, replacing and masking defective resources, and optimizing the hardware/software binding of the tasks on the particular network nodes. However, suitable applications for exploiting partial runtime reconfiguration are manifold and not limited to fault tolerant distributed control systems, but this application will demonstrate powerful capabilities for reconfiguring the hardware and software of a system autonomously at runtime. These abilities may then be used – all or in part – in various other domains including networking for adapting to changing protocols, digital signal processing for parametrizing filters or exchanging algorithms, cryptography by swapping sequentially required accelerators (e.g., the result of an asymmetric key exchange will typically be used by a following symmetric encryption) or FPGA-based systems in general. In these systems, mutually exclusive tasks may share the same resources (e.g., multimedia encoding versus decoding in a multimedia device), start-up processes could be accelerated (e.g., the software stored on a slow flash memory may be decompressed by a hardware accelerator that will be replaced with another hardware module after booting), to mention just a few possibilities.

The answers on methodologies required to efficiently implement runtime reconfigurable systems on FPGAs are the most important and will consequently constitute the main contribution in this thesis. For demonstrating the effectiveness of partial reconfiguration, it is not sufficient to prove that modules can be swapped at runtime on an FPGA, it is also necessary to achieve this with a material benefit as compared to a static only implementation. This implies that partial modules can be integrated into a system without a relevant overhead for providing an interface to the reconfigurable modules or by leaving considerable amounts of resources unused on the device. Furthermore, it must be possible to implement a partial system without dropping the clock speed of the modules. For this reason, a novel on-chip communication architecture for FPGAs has been developed, analyzed, and implemented that is capable to efficiently integrate dozens of partial modules into an FPGA at runtime. It is demonstrated that present FPGA technology provides sufficient capabilities to build corresponding systems for real world applications. In addition, focus is put on speeding up self-adaptation because of runtime reconfiguration is in many cases only feasible if the process can be accomplished within certain time boundaries. For example, if reconfiguration is utilized to mask faults in a control system, the time to repair is typically critical, or when accelerators are swapped over time in a signal processing application this must often fulfill real time constraints. Consequently,
all methodologies will be examined with respect to their temporal behavior and addi-
tional techniques have been introduced for reducing the reconfiguration latency, in-
cluding configuration prefetching, hardware accelerated configuration data decom-
pression, and high-speed defragmentation.

Another important question, answered throughout this thesis, is how the implemen-
tation of self-adaptive systems can be automated with the help of tools. The capability
of runtime reconfiguration implies an increased design and test complexity and ade-
quate tools are essential for enhancing design productivity. If, for example, a system
may require several months of extra design and test time to save, let’s say, half the
cost of only an FPGA by exploiting partial runtime reconfiguration, it can be more
economic to use a static only design for bringing the system faster to market at a
lower risk. For bridging the design tool gap in implementing runtime reconfigurable
systems, a novel and easily usable design flow – including powerful tools – will be
introduced in this thesis. The flow allows to compose systems based on fully imple-
mented modules that can be integrated to an initial system by merging configuration
data or netlists while providing the option to swap modules at runtime without influ-
encing other parts of the system that are not involved in the reconfiguration process.
The proposed system composition allows to integrate modules among different sized
FPGAs without an additional synthesis or place and route step. The modules can still
be parameterized (e.g., adjusting the addresses or interrupts) while maintaining their
timing. All this results in a highly modularized design flow that enormously simpli-
ifies the system integration phase, thus, making this flow attractive also for complex
and fully static only systems. In other words, this fully component-based flow can
gain more efficiency to traditional flows than the expected additional effort for im-
plementing runtime reconfiguration capabilities in systems on FPGAs.

1.1 Basic Definitions on Reconfigurable Computing

The research in the reconfigurable computing domain is quite emerging and very ac-
tive, thus many terms and definitions have been introduced from multiple researchers
working on the same topic. The goal of this section is to clarify these definitions.

**Definition: Reconfigurable Computing.** Reconfigurable computing is defined
as the study of computer systems embracing reconfigurable devices. This in-
cludes architecture aspects, models, algorithms, and applications. Furthermore,
reconfigurable computing is a computer paradigm combining the flexibility of
programmable software systems with the performance of pure hardware process-
ing.

A *configurable device* is a chip allowing adoptions after or during production of a
system, and, a *reconfigurable device* is respectively a chip allowing adoptions at any
1. Introduction
time in the system, even at runtime. This definition has to be qualified such that the adoption must include a certain level of complexity. For example, exchanging an add operation with a shifter will not be considered as reconfiguring the device, while the change from a Gaussian window operator to a median noise canceller in an image processing system will be. Besides these larger adoptions in space on a chip, a configuration is commonly executed for a longer time in terms of clock cycles or iterations of an algorithm. Strictly following this definition, a CPU is not a reconfigurable device as the instruction stream causes only relatively small changes and because instructions change on a per cycle base.

Configurable devices can be distinguished in many ways and a fundamental classification of reconfigurable devices is their granularity. This includes 1) fine-grained reconfigurable devices such as FPGAs that are based on processing elements operating at bit-level and 2) coarse-grained devices that provide more complex functional blocks such as ALUs. Coarse-grained devices have demonstrated material performance and efficiency benefits (in terms of power or chip area) over traditional von Neumann or Harvard architectures [KSHT09], but these devices are commonly designed for a very specific application domain, e.g., digital signal processing (DSP) [BMN+01]. On the other side, fine-grained FPGAs can virtually implement any digital system or algorithm. Consequently, FPGAs may host a complete SoC while coarse-grained architectures are often embedded into an SoC, e.g., Sony’s Virtual Mobile Engine, the picoChip array, or Tilera’s TILE64 processor. For increasing area and power efficiency on FPGAs, multiple hard-IP cores have been integrated into FPGA architectures, hence, resulting in mixed-grained devices. This includes dedicated multipliers for accelerating DSP algorithms, CPU cores, or peripherals such as PCIe interface modules or memory controllers.

In this thesis, focus is put on distributed embedded control systems which comprises various different application domains, including networking, control systems, and signal processing. Furthermore, the individual nodes of such a network demand a CPU, hence, an FPGA platform is most suitable to meet all these requirements by one single device. However, many of the proposed methodologies throughout this monograph can also be applied to coarse-grained architectures.

FPGA architectures can be classified with respect to their configuration capabilities, as illustrated in Figure 1.1. At the highest level, FPGAs can be separated into one-time configurable devices that can only be used as an ASIC substitute and configurable FPGAs. Configurable FPGAs can in turn be distinguished in globally and partially reconfigurable devices. When globally reconfiguring an FPGA, the complete device configuration is exchanged. As a consequence, all internal states get lost and the FPGA will have to restart its operation. This is appropriate for an in-field update of the FPGA but is unlikely for self-adapting reconfigurable systems as proposed in this thesis.

Consequently, focus will be put on partially reconfigurable systems, which allow to exchange just a fraction of the resources of an FPGA. Partial reconfiguration can be
1.2 Overview

This monograph comprises three main chapters that are devoted to the three key contributions: Chapter 2 presents operating system services for distributed control systems, Chapter 3 reveals methodologies required to integrate hardware modules into future FPGA-based SoCs at runtime with the help of partial reconfiguration, which,

---

\[\text{Definition: Adaptive Computing Systems.} \]

Adaptive computing systems are computing systems that are able to adapt their behavior and structure to changing operating and environmental conditions, time-varying optimization objectives, and physical constraints like changing protocols, new standards, or dynamically changing operation conditions of technical systems.

With respect to distributed embedded control systems, this definition includes the network level as well as the node level. In other words, a distributed adaptive computing system can autonomously adapt the network, e.g., for compensating a link defect or adapt the nodes of a network, e.g., by reconfiguring a module on the FPGA.

---

\[\text{Figure 1.1: Classification of FPGAs by their configuration capabilities. In this Thesis, focus is put on passive and active partially reconfigurable FPGAs.} \]

performed either passive by stopping the FPGA (e.g., by disabling all clocks) or active where the operation can seamlessly continue during the reconfiguration process\(^1\). When using active partial runtime reconfiguration, the system can be designed to perform the reconfiguration autonomously without additional off-chip control. This technique is called self-reconfiguration and constitutes a main point in this work.

With putting focus not only the reconfigurable device but on building complete reconfigurable systems in general, the term adaptive computing systems is widely used. A straight and universal definition of this term is given by Teich in [Tei07]:

---

\[\text{The differentiation in active and passive reconfiguration follows the documentation provided by the Xilinx Inc. that is the market leader in runtime reconfigurable FPGAs. However, the currently largest partially reconfigurable FPGAs (in terms of LUT count) are designed by Abound Logic, Inc. and provide 750k 4-input LUTs [Abo09].} \]
1. **Introduction**

in particular, comprises a novel communication architecture. Finally, Chapter 4 covers all design and tool aspects for implementing self-adaptive systems on FPGAs. In the following, a brief overview of these chapters is given.

Exploiting the potential flexibility of a distributed embedded control system requires sophisticated operating system services for the resource management and for self-adapting the system. The flexibility stems from the underlying system model, revealed in Section 2.1, that separates the functionality, given by a set of communicating tasks, from the network topology. This allows, for example, to relocate tasks from one node within the network to another, as described in Section 2.2 for migrating software tasks and in Section 2.3 for migrating hardware modules. It is demonstrated that this is possible completely transparent for the migrated tasks and operation can seamlessly continue on another node by using checkpointing techniques for both, software tasks and hardware modules, in order to include also the internal states when migrating functionality. Moreover, with the operating system service of hardware/software morphing it is alternatively possible to change the implementation style of a task between software and hardware, or vice versa, as detailed in Section 2.4. Again, this is transparently possible without losing internal task states. For reconfiguring the network, e.g., when migrating a task or when rerouting broken links, appropriate network protocols and services are necessary which, in particular, has lead to the definition and implementation of a proprietary network protocol, as introduced throughout Section 2.5. All these services have been integrated into a demonstrator running an automotive application in a distributed manner, as depicted in Section 2.8. Among other features, this demonstrator can detect and autonomously compensate node or link failures by migrating tasks and rerouting traffic in a small distributed embedded system.

The capability to migrate hardware tasks and to morph software tasks to hardware implies that hardware modules can be integrated into a system with the help of partial runtime reconfiguration. This demands strategies for carrying out the communication between a static system, providing the basic hardware (e.g., a CPU, a memory controller, and some peripheral modules), and the reconfigurable modules, as deeply examined throughout Chapter 3. When designing FPGA-based runtime reconfigurable systems, the device resources have to be partitioned into a static region and a reconfigurable area tiled into a plurality of resource areas for hosting reconfigurable modules. In Section 3.1, the impact of such a tiling on the placement flexibility is discussed. After this, in Section 3.2, the impact of the tiling on the resource overhead is investigated. There, it is proven that existing approaches are in most practical cases not sufficient to achieve a material benefit when using runtime reconfiguration and that the tile grid has to be arranged much finer as known from existing approaches, with, in some cases, hundreds of individual tiles. Such a fine tile grid has to be physically mapped on an FPGA which requires an according architecture model as introduced in Section 3.3. Such a model has to reflect the geometrical aspects of the logic and
1.2 Overview

The research on reconfigurable computing systems has been quite active for more than a decade and various techniques for integrating reconfigurable modules have been proposed before. Section 3.4 provides a comprehensive survey on related work in the field of physically integrating modules into FPGA-based systems. This survey does not disclose techniques to efficiently provide the communication when tiling the reconfigurable area in a very fine grid. This issue is addressed in Section 3.5 with the introduction of a novel bus-based communication architecture that allows integrating modules by partial reconfiguration in a hot-plug manner. This architecture is capable to provide a bus connection in hundreds of individual tiles at high speed and low resource overhead. Besides buses, partial modules often require dedicated point-to-point links with other parts in the system. For this reason, communication macros, tailored to streaming ports, have been developed, as revealed in Section 3.6. The proposed communication architecture is experimentally evaluated in Section 3.7 and compared with state-of-the-art Solutions in Section 3.8.

The techniques for integrating modules into an FPGA-based system at runtime require additional methods and tools for building such reconfigurable systems, as covered in Chapter 4. The communication architecture, for example, has to be implemented regularly structured and existing tools are not capable to build such structured circuits for FPGAs. This resulted in the development of a novel and easily usable framework for component-based system design. The framework will be presented in Section 4.1 and includes solutions for simulating runtime reconfiguration (Section 4.1.1), a tool for the synthesis and mapping of the proposed communication architecture which additionally comprises a floorplanner for defining and managing reconfigurable regions on an FPGA (Section 4.1.2), and a program for assembling partial configuration files as well as the initial configuration (Section 4.1.3). After these offline design aspects, Section 4.2 puts focus on the technical issues required to load and parameterize partial module configurations into a particular region on an FPGA at runtime. Loading configurations to an FPGA is in many cases time-critical and the configuration time dictates if runtime reconfiguration can be applied in a system or not. Hence, in Section 4.3, methods are proposed for speeding-up the configuration process. This includes configuration prefetching techniques, hardware accelerated bitstream decompressing, and FPGA architecture extensions for fast defragmenting the module layout.

These chapters constitute a monograph that reveals novel concepts for utilizing runtime reconfiguration in distributed control systems and that provides according methodologies and tools to efficiently implement such systems.
2. Self-adaptive Reconfigurable Networks

Connecting several control units via a network is becoming important for a rising number of complex embedded systems. In such systems, each control unit is specialized to execute certain functionality. With the introduction of reconfigurable hardware (commonly FPGAs) as a centric part of the control units, a new dimension of flexibility can be implemented with the help of runtime reconfiguration. On a node level, this includes reconfiguring hardware and software tasks on a single control unit, called ReCoNode in the following, while the reconfiguration may also involve a network level where tasks are migrated among different ReCoNodes of a so-called ReCoNet. An example of a ReCoNet is depicted in Figure 2.1a). The capability of reconfiguring a ReCoNode or a ReCoNet allows the system to automatically adapt to environmental changes, including faults, network topology changes, or varying workload scenarios, as illustrated by the different examples in Figure 2.1a) to 2.1e).

This comprises many novel theoretical aspects like autonomously taking decisions which task is executed where, hence answering the question on which node (network level) and at which placement position on a particular reconfigurable device (node level) a task is executed. Additionally, reconfiguration permits to decide how – in software or in hardware – a task is executed. Consequently, reconfiguration performs typical offline design problems like hardware/software-partitioning and the task binding dynamically at runtime.

With a rising amount of control units in distributed embedded systems, like for example, 60 and more ECUs (electronic control unit) in a car [THKS06], reliability becomes a major concern. The self-adaptive capabilities of a ReCoNet/ReCoNode provides novel strategies for improving fault tolerance in distributed embedded control systems by self-healing techniques and a better resource utilization by self-optimization. For instance, instead of using Triple Modular Redundancy (TMR) with a fixed binding of the redundant task, a ReCoNet may adapt communication, task binding, or even the implementation style (hardware or software) according to the currently available resources or load of the network components at runtime. Obviously, in order to improve fault tolerance, focus will be put on decentralized methodologies for self-optimizing, self-adapting and self-healing a ReCoNet. This makes the system more robust against total loss of state information as compared to centralized approaches. However, by focusing on small networks, with, for example, not more than a hundred ReCoNodes, gathering global information (e.g., the present network topology of the
2. Self-adaptive Reconfigurable Networks

Figure 2.1: Automotive example of a self-adaptive and self-repairing distributed reconfigurable ReCoNet. a) The initial system consists of four electronic control units (ECU1, . . . , ECU4) that are linked together to a small network. The network provides a control application where a sensor task $S$ gathers the state of the cockpit switches and sends it further to a controller task $C$ who is in charge of activating the front and rear lights via the actuator tasks $F$ and $R$. Each ECU consists of reconfigurable hardware (an FPGA) and a CPU, such that hardware and software tasks can be arbitrarily executed or exchanged at runtime. For this reason, such ECUs are also called ReCoNodes. In addition to this reconfiguration on a node level, the system can also reconfigure on a network level (e.g., by migrating a task from one ECU to another one). b) In case of a broken link, the system will detect the link fault and automatically reroute the traffic over alternative network resources. c) When a complete node drops out, the system will detect the node failure and automatically activate a redundant task on another node. Note that this also affects the routing among the ECUs of the network. d) The system is further capable to integrate new tasks (see task $N$) into the system that may communicate with already existing tasks. e) A self-adaptive system may also integrate new nodes (see node ECU5). Due to the self-optimizing capabilities of the system, tasks may autonomously migrate to this node without affecting the currently running application.
ReCoNet might be applicable for each ReCoNode. Such networks constitute by far the majority of all distributed control systems. While different levels of granularity have to be considered in the design of self-adaptive, self-healing, and self-optimizing reconfigurable networked embedded systems, focus is put onto system level aspects throughout this chapter.

The goal of this chapter is to propose novel methodologies for exploiting the—so far mostly unused—capabilities of software—and especially—hardware runtime reconfiguration to fulfill the requirements, which, in particular, includes the reliability, of future distributed embedded systems. The proposed and examined methods include the following services:

**Dynamic Rerouting:** For self-healing broken links and for establishing the communication between the tasks, which are able to migrate within the ReCoNets among different ReCoNodes, routing has to be carried out, which, in particular, handles changing topologies.

**Task Migration:** For repairing node failures and for optimizing the task binding of a ReCoNet, methods for migrating software tasks as well as hardware tasks will be introduced. As the self-optimization should be transparently provided for the tasks involved in the task migration, internal states of the tasks have to be considered.

**Hardware/Software Morphing:** The possibility of reconfiguring both, the software and the hardware of the system allows to swap the execution between these two implementation variants. This capability provides additional freedom for self-optimization and self-healing the system.

With these services, the binding of tasks and communication demands to resources (see also Section 2.1.2), as well as the partitioning between providing functionality in hardware and software, can be modified and optimized at runtime. This will be termed *dynamic hardware/software partitioning*. Dynamic hardware/software partitioning will be discussed in consideration of several constraints and objectives. This includes resource limitations like the available CPU performance, the free reconfigurable hardware resources (in case of an FPGA, this is the free available area of the device), and the available capacity on the network links. Furthermore, reliability aspects of the dynamic hardware/software partitioning will be incorporated. In addition, timing aspects for the proposed services will be analyzed.

Different to architecture or register transfer level, where methods for detecting and correcting transient faults, such as bit flips, are widely applied, static topology changes like node defects, integration of new nodes, or link defects are in the main focus of a ReCoNet. When designing fault tolerant systems, such a ReCoNet, a *Fault Model* has to be defined that specifies which kind of faults shall be detected and repaired. In general, a *fault model* has to be created considering the assumed *fault
2. Self-adaptive Reconfigurable Networks

Types [CPS98] (e.g., data processing faults, data transmission faults, or timing faults) and the assumed fault behavior (e.g., fail silent or Byzantine faults) [CAS94, LSP82]. In the context of a ReCoNet, attention is directed to link failures and node failures with a fail silent behavior. As will be demonstrated in Section 2.7, errors in the communication (e.g., bit-flips, packet loss, or packet reordering) will also be detected and masked or compensated within the implementation of a ReCoNet.

The central issues of this chapter are the technical aspects behind online hardware/software-partitioning which describes the procedure of binding functionality to resources in a ReCoNet at runtime. This chapter will continue with the introduction of definitions and a formal model of a ReCoNet in Section 2.1. Next, methodologies for the migration of software and hardware tasks will be revealed on a formal model basis in Section 2.2 and Section 2.3. Task migration is based on checkpointing concepts such that internal states will not get lost during this process. Checkpointing is further used for the task morphing which allows to swap the execution of a task between either hardware implementation on an FPGA or software implementation on a CPU, as presented in Section 2.4. Here, the theoretical aspects of swapping the execution between hardware and software execution without losing internal states will be modeled and analyzed. The novel techniques for self-adaptivity have been implemented and integrated into an operating system, as well as experimentally evaluated, as summarized in Section 2.6. Section 2.5 discusses requirements and theoretical fundamentals for implementing the network of a ReCoNet. The corresponding implementation be presented in Section 2.7. Finally, Section 2.8 concludes this chapter and reveals a ReCoNets that demonstrates self-adaptive and self-healing in a distributed automotive driver assistance application.

2.1 Basic Definitions and Terminology

Before covering the methods for allowing self-adapting and self-healing in a ReCoNet in detail, the basic system model of a ReCoNet will be introduced.

2.1.1 System Model

The key idea for providing the flexibility of a ReCoNet is to separate the functionality from the network architecture. Hence, the task binding is not fixed to a specific node in a ReCoNet and a task may be hosted at different ReCoNodes at runtime (e.g., by task migration). In the following, a formal model of such a ReCoNet will be presented.

**Definition: Network model.** A network model is a graph $G^n$ consisting of a task graph $G^t$, an architecture graph $G^a$, and mapping edges $E^m$: $G^n = (G^t, G^a, E^m)$.

The network model denotes how an application, modeled by a task graph, is bound to the resources of a certain ReCoNet network, as specified by the architecture graph.
2.1 Basic Definitions and Terminology

An example of such a network model is plotted in Figure 2.2.

**Definition: Task graph.** A task graph $G^t$ is a graph consisting of vertices $V$ representing the application tasks running in a ReCoNet and directed edges $E^t$ modeling the communication dependencies between the application tasks: $G^t = (V, E^t)$.

Note that tasks may be replicated for fault tolerance reasons which results in corresponding communication dependencies, as sketched in Figure 2.2. In distributed control systems, task graphs often have an acyclic structure consisting of a set of sensor tasks (input vertices), control tasks (inner vertices), and actuator tasks (output vertices). Such a structure will also be called a sensor-controller-actuator chain in the following.

**Definition: Architecture graph.** An architecture graph $G^a$ is a graph consisting of vertices $N$, representing the ReCoNodes, and bidirectional edges $E^a$, constituting the communication links between the nodes in a ReCoNet: $G^a = (N, E^a)$.

The architecture of a ReCoNet is not fixed and the topology of the system may change at runtime. For example, links may fail or the network may be extended with additional ReCoNodes. Consequently, the nodes $N$ in a ReCoNet as well as the communication links $E^a$ are assumed to vary at runtime.

**Definition: Mapping edges.** The mapping edges $E^m = E^{mt} \cup E^{mp}$ specify a possible binding of tasks $v_i \in V$ to ReCoNodes $v_j \in N$: $e^{mt} = (v_i, v_j) \in E^{mt} \subseteq V \times N$. Additionally, mapping edges specify a possible binding of communication dependencies $E^t$ to paths $2^N$ in the architecture graph $G^a$: $E^{mp} \subseteq E^t \times 2^N$.

Communication dependencies may be routed over multiple hops and a path is a sequence of vertices $(v_0, v_1, \ldots, v_k)$, with $v_i \in N$. The path contains these vertices and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, with $(v_{i-1}, v_i) \in E^a$. The path is simple, i.e., all vertices in the path are distinct. The start vertex $v_0$ and the end vertex $v_k$ of the path are defined by the task binding: $e^t = (t_i, t_j) \in E^t$ with $t_i$ and $t_j \in V$, then $e^{mt}_0 = (t_i, v_0) \in E^{mt}$ and $e^{mt}_k = (t_j, v_k) \in E^{mt}$ with $v_0$ and $v_k \in N$. Note that $v_k$ can be $v_0$ which implies that two dependent tasks are bound to the same ReCoNode.

2.1.2 Concepts for Self-adaptive Reconfigurable Systems

As a new capability of future distributed embedded systems, a ReCoNet is capable to integrate new hardware or software tasks at runtime. Furthermore, with the help of task migration and hardware/software morphing, the systems can autonomously react on changing load scenarios or resource defects. This allows self-adapting the functionality of a system to changing demands as well as self-healing to maintain the functionality of a system. However, self-adaptation and self-healing is typically required to be accomplished very fast. For this reason, a two step strategy consisting of a so-called fast repair phase and a non time-critical hardware/software partitioning
2. **Self-adaptive Reconfigurable Networks**

![Network model diagram](image)

**Figure 2.2**: Network model. The network model of a ReCoNet consists of a task graph $G^t$, an architecture graph $G^a$, and mapping edges $E^m = E^{mt} \cup E^{mp}$ denoting the binding of an application onto the architecture. For the sake of clarity, only the task binding $E^{mt}$ is shown while the communication binding $E^{mp}$ has been omitted. The tasks $t_4$, $t_5$, and $t_6$ have been duplicated with corresponding replica tasks $t_4'$, $t_5'$, and $t_6'$ for fault-tolerance reasons. The dashed edges in the task graph can be activated in case of a fault. For example, if node $v_2$ fails, the tasks $t_5$ and $t_6$ will fail, too, and the replica tasks $t_5'$ and $t_6'$ will become active on node $v_4$ and data from task $t_4$ will now be sent to the activated replicas. Each ReCoNode provides is assumed to include at least CPU, memory, I/O interface modules and resources for runtime reconfigurable hardware tasks (RCHW).

...phase is used as illustrated in Figure 2.3 and examined in [Str08]. In case of a node or link defect, which implies a change of the network topology, redundant replicas may be activated and a new routing for the task-to-task communication is determined. After this time-critical phase, the task binding is optimized and, if required, new replica tasks are generated.

The optimization of the task binding shall in general respect multiple objectives, including the hardware and software load on a specific node, the total load in the network, the ratio between hardware and software load, the available capacity on the communication links, or the reliability of the system. For fault tolerance reasons, the optimization algorithms have been designed in a distributed manner and each ReCoNode requires only its own and the present hardware and software load of its direct neighbors. The first investigated algorithm by Steichert [Str08] is a *discrete diffusion-based task binding* that determines locally on each ReCoNode whenever to distribute...
2.1 Basic Definitions and Terminology

- defect
- extension
- node/link defect
- switching to replicas
- binding of new tasks
- (re-)routing of task-to-task communication
- additional node/link
- fast repair
- new arriving task
- optimization of task binding
- task replication

Figure 2.3: Online two phase strategy used in a ReCoNet. In case of a node or link defect, the fast repair phase activates replicated tasks and reroutes traffic to or from the replicas. If a new ReCoNode is integrated into the network, the fast repair phase will determine new routes to the tasks routed on the new tasks. If a new task arrives at one node in the network, this node searches for a node providing sufficient resources to bind the new task. After this, the hardware/software partitioning phase optimizes the task binding and creates new replicas, when required. The figure is taken from [Str08].

load to or receive load from the adjacent nodes. The decisions are taken according to the hardware and software load differences between a particular node and its direct neighbors. This algorithm produces a uniform distribution of the hardware and software load within the network by migrating tasks among the network.

After this load balancing step, task replication will be performed, if necessary. Note that it is assumed that only one defect occurs at each point of time (e.g., a link or node failure) and that the minimum defect arrival time will larger than the time required for the fast repair and hardware/software partitioning steps. A fault may result in a decomposition of the network in two or more disconnected parts (e.g., if the center node of a star topology fails). Hence the replica placement should be arranged such that after a decomposition one part contains at least one instance of the tasks constituting an application (e.g., a sensor-controller-actuator chain). See [SGW+08] for more details. The instance may be either an active running task or a replica that can be activated after a defect. It is possible to identify regions within a network that will not decompose in case of a single fault. This problem is know from graph theory.
2. Self-adaptive Reconfigurable Networks

as finding biconnected components and Hopcroft and Tarjan presented in [HT73] a corresponding algorithm with linear time complexity. A biconnected component of a undirected graph is a maximal set of edges such that any two edges in the set lie on a common simple cycle [CLRS01]. With the knowledge of the biconnected components in a ReCoNet, the replica placement will be performed to increase the reliability of the system as discussed in [Str08].

In this thesis, the technical bases required for such self-adapting and self-healing networks called ReCoNets will be proposed and analyzed. This includes task migration techniques for hardware and software tasks (e.g., for performing a load balancing step), methodologies and tools for hardware/software morphing (e.g., for balancing load between reconfigurable area on an FPGA and computing time on a CPU), and rerouting strategies to provide communication between the tasks of a ReCoNet after a defect or in case of a task migration.

2.2 State Dependent Software Task Migration

This section reveals methodologies for migrating software tasks from one ReCoNode to another including their internal state. This feature is essential when migrating tasks onto other nodes at runtime as part of the dynamic hardware/software partitioning and will be used for autonomically reacting on changing load scenarios or node defects. In case of a task migration, the state information needs to be copied together with the program binaries to the destination ReCoNode that shall seamlessly continue execution. Consequently, methodologies are required for 1) dynamically integrating software tasks into a ReCoNode, and 2) transferring the state of tasks efficiently. In order to accomplish this, related work, such as work on checkpointing techniques, will be reviewed. Based on this, methods for migrating software tasks in a ReCoNet will be proposed and analyzed.

2.2.1 Related Work on Software Checkpointing

Many techniques have been developed for adding high reliability and availability to distributed embedded systems. These techniques include 1) transactions, 2) group communications, and 3) rollback recovery, as identified by Elnozahy [EAWJ02]. Transaction processing, which is commonly used in databases and some modern file systems, processes information atomically in the sense that all operations of a transaction have to be accomplished successfully for also processing the transaction successfully [GR92]. Group communication is often used in ad-hoc networks for guaranteeing that messages are sent reliably and omission free to members of a group (a set of tasks or nodes in a network)\(^1\). Rollback recovery resets the state of a system

\(^1\)This should not be mixed up with the term secure group communication (SGC) that refers to a scenario in which peers of a group can exchange messages, such that outsiders are unable to glean
2.2 State Dependent Software Task Migration

Figure 2.4: Global checkpoint consistency. a) ReCoNet with three communicating tasks. b) Waveform illustrating times of stored checkpoints and transferred messages between the tasks. The three saved (local) checkpoints A, B, and C represent an inconsistent state for the complete system, because task $t_3$ would receive message $m_2$ two times in case of a rollback. Opposed to this, the three checkpoints A, B, and D represent a consistent global checkpoint. The most recent consistent global checkpoint is called the recovery line. Note that message $m_1$ does not violate the consistency, as it is a message that is sent but not yet received (after a rollback). Such messages are called intransit messages and have to be logged.

Back to an earlier consistently stored version, which is typically gathered by using checkpointing.

Definition: Checkpoint A checkpoint is a set of data items representing the image of the last error-free state of a module of computation from which in case of the occurrence of a fault may be restarted [KHT07].

Checkpointing and Rollback Recovery Protocols

Checkpointing and rollback recovery protocols have to ensure that the state in a system with multiple communicating tasks can be consistently recovered, as illustrated in Figure 2.4. Such protocols can be classified into three categories: 1) uncoordinated checkpointing, 2) coordinated checkpointing, and 3) communication-induced checkpointing.

In uncoordinated checkpointing, each process decides autonomously when to store a checkpoint. This allows taking checkpoints at the most convenient moment (e.g. if a task is in an idle phase). Without coordination, there is a possibility of the domino effect, which may result in losing a larger amount of useful work [WCLF95]. In its extreme, the domino effect can cause a system to possibly rollback all the way to the beginning of the computation, as depicted in Figure 2.5. Another drawback of uncoordinated checkpointing, as shown in the figure, is that more checkpoints have to be stored as compared to other schemes and that many checkpoints may be useless.

---

any information from these messages [ZRM05].
Figure 2.5: Domino effect. The fault of task $t_3$ forces a rollback to checkpoint I. This would send message $m_8$ a second time, hence forcing task $t_2$ to rollback to checkpoint H. Because of the messages $m_1, \ldots, m_7$, this continues until the rollback terminates at the three consistent checkpoints A, B, and C. The example is taken from [EAWJ02].


t
recovery
line

Coordinated checkpointing avoids the domino effect by synchronizing the moment of taking the checkpoints such that a consistent global state will be stored. This scheme requires only one checkpoint to be stored per task and performing a rollback is trivial. Coordinated checkpointing pays a latency penalty to organize taking the checkpoints synchronously, but however, it can be applied to large scaled systems as demonstrated by Elnozahy [EP04]. A common approach to take periodically a synchronized checkpoint in a distributed system is to use a global time-base [TKT92, LWK03]. An issue to consider when applying checkpointing in distributed systems are messages that are sent between the tasks. In the easiest case, communication can be blocked when taking a checkpoint [TS84]. In that work, a checkpoint coordinator broadcasts a checkpoint request to a set of tasks that will stop execution and flush their communication channels (message queues). This process is acknowledged by all tasks back to the coordinator that will trigger the entire checkpoint store operation upon successfully receiving all acknowledge messages. As an alternative, non-blocking checkpoint protocols have been developed where messages are stamped with a checkpoint index. This allows to recover the relative position of the messages with respect to a particular checkpoint [EJZ92].

Communication-induced checkpointing allows tasks to take local checkpoints independently while avoiding the domino effect with the help of checkpoint index stamps that are piggybacked on all messages [BQF99]. As opposed to coordinated checkpointing, which triggers the checkpointing independently to the transmission of the application messages, communication-induced protocols can cause to take further forced checkpoints on the reception of an application message, as depicted in Figure 2.6.
2.2 State Dependent Software Task Migration

Figure 2.6: Communication induced checkpointing. Starting from the left recovery line \( \{C_{1,1}, C_{2,1}, C_{3,1}\} \), further checkpoints may be taken uncoordinated \( (C_{1,2} \text{ and } C_{2,2}) \). All messages piggyback a checkpoint stamp allowing the receiving task to determine the recovery line to which the system must rollback in case of a fault. For example, when taking checkpoint \( C_{1,2} \), task \( t_1 \) knows that it has to rollback to the old checkpoint \( C_{1,1} \) when the system restarts task \( t_2 \) with the checkpoint \( C_{2,1} \). This is because task \( t_1 \) has received message \( m_{2,1}^1 \) that will then be resent to task \( t_1 \). For avoiding the domino effect, task \( t_1 \) can force receivers of its own messages to take checkpoints which is induced by the start of the communication (here the message transfer of \( m_{3,1}^2 \) to task \( t_3 \)). The forced checkpoint results in a new recovery line and allows to discard all local checkpoints of the left recovery line.

Related Work on Checkpointing Implementation Issues

Checkpointing can be implemented 1) at the operating system level or 2) at the user program level [RHV06]. In the first case, the checkpointing is carried out transparently to the user task with the help of the same mechanisms that are used for a context switch by an operating system. For taking a checkpoint, the corresponding task is stopped and its internal state (e.g., the register file and allocated memory regions) will be copied to secure memory. Opposed to this, the tasks can perform the checkpoint storage by themselves and the programmer of the application is responsible to store the state (typically with the help of dedicated library functions).

If the state has to be used on another node, memory addresses have to be relocated. This is in particular difficult with stacks that can contain various data items, including variables, operands, pointers to data structure, or return addresses from nested loops. The kind of the data item is encoded implicitly in the order on the stack and only the user program can interpret its own stack. Consequently, relocating pointers on a stack is impossible. However, by using virtual memory, relocating pointer addresses can be omitted and the whole virtual memory (including the stack) that is allocated to a task can be transferred to another machine or node [Sue00].

Virtual memory is not applicable in embedded systems providing very limited resources only. Hence, performing checkpointing at the user level is more convenient for such systems. Furthermore, checkpointing at the user level allows to reduce the
2. Self-adaptive Reconfigurable Networks

size of the checkpoint, for example, if there is temporal memory that will be written before it is read again. Then, this memory content is not required to be included in the checkpoint or restored during a rollback operation. Identifying such checkpoints can be automated by a compiler [LF90, BPK94].

2.2.2 Migrating Software States Using Checkpointing

Besides the task binary that has to be copied during task migration and started on the destination ReCoNode, each task contains an internal state that is also required to be migrated to the destination. Therefore, it has to be evaluated what kind and amount of information has to be transferred for each checkpoint, in case of task migration. The following list denotes various data items that constitute the state of a software task. They will be discussed with respect to checkpointing in a ReCoNet:

*Register values:* This includes in particular the program counter and the stack pointer. The amount of data stored in the registers of a CPU is relatively small. When checkpointing is carried out by the operating system, this data is necessarily part of the checkpoint. Checkpointing on the operating system level is in general only applicable for systems providing virtual memory, because of the stack, as identified in [Sue00]. For embedded systems, which require very fast and/or very consistent response times, virtual memory is often not used due to decreased determinism. In addition, a ReCoNode does not only provide software functionality, but also hardware tasks. Because of the address translation required for providing virtual memory, direct memory access from hardware tasks will become very costly (in terms of logic and memory latency). These problems are avoided, if the tasks provide the checkpointing functionality by themselves. Then, the register values are not further considered as part of the checkpoint [RHV06].

*Stack and heap:* A stack is commonly used for 1) storing local variables, 2) parameter and result passing to and from subroutines, and 3) storing return addresses from subroutine calls. The latter one, represents, in particular, the subroutine caller hierarchy. A heap is used for storing dynamic data structures. Depending on the software task, the size of the stack and heap can be huge. For example, by using recursion, the stack will rise with every recursive invocation. In general, a stack may contain pointers to program binaries as well as to data items stored in memory. In both cases, an address remapping would be required when using the stack on another ReCoNode and if physical memory is used. Address remapping is not required when using virtual memory, as discussed in [Sue00]. Analogously, a heap can also only be migrated, if it is stored in virtual memory. These difficulties arise for checkpointing at the operating system level, but can be circumvented when checkpointing is carried
out by the tasks themselves, as revealed in Section 2.6.1. Then the stack is not considered as part of the checkpoint, which implies that checkpoints cannot be taken at any time (e.g., during the executing of a function).

**OS information:** Software tasks can employ further resources from the local operating system like, for example, files, devices, or semaphores. While files might be copied during task migration, it cannot be guaranteed that a device or semaphore will be available on the destination ReCoNode. Hence, they are not supported for task migration.

**Variables:** Variables are obligatory to be transferred when migrating a software tasks. This can be provided implicitly by the operating system (when using virtual memory) by copying all data memory segments of a task to the destination ReCoNode. Alternatively, variables can be explicitly copied to the checkpoint by the tasks themselves. In particular, this allows to skip dead variables (variables that will be never read in the future or that will only be read after they are written). In [LF90, BPK94], corresponding tools are described for automatically including such checkpointing functionality into software programs.

It can be summarized that checkpointing at the task level is more appropriate than on an operating system level, because of the limited capabilities of the assumed ReCoNodes (no runtime environment and no virtual memory). As already mentioned, virtual memory is unlikely to be used together with hardware tasks that posses DMA capability. In addition, it comprises a decreased determinism in the latency to access memory, which may prevent virtual memory to be used in real-time systems. Besides virtual memory, there are further restrictions or limitations that have to be considered when migrating tasks in a ReCoNet.

As mentioned above, the stack represents the subroutine caller hierarchy. If checkpointing is restricted to be only performed in idle phases of a software task, then there exists no subroutine caller hierarchy that has to be considered during task migration. Such idle phases exist in many tasks (e.g., after a sensor value has been processed). A further restriction is that variables must be accessible when taking a checkpoint. For example, variables might not be accessible, if they are declared as static in sub-functions.

With these restrictions, state dependent software task migration is appropriate for a ReCoNet, as demonstrated in Section 2.6.1. In that section, details on the implementation issues of software task migration will be revealed and in Section 2.6.2 software checkpointing will be experimentally evaluated.

### 2.3 State Dependent Hardware Task Migration

Dynamic hardware/software partitioning comprises also the migration of hardware tasks among different nodes in a ReCoNet. This permits, for example, that a ReCo-
2. Self-adaptive Reconfigurable Networks

Node delegates one or more hardware tasks to other nodes in a ReCoNet if, for example, the local hardware resources (FPGA area) are heavily loaded. Consequently, methods are required for integrating modules into a ReCoNode with the help of partial reconfiguration at runtime. This aspect is deeply examined in Chapters 3 and Chapter 4.

In general, a task contains an internal state that has to be sent together with the configuration data in case of a task migration. In addition, tasks can exchange checkpoints for seamless continuation of operation, for example, in case of a node failure. In the latter case, the state is not transferred once but periodically from one ReCoNode to another. Consequently, techniques are demanded that permit to read a consistent state from a module as well as to initialize a module with this state. The state of a hardware module is a collection of the individual states of all memory elements (registers and flip-flops) used in processing, communication, or data storage parts of the module. A state is consistent, if it represents the global state off all these memory elements of the whole module as it may occur during failure free, correct execution. This state must be observable at a dedicated point of time, like for example at a certain clock cycle\(^2\).

In this section, it is investigated how checkpointing mechanisms that are well known for software systems [EAWJ02], can be utilized in hardware modules running on reconfigurable devices.

From a theoretical perspective, a hardware task will be modeled as a finite state machine that has to be transformed such that its internal state can be accessed via the input and output for taking and restoring a checkpoint. With a hardware task, the state is represented by the content of its memory elements. Therefore, for the implementation of hardware checkpointing, all flip-flops and memories must be made accessible within a hardware module.

Besides discussing the theoretical fundamentals on hardware checkpointing, a further important aspect, examined throughout this section, is the question of how the state of a hardware module can be extracted and restored with a) low latency penalty and b) with low resource overhead. In addition, these penalties and resource overheads have to be estimated as soon as possible (e.g., when information about the state space and the data types used to implement a hardware module is available) in order to take the optimal checkpointing technique for a particular use case.

After an overview of related publications on accessing the state from modules implemented on FPGAs in the next section, Section 2.3.2 reveals the model and theoretical background of hardware checkpointing. Next, in Section 2.3.3, the states in the interfaces will be considered. Later in this chapter, multiple techniques for taking and restoring checkpoints in hardware tasks are investigated (Section 2.6.3).

\(^2\)There might exist situations where a register snapshot, taken at a certain clock cycle, is not representing a consistent state. For example, if the snapshot is taken during the evaluation of a multy cycle path.
Implementation details on automatically integrating hardware checkpointing into given modules, as well as an experimental evaluation, will be presented later within this chapter in Section 2.6.4.

2.3.1 Related Work on Hardware Checkpointing

State extraction and restoration of hardware modules using only the FPGA configuration interface has been proposed in systems where FPGAs are used for preemptive task execution [Bre97, SLM00, LWH02, KP05]. In the approaches, multiple hardware tasks share the same FPGA resources over time. One main difference between hardware checkpointing and preemptive hardware task execution is that checkpoints are rapidly taken in order to decrease the loss of computation in case of a fault, while a hardware task preemption will be typically triggered rarely because of the time-consuming reconfiguration process. Consequently, the time overhead to access the state of a hardware module is more critical when the state extraction is used for hardware checkpointing. Another difference is that the extracted state data will be used on the same device in case of hardware task preemption, while in the case of hardware checkpointing, the state data should be transferable between different sized devices or even different FPGA types.

Scalera and Trimberger proposed multi-context FPGAs for swapping between multiple configuration planes in a single cycle [TCJW97, SV98]. As these planes are assumed to be read/writable, it would be possible to use multi-context FPGAs for hardware checkpointing. In this case, taking and restoring a checkpoint may be performed on an inactive plane. However, such FPGAs would be very costly and are not available.

Asadi and Tahoori proposed to use hardware checkpointing for recovering FPGAs from single event upsets [AT05]. In their system, the configuration data is continuously read back and compared with the original configuration by computing CRC checksums. The authors aim to use checkpointing to restart execution with an error-free state without presenting any further details.

Reading back or initiating the present register state of a module is widely used in ASIC design for test and debug purpose. Here, the flip-flops of an ASIC are included into a scan chain for providing access to the state of the hardware. For the same...
2. Self-adaptive Reconfigurable Networks

![Diagram of FSM and CFSM](image)

**Figure 2.7:** a) Finite State Machine (FSM) and b) the corresponding Checkpoint-FSM (CFSM). The original module will be transformed, such that its state can be read or set from outside. The read and write process is controlled by the external signals save and restore.

reason, scan chains have been implemented on FPGAs [WGNH01, TT03].

Besides these general approaches, hardware checkpointing on FPGAs has been addressed rarely. Huang and McCluskey proposed a checkpoint scheme for a fault-tolerant LZ-compressor [HM01]. The approach is highly optimized but no automatic design flow or tool for checkpointing was presented.

### 2.3.2 FSM-based Task Model

Each hardware module is modeled by a Finite State Machine (FSM). An FSM is a sextuple $(\mathcal{I}, \mathcal{O}, \mathcal{S}, \delta, \omega, s_0)$ with a set of inputs $\mathcal{I}$, a set of outputs $\mathcal{O}$, a set of internal states $\mathcal{S}$, a state transition function $\delta$, an output function $\omega$, and an initial state $s_0$ (see also Figure 2.7a). In order to allow restoring a module from a checkpoint, the input values $i \in \mathcal{I}$ and the output values $o \in \mathcal{O}$ have to be restored together with the internal state $s \in \mathcal{S}$ that are all captured during the checkpointing process. This feature has to be supplied by the interface of the hardware module. An FSM providing the capability to save and to restore checkpoints will be called a **checkpoint FSM** (CFSM). The most general case of a CFSM is presented in Figure 2.7b).

For supporting to take and to restore checkpoints in a hardware module, a given original FSM specification has to be extended. Given a conventional FSM $a = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \delta, \omega, s_0)$ and a set of checkpoints $\mathcal{S}_c \subseteq \mathcal{S}$, the corresponding CFSM $a_c = (\mathcal{I}', \mathcal{O}', \mathcal{S}', \delta', \omega', s'_0)$ can be derived as follows:

- $\mathcal{I}' = \mathcal{I} \times \mathcal{S}_c \times \mathcal{I}_{\text{save}} \times \mathcal{I}_{\text{restore}}$;
- $\mathcal{I}_{\text{save}} = \mathcal{I}_{\text{restore}} = \{0, 1\}$;
- $\mathcal{O}' = \mathcal{O} \times \mathcal{S}_c$;
- $\mathcal{S}' = \mathcal{S} \times \mathcal{S}_c$.

A given original FSM is transformed into a CFSM by extending the original input set $\mathcal{I}$ by two control signals $I_{\text{save}}$ and $I_{\text{restore}}$ and a checkpoint $i_c \in \mathcal{S}_c$. With the current
2.3 State Dependent Hardware Task Migration

state being \((s, s') \in \mathcal{S}'\), the state transition function \(\delta' : \mathcal{S}' \times \mathcal{I}' \rightarrow \mathcal{S}'\) is given by:

\[
\delta' = \begin{cases} 
(\delta(s, i), s') & \text{if } i' = (i, -, 0, 0) \\
(\delta(s, i), s) & \text{if } i' = (i, -, 1, 0) \land s \in \mathcal{S}_c \\
(\delta(s, i), s') & \text{if } i' = (i, -, 1, 0) \land s \notin \mathcal{S}_c \land \text{save checkpoint} \\
(i_c, s') & \text{if } i' = (i, i_c, 0, 1) \land \text{restore checkpoint} \\
(i_c, s) & \text{if } i' = (i, i_c, 1, 1) \land s \in \mathcal{S}_c \land \text{swap state with checkpoint } s_c \\
(\delta(s, i), s') & \text{if } i' = (i, i_c, 1, 1) \land s \notin \mathcal{S}_c \land \text{continue to swap checkpoint}
\end{cases}
\]

Similarly, the output is expanded to support checkpoint extraction by linking the present state to \(O'\). The output function \(\omega'\) is defined as: \(\omega' = (\omega(s, i), s')\); and the initial state is \(s'_0 = (s_0, s_0)\).

This system model provides full control over saving checkpoints or initiating a rollback by restoring the last saved checkpoint. If a checkpoint store operation is requested in case of \(s \notin \mathcal{S}_c\), the CFSM will continue its operation in the run mode until the next possible checkpoint has been reached. Note that the CFSM can alternatively swap between the present working state and the checkpoint state. This can be used, for example, for sharing the same hardware resources by multiple hardware tasks (e.g., multiple identical cryptographic modules for multiple secured connections running in parallel). In case \(\mathcal{S}_c = \mathcal{S}\), it is possible to save a checkpoint in each state.

However, it can be advantageous to omit some states from the set of possible checkpoints. If, for example, a hardware module contains a lot of intermediate register values during execution, it is possible to reduce the amount of memory required to store a checkpoint by taking checkpoints only when this module is in an idle state, thus, allowing to omit saving the intermediate register values. In this case, only the idle states are in the set of checkpoints \(\mathcal{S}_c\). Figure 2.8 gives an example of a modulo-4-counter module with the states \(\mathcal{S} = \{0, 1, 2, 3\}\) and the derived CFSM with the set of checkpoint states \(\mathcal{S}_c = \{0, 2\}\). In this example, the state space of the checkpoint is reduced which allows to reduce the memory footprint of the checkpoint. In general, it is possible to omit a register from the checkpoint if this register is written before it will be read (write before read) under all possible state sequences started from a checkpoint state \(s_c\). It is further possible to initialize saving a checkpoint and restoring another checkpoint at the same time. In this case, \(s\) is exchanged with \(s'\) at the next possible checkpoint.

2.3.3 Checkpointing Input and Output States

In most cases, not only the internal state \(s\) has to be restored in case of a rollback but also the input and output states. Furthermore, a rollback of one task is not transparent to its environment and all tasks in the entire sensor-controller-actuator chain
2. Self-adaptive Reconfigurable Networks

![Diagram](image)

**Figure 2.8:** a) Given FSM of a modulo-4-counter and b) corresponding CFSM for \( S_c = \{0, 2\} \), i.e., only in states 0 and 2 saving of a checkpoint is permitted. The state space \( S' \) is given by \( S \times S_c \), which is \( \{0, \ldots, 3\} \times \{0, 2\} \). When exemplarily starting from state \((0, 0)\) in run mode, which is set by the input assignment \((-0, 0, 0)\), the counter will change to state \((1, 0)\). If now a checkpoint is requested to be stored with \( i' = (-1, 0, 0) \), the CFSM will count further to state \((2, 0)\), because the counter state 1 is not a checkpoint state in this example \(s \not\in S_c\). After reaching the state \((2, 0)\), the counter is incremented and the stored checkpoint is overwritten with the present counter state 2. Therefore, the next state will be \((3, 2)\), if the input is still \(i' = (-1, 0, 0)\).

have to restore their last checkpoint when a coordinated checkpoint scheme is used. However, external events between the checkpoint and the fault will still get lost, for example, input values from a sensor task. For systems where losing external events is prohibited, a special memory, called checkpoint FIFO was developed. An example of this FIFO is sketched in Figure 2.9, where the checkpoint FIFO is illustrated as a ring buffer with a write pointer \( WR \) and a read pointer \( RD \). A specific capability of a checkpoint FIFO is its ability to reset the read pointer to an entry of the FIFO that has been read already before. The idea of a checkpoint FIFO, as opposed to a commonly used FIFO, is to keep data that has to be read from the FIFO for allowing a rollback of the checkpoint FIFO. Whenever a checkpoint is saved in the attached CFSM, the present read pointer position \( RD \) is included into the checkpoint of the CFSM. The read pointer, as part of the checkpoint, is labeled \( RD_c \) in Figure 2.9. FIFO data between \( RD_c \) and \( RD \) is not allowed to be overwritten. In the rollback case, the read pointer \( RD \) is set to the position of the stored read pointer at the last taken checkpoint \( (RD_c) \). Consequently, the output values since the last checkpoint will be supplied again to the CFSM. Note that \( RD \) and \( WR \) play the role of typical FIFO pointers with a corresponding read and write port each, while \( RD_c \) is a marker for the last stored checkpoint. The checkpoint FIFO concept ensures to maintain the order of all...
2.4 Hardware/Software Morphing

The decision which particular task is executed by dedicated hardware or as a software program running on an ISP (Instruction Set Processor) is traditionally taken at design time. This partitioning has massive impact to the overall performance of a system, and methods have been proposed that allow to take the final decision, which function should be implemented in hardware or in software, as late as possible [VSV+01] within the design flow. With the introduction of reconfigurable hardware, the hardware/software partitioning problem can be solved online. A ReCoNode, featuring an ISP for executing software and an FPGA for running dedicated tasks in hardware, provides an ideal platform for allowing dynamic hardware/software partitioning. This permits to adapt the system to environmental changes, such as the current computational demands or the allowed power consumption, at runtime. While some tasks in a system have high throughput requirements that can only be fulfilled with dedicated hardware modules (e.g., video preprocessing), other tasks might be highly control flow dominated (e.g., a graphical user interface), and hence, are more ap-
propriate for software implementation. However, for a wide range of applications, both implementation alternatives are feasible and are consequently suitable for applying hardware software morphing. Morphing is especially useful if the demand of an algorithm varies at runtime.

Note that morphing is not limited to change between hardware and software execution. It may also be used to switch among different implementation alternatives within the same domain (hardware or software). This states a further option of using runtime reconfiguration for self-adapting a systems. For example, if data is sent at a variable data rate via a secured connection, the corresponding cryptographic task implementation may be swapped between different hardware modules (for trading off throughput versus area) or software execution, depending on the present data rate.

The remainder of this section is organized as follows. After revealing related work on hardware/software morphing, a formal model for morphing will be sketched in Section 2.4.2. After this, technical aspects required to translate states between the hardware and software domain will be discussed in Section 2.4.3. Later in this chapter, in Section 2.6.7, a case study will be presented.

### 2.4.1 Related Work on Hardware/Software Morphing

Adapting systems to environmental changes is proposed by Stitt et al. by profiling the software that is currently running on an ISP (MIPS). With the acquired data, hardware modules have been generated, synthesized, and mapped to FPGAs at runtime [SLV03] for accelerating software kernels. However, the method was only demonstrated for simple Boolean functions and the authors omit information about how this approach could be properly implemented in real-world systems. Gathering information about the current state of a system and adapting it at runtime was also proposed by Xu et al. [XRT03]. In that work, information about FPGA defects are transferred via a network to a remote reconfiguration server that is responsible to generate a functional system specific configuration to be used by the client located in the field. This technique was not used to automatically generate hardware accelerators for software binaries, but it demonstrates that it is feasible that an embedded system initiates the synthesis and generation of new configurations at runtime, which may be combined with the approach from Stitt.

Instead of running the synthesis at runtime, it is more realistic to implement design alternatives offline and allowing to switch between the variants depending on the current demands or available resources of the system. Such a system was developed at the Interuniversity Microelectronics Centre (IMAC) [MNC+03, MNM+04] and is based on a state machine model, as depicted in Figure 2.10. The idea of this approach is to identify certain states at which the execution can be switched between hardware and software. When the systems takes the decision to change the implementation style, the task will continue operation up to the next change point and then invoke the OS for controlling the switching between the implementation alternatives.
2.4 Hardware/Software Morphing

Figure 2.10: a) Task with two switch points at which the implementation style can be changed between hardware and software. b) For performing the switching, the task is brought from a switch point into an interrupt state from where the OS can access the state of the hardware or software implementation. c) Example of switching from software to hardware execution. The figures are taken from [MNM+04].

It is assumed that the OS can also migrate the internal state between the different implementations. However, the authors consider FPGA readback and bitstream manipulations for this purpose but restrict their system to tasks that run to completion and where no internal state has to be transferred from a software to a hardware implementation or vice versa. This model is considerable limiting the algorithms that are suitable for implementing hardware/software morphing and the technique cannot be applied if the next iteration of an algorithm depends on a prior run (e.g., when implementing a FIR-filter).

2.4.2 Modeling Hardware/Software Morphing

The checkpoint FSM model, which is based on finite state machines and which has been introduced for modeling the checkpointing of hardware modules in Section 2.3.2, is the base for modeling hardware/software morphing. Regardless of the final implementation in hardware or in software, a checkpoint FSM allows 1) to control and 2) to perform accessing the internal state of a ReCoNet task. But that model has to be refined for explicitly allowing state-dependable hardware/software morphing, as revealed in the following section. While the proposed software and hardware checkpointing techniques are the base to read or to write the state of a task, focus will now be put on how and when the state can be translated between a hardware and a software refinement of a task specification. Such a specification (that is independent with respect to its refinement) is assumed to be the base for implementing both, the hardware task refinement, as well as the software counterpart.

In a system providing the morphing of tasks, the set of task implementations \( \mathcal{T} \) is bipartitioned in hardware \( \mathcal{T}^{HW} \) and software tasks \( \mathcal{T}^{SW} \), where \( \mathcal{T} = \mathcal{T}^{HW} \cup \mathcal{T}^{SW} \) with
When performing hardware/software morphing, the implementation style is changed from hardware to software, or vice versa. However, changing the implementation style of a task also affects its binding and, for example, a hardware task, that was bound to an FPGA will have to be bound to an ISP after morphing. Consequently, this will affect the communication binding and the task interfaces, and hence the interface implementation as well\(^4\).

Considering the FSM-based task model, introduced in Section 2.3.2 on Page 24, the state \(S\) of an implementation independent task specification will be transformed into a software state \(F_{SW} : S \rightarrow S^{SW}\), when compiling the specification to a program binary or into a hardware state \(F_{HW} : S \rightarrow S^{HW}\), when synthesizing the specification to a netlist for generating the FPGA configuration data. The two functions \(F_{SW}\) and \(F_{HW}\) denote a mapping of states specified by an abstract specification to states encoded in the entire hardware or software refinements. \(S^{HW}\) is an abstraction of the state of a hardware module and \(S^{SW}\) an abstraction of the state of a software process (represented, e.g., by CPU registers or variables), respectively. In general, \(S^{HW}\) and \(S^{SW}\) are not equal to \(S\), because some elements in \(S\) may be trimmed during optimization. On the other hand, the refinements will typically increase the state space, thus, \(|S^{SW}| > |S|\) and \(|S^{HW}| > |S|\). For instance, when utilizing a library function, this may result in additional internal states not specified in \(S\). Furthermore, the hardware and software refinements are in general not identical: \(S^{HW} \neq S^{SW}\). For example, assuming the following function:

```plaintext
1: boolean example function(A, B, C, D: boolean) {
2:     tmp1 = A AND B;
3:     tmp2 = C AND D;
4:     return {tmp1 OR tmp2};
}
```

then a software implementation of this function will execute the instructions in line 2-4 sequentially on an ISP, while it is possible to evaluate the complete function in parallel by a single look-up table, if it is mapped to an FPGA. In the latter case, all input operands are directly connected to the table inputs and the states of the signals \(tmp1\) and \(tmp2\) will be eliminated.

The example points out that morphing is not possible in all states of \(S^{HW}\) and \(S^{SW}\). In general the mapping from \(S\) to \(S^{HW}\) and \(S^{SW}\) by the functions \(F_{HW}\) and \(F_{SW}\) is not injective. Consequently, identifying a state from the hardware or software refinements that has been mapped from \(S\) (i.e. the original specification) is only feasible within the subsets \(S_{inj}^{HW} \subseteq S^{HW}\) and \(S_{inj}^{SW} \subseteq S^{SW}\), as illustrated in Figure 2.11. Then, the mapping \(F_{inj}^{HW} : S_{inj}^{HW} \rightarrow S\) and \(F_{inj}^{HW} : S_{inj}^{HW} \rightarrow S\) denote injective functions from the refinements back to the states defined in the original specification. Note that such states \(S_{inj}^{HW}\) and \(S_{inj}^{SW}\) can be forced\(^5\), such that morphing is possible in a definable

---

\(^4\)Refer to [JW05] for a comprehensive overview on hardware/software interface techniques.

\(^5\)This can be achieved by preventing possible optimizations. For example, synthesis tools typically
2.4 Hardware/Software Morphing

Figure 2.11: a) Starting from an implementation independent specification with the states $S$, compiling will transform $S$ to software states $S^{SW}$ and synthesizing will transform $S$ to hardware states $S^{HW}$. Due to possible optimizations, not all states of $S$ will be mapped and the number of states typically increases after compilation or synthesis. Only from the subsets $S^{HW}_{inj}$ and $S^{SW}_{inj}$ that denote the mapping of the injective fractions of the functions $F^{SW}$ and $F^{HW}$, it is possible to uniquely identify a state in the original state set $S$. Morphing is only possible in states $\Omega \subseteq S$ that can be uniquely identified from both, the software and the hardware domain.

subset of the states that will be called morph points $\Omega \subseteq S$ (see also Figure 2.11).

**Definition: Morphpoint** A morphpoint represents a state $s \in \Omega \subseteq S$ for which equivalent counterparts exist in both refinements, the hardware and the software domain:

$$\Omega = F^{HW}_{inj}(S^{HW}_{inj}) \cap F^{SW}_{inj}(S^{SW}_{inj}).$$

For identifying such morph points, the state $S$ has to be separated into two sets: 1) a state that is related to the control path $S^{ctrl}$ and 2) to the data path $S^{data}$, with $S = S^{ctrl} \times S^{data}$. Morph points are defined in $\Omega \subseteq S^{ctrl}$ and the state of the data path has to be transformed into the entire other domain (hardware or software) by translating between the state encoding of $S^{SW}_{data}$ and $S^{HW}_{data}$.

The checkpoint FSM model, presented in Section 2.3.2 on page 24, includes explicitly that saving a checkpoint can be restricted to be performed only within a set of definable states (called checkpoint states $S_c$). Then the checkpoint states are a subset of the states and the morphpoints are a subset of the checkpoint states: $\Omega \subseteq S_c \subseteq S$. Note that $S_c$ can be defined different for hardware and software tasks. Additionally, it is possible to take checkpoints that are not morphable, if $S_c \setminus \Omega \neq \emptyset$. This may be used, for example, if the software implementation of a task is much slower than the hardware counterpart, thus requiring much more time to rollback and recompute a task in case of a fault. Then, taking software checkpoints more frequently than just

provide *keep pragmas* for maintaining specific signals even if optimizations could remove these signals (e.g., the signals $tmp_1$ and $tmp_2$ in the Boolean example function).
2. Self-adaptive Reconfigurable Networks

at the morphpoints may reduce the worst case time to recover from a fault.

2.4.3 State Translation

For providing hardware/software morphing without losing internal states, the hardware and software state encodings must be translated among these domains. Each implementation style may use individual encodings for data items, like enumerated states, variables, or constants. The encodings will commonly determined during software compilation or hardware synthesis. However, there exist corresponding data types in the hardware and the software domain and some of them can be directly exchanged. For instance, integer types exist in hardware as well as in software. Consequently, it might be possible to use an identical encoding in both, the hardware and the software refinements. In general three strategies for translating a state between hardware and software have to be considered:

- **Direct state exchange** allows to use the same identical state for the software and the hardware implementation of a task. For copying, checkpointing techniques can be used. Note that this may require to change the order of the individual data items that are stored in a checkpoint.

- **State translation via inverse functions.** Here, the idea is to firstly translate the state back into the original state space $S$, and then, to use the mapping functions ($F_{SW}$ and $F_{HW}$, depending on the morph direction) to translate the state into the target domain. This requires that both, the mapping functions and the inverse mapping functions are known.

- **Direct state translation** aims on directly translating the state encoding without the intermediate step via the original state space $S$. This might be possible, as data types are defined in the original specification and when their encoding in the hardware and software refinements are known.

These strategies may be mixed in a hybrid fashion for different parts of the state. The following paragraphs will reveal the state translation of various data types more detailed.

**Integers**

While in the software case, integer types are coded in fixed sizes (e.g., as a byte or as a 32-bit signed/unsigned), the hardware counterparts are more flexible as they are based on bitvectors that are optimized individually for specific number ranges. The different integer number encodings can be converted when respecting the particular number ranges. Assuming the bit vector representations of the two integer values
2.4 Hardware/Software Morphing

\[ \text{Int}^{SW} = \{ \text{Int}_m^{SW}, \ldots, \text{Int}_0^{SW} \} \] and \[ \text{Int}^{HW} = \{ \text{Int}_n^{HW}, \ldots, \text{Int}_0^{HW} \}, \] then their translation between hardware and software encoding is as follows:

\[
\text{translate}_{\text{Int}^{HW}\rightarrow\text{SW}} = \begin{cases} 
\text{Int}_i^{SW} = \text{Int}_i^{HW} & \forall i \leq n-1 \\
\text{Int}_i^{SW} = \text{Int}_n^{HW} & \forall i \geq n
\end{cases}
\]

\[
\text{translate}_{\text{Int}^{SW}\rightarrow\text{HW}} = \begin{cases} 
\text{Int}_i^{HW} = \text{Int}_i^{SW} & \forall i \leq n-1 \\
\text{Int}_n^{HW} = \text{Int}_m^{SW}
\end{cases}
\]

For integer representations in hardware modules, further optimizations behind the truncation to the required size may apply. This comprises replacing unchanged bits of a register with constant values and a substitution of multiple bits that always possess an identical state to a single bit reference. Such optimizations for constant values or redundancy can be handled by considering report files that are generated by the synthesis tools and allow to reconstruct integer representations for software execution or to compose compatible bitvectors to be used within the hardware. Alternatively, synthesis tools can be forced to omit such optimizations. This may be set with the help of dedicated pragmas, special synthesis options, or by putting such integer registers into the interface of the module.

Boolean values are a special case of an unsigned integer with the number range from zero to one. Consequently, Boolean values are represented with a single bit signal in hardware. In software, this type is usually mapped to an integer.

**Enumerated Type**

Besides integer types, it is possible to define enumerated types that are in particular used to specify states in state machines. While in the software domain enumerates are mapped per default to consecutive integer values, this is unusual in hardware. One well known format for state encoding in hardware is *one-hot encoding*. In this format, an enumerated type with \( n \) states is mapped to an \( n \)-bit wide bitvector and each state is encoded by an individual 1-out-of-\( n \) code. With this encoding, it is possible to determine a specific state by evaluating a single bit value.

The principally different state representations of enumerated types in software and hardware can be translated using the following techniques: 1) forcing the encoding of enumerates in both domains to predefined values (e.g., one hot), 2) forcing the hardware domain to use the same state encoding as determined during software compilation, 3) forcing the software domain to use the same state encoding as determined during hardware synthesis, and 4) generate a translate function that performs a

---

6The size of the used integer number depends on language and compiler version. For example, Visual C++ 4.2 equates bool with a 32-bit integer inside the standard C++ header files that contain the type definition, while all later versions implement Boolean variables with a size of 1 byte.

7Note that in the VHDL language, the state encoding of enumerated tapes is the only specified synthesizable attribute in this language [Ash08].
2. **Self-adaptive Reconfigurable Networks**

mapping between states in the hardware or software domain (e.g., by a state look-up table). As the state encoding has massive impact to the hardware module with respect to the performance (achievable clock frequency) and cost (logic resources), the latter two techniques should be used.

**Further Discussion on State Translation**

With the capability of translating integer types and enumerated types, hardware/software-morphing can be applied to the majority of tasks that are suitable for an implementation in both domains. Floating point numbers, as a further common data type, are costly to be implemented in hardware on FPGAs and various optimization techniques are typically applied for this data type, like dedicated number representations that differ from the standard IEEE 754 format or special circuitry like operator chaining [DL09]. Consequently, translating the state of floating point can be very complex and might prevent to apply morphing.

A further issue, when translating states between hardware and software, is how tasks represent *hierarchies* and how the hierarchy level is controlled. In the software domain, it is common to use a stack that represents hierarchies and functions can be reentered, thus, allowing recursion. In contrast to this, a task implemented in hardware may typically express hierarchies by communicating state machines and recursion is not appropriate to be implemented in hardware. Furthermore, multiple dedicated instances of the same or different functionality may run in parallel in hardware tasks. For the tasks of a ReCoNet, this issue can be solved by selecting morphpoints at states that will not result in hierarchies, like, for example, idle states.

The state translation can either be done in the hardware or in the software domain of the system, depending on allowed latencies and on the rate how often a state translation process is required.

With these restrictions, a state translation that is required for stateful hardware/software morphing is possible and the execution style of a task can be transparently exchanged at runtime. In Section 2.6.6, a tool with a corresponding design flow will demonstrate that morphing capabilities can be automatically added to a task specification in order to implement both, the morphable hardware and the corresponding software task that are based on the same specification. This design flow will be used to implement a case study that is evaluated in Section 2.6.7

2.5 **Reconfigurable Networks for Self-adaptive Systems**

This section is devoted to the topology, management, and real-time behavior of a network that is used in self-adaptive and self-repairing ReCoNets.
The research on networks is very active and many techniques and standards have been established (see [Tan02] for a comprehensive survey on network technology). However, with respect to the requirements of small embedded distributed systems, it was found that existing commercial off-the-shelf (COTS) network components and protocols do not perfectly match the requirements of a typical ReCoNet. According to the Open Systems Interconnection Reference Model (OSI model), the following aspects have to be considered in a ReCoNet:

1. **Physical Layer.** For enhancing fault tolerance, point-to-point links are preferable against a shared bus, as a single failure may halt the complete network in the latter case. In addition, point-to-point links allow to isolate a node completely from the network.

2. **Data Link Layer.** The data items in a ReCoNet should be transferred reliably between the nodes, thus, a ReCoNet demands error detection and recovery techniques. This includes forward error correction (error detecting and correcting codes) as well as backward error correction. Another issue is that link defects should be detected reliably and with low latency in order to quickly reroute links in case of a link or node failure.

3. **Network Layer.** A lot of data in an embedded distributed control system originates from simple sensor or actor states and can be encoded with a few bits, thus, dictating to trim the packet size and especially the header information to enhance efficiency. On the other side, larger data items require larger packets for keeping the overhead low. As a result, the packet size has to be adjustable. Next, the routing should be implemented by dedicated hardware routers for providing more CPU load to the software tasks (hardware forwarding). Furthermore, data in an embedded system may be transferred at different service or priority levels as requested by the transport layer.

4. **Transport Layer.** Within a ReCoNet network, some applications may demand error recovery techniques (e.g., by retransmitting data items) that should be transparently provided to the applications. Besides data integrity, some tasks request to maintain specified throughput (quality of service) and time margins (real-time operation).

5. **Session Layer.** Nodes and tasks have to exchange alive messages or checkpoints to monitor the state of redundant tasks in a ReCoNet and to allow seamless recovery in case of a fault. As tasks can migrate among the ReCoNodes, the actual placement position of a task should be transparent for any task-to-task communication.
2. **Self-adaptive Reconfigurable Networks**

**Figure 2.12:** a) Point-to-point network versus b) redundant bus topology.

6. **Presentation Layer.** The representation and the encoding of data items has to be ensured also for heterogeneous ReCoNets. For example, one ReCoNode may provide a Microblaze softcore processor with big-endian byte order that is communicating with a NIOS softcore processor that is little-endian.

7. **Application Layer.** The ReCoNet protocol should provide an API that is identical for task-to-task communication within the same node or when using the network for a communication among several ReCoNodes.

All these different aspects have to be simultaneously respected in distributed embedded reconfigurable systems. In the following, aspects on the topology are discussed. After this, in Section 2.5.2, it is proposed how the link state (which denotes if a link in a ReCoNet is available or not) can be quickly determined. In Section 2.5.3 routing algorithms are discussed for their suitability to compute routing rules in a ReCoNet. Real-time aspects of a ReCoNet are analyzed in Section 2.5.4. All aspects that are related to the implementation of a ReCoNet will be summarized in Section 2.7 later in this Chapter.

### 2.5.1 Network Topology

A fundamental decision to be made in the design of networked systems is the selection of a good topology. Here, a choice must be taken between point-to-point networks with exclusive links between the network nodes or buses that use a shared medium among all nodes of the system (see Figure 2.12). Both variants have their own assets and drawbacks as summarized in Table 2.1.

In addition to point-to-point and bus topologies, there exist hybrid solutions allowing to combine the advantages of both topologies. In the automotive domain, for example, it is common that different separated CAN buses are linked together via gateways. This allows a higher accumulated throughput and improves reliability by separating the control units by their functions (e.g., power train and cabin control) into different sub-buses.

With putting focus on reliability and flexibility, a point-to-point topology is preferable in a ReCoNet. Buses can also be implemented fault-tolerant with redundant
2.5 Reconfigurable Networks for Self-adaptive Systems

Table 2.1: Qualitative comparison between point-to-point networks and bus-based topologies.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Point-to-point</th>
<th>Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>High (multiple nodes can send simultaneously on different links)</td>
<td>Medium (only one node at a time)</td>
</tr>
<tr>
<td>Management</td>
<td>Routing</td>
<td>Scheduling or arbitration</td>
</tr>
<tr>
<td>Reliability (link)</td>
<td>Rerouting on alternative path</td>
<td>Switching to redundant wire</td>
</tr>
<tr>
<td>Reliability (node)</td>
<td>Activate spare and reroute</td>
<td>Activate spare (eventually, compute new bus schedule)</td>
</tr>
<tr>
<td>Link isolation</td>
<td>Possible via adjacent nodes</td>
<td>Switching to redundant wire</td>
</tr>
<tr>
<td>Node isolation</td>
<td>Possible via adjacent nodes</td>
<td>Difficult</td>
</tr>
<tr>
<td>Time synchron.</td>
<td>Relatively costly</td>
<td>Relatively simple</td>
</tr>
<tr>
<td>Broadcast</td>
<td>Difficult to implement</td>
<td>For free</td>
</tr>
<tr>
<td>Scalability</td>
<td>High</td>
<td>Limited</td>
</tr>
</tbody>
</table>

physical media. As depicted in Figure 2.12 b), communication can be switched to an alternative path in the case of single failure. In this example a second failure will halt the complete system. Such systems have been demonstrated multiple times for fault-tolerant dual CAN buses (e.g., [KGHL98, LY06]). Opposed to this, the point-to-point counterpart (Figure 2.12 a)) can tolerate any two link failures and in some cases, even up to three broken links in the network, while still being able to provide communication to all nodes.

2.5.2 Fast Link State Detection

In case of a link failure, a ReCoNet should be able to establish an alternative route in the shortest possible time. This demands a stable link state detection that can quickly detect a link failure while being robust against bit errors or transient effects. The latter case ensures that an unreliable link will not produce a high interrupt load by rapidly notifying the network driver about new link states.

In a ReCoNet, the physical communication is carried out on wired point-to-point links. For the following, it is assumed that only the respective receiving ReCoNode can sense on a wire for faults. In general, detecting link failures is based on observing the input data at a receiving ReCoNode for expected events within a certain time frame. Here, such events include the successfully reception of data from an adjacent node. Depending on the present task binding and network topology in a ReCoNet,
some links may contain a very low load, thus, carrying a low amount data. In this case, for reducing the latency of the link state detection, dummy traffic has to be automatically generated (e.g., by the router hardware), if no other data can be sent. This will ensure a high deterministic rate of events at all receiver ports in a ReCoNet. Consequently, the time frame in which events are expected can be kept small to detect the link state at also low latency. This will be evaluated at an experiment in Section 2.7.6.

2.5.3 Algorithms for Dynamic Rerouting

Routing is the process of forwarding traffic in a ReCoNet. This is typically provided by analyzing data inside the traffic (e.g., a target address) to forward the traffic to a particular port within the router of a ReCoNet. A routing algorithm is in charge of computing rules that determine which port is taken within a router (e.g., the port to be taken for a certain target address). Due to the point-to-point nature of a ReCoNet, there exist multiple possible network topologies including, ring, mesh, chain (line), tree, or star that may occur anytime at runtime. Consequently, a suitable routing algorithm must be able to find paths for all possible topologies.

For a ReCoNet with a relatively low amount of nodes (up to a hundred), both, distance vector algorithms and link state algorithms are suitable for computing all routes in the complete network. Distance vector algorithms (e.g., Bellman-Ford [Bel58]) exchange the cost to reach other nodes in the network when using a particular port. The path will be selected in each node by sending the traffic further on the port resulting in the lowest cost. Opposed to this, link state algorithms flood the complete network state to all nodes in the network. Each node can then compute the shortest path, for example by using Dijkstra’s algorithm. Alternatively, one node may gather the network state and compute the routing for all nodes in the network. However, running the routing decentralized is inherently fault-tolerant and provides faster computation by parallel execution. For this reason, only distributed algorithms are considered for a ReCoNet. Both, a distance vector routing algorithm as well as a link state algorithm have been tested, as revealed in Section 2.7.6.

Routing Protocols

Routing is a deeply studied topic in computer science and many routing algorithms and routing protocols have been proposed. For a ReCoNet, ad hoc network routing protocols are most suitable as they do not rely on a specific network topology. Good references on the mass of publications on ad hoc routing are [AY05, Roy99].

---

8A routing protocol specifies a standard or convention of how and when nodes in a network compute a new route or exchange network state information while the choice of a route is computed by a routing algorithm.

9Wikipedia is currently listing more than 120 ad hoc network routing protocols
related publications put focus on mobile wireless networks where conditions like power consumption, reliability of the connections, variable bandwidth, or security have to be addressed [BHSW07]. These aspects are not of prime importance for a ReCoNet with cable-based links between the nodes.

Ad hoc routing protocols are classified in 1) pro-active (table-driven) protocols, where the system provides up-to-date routes to all possible nodes before a communication request, and 2) reactive (on-demand) protocols, where the network state is computed on a communication request. Pro-active protocols may imply an overhead for computing unused routes and require to keep track about all changes in the network topology but they provide the routing information with a much lower latency as compared to reactive protocols [PR99, Per99]. Because of the hardware forwarding, a pro-active protocol is more suitable for a ReCoNet. This makes the communication latency predictable, requires smaller buffers, and simplifies the task-to-task communication in case of a task migration. See also the implementation details of the task-to-task communication in Section 2.7.4 for more issues on the routing.

Port Addressing

So far, it was implied that the routers of a ReCoNet evaluate node addressing information in order to decide to which port data is forwarded to. But the question on which level and how the communication between the tasks of the network is carried out is a general question. For a ReCoNets, this means that it must be decided if traffic is sent by interpreting 1) node addresses or 2) task identifiers within the routers. As the number of Tasks in a ReCoNet will be typically much higher than the number of ReCoNodes, node addressing can be implemented more efficiently in hardware, because the routing tables (storing the output port for a particular destination) will be smaller. On the other side, directly addressing tasks allows a more flexible distribution of the network load in a ReCoNet. For example, the communication of various tasks between the same source and destination nodes can be routed on different paths in a ReCoNet, when routing by tasks addresses. However, in case of a resource defect, only one path through the network may exist. In this case, there is no option for routing frames on different alternative paths and a single link must provide sufficient capacity to fulfill all communication demands. Thus, routing via node addresses is more appropriate for a ReCoNet implementation.

2.5.4 Real-time Communication

Providing communication with guaranteed latencies and/or with a certain throughput level is a requirement in many embedded systems. For example, safety-critical sensor or actuator data may have to be transferred within a fixed time margin. Or if the network has to transfer multimedia data, this may demand minimum throughput. This section will analyze the constraints under which Real-Time communication can
be ensured in a ReCoNet under the assumption that traffic shaping is used to limit the amount of data per time that is generated and sent over the ReCoNet.

As the network topology of a ReCoNet may change at runtime (e.g., if a link fails), the worst case capacity that is available before the whole network collapses is bound to the capacity of a single link. For example, in a chain of ReCoNodes, where all packets are sent to one node located at the end of the chain, a single link must provide sufficient capacity for the complete traffic. By limiting the total send rate of high priority packet within a complete ReCoNet to the capacity of one link, it can be guaranteed that a packet is delivered to its destination node within a certain time limit. Note that this does not ensure that the packet is correctly transmitted. This has to be ensured by additional error detection and correction mechanisms. In addition this does not guarantee that packets can be generated at the source ReCoNode or processed at the destination at this speed.

Traffic shaping assigns, within a periodic time frame, which initiator of traffic (e.g., a task) is allowed to send data to a network within particular time slots. In the following, it will be investigated, how big the buffers have to be for a given time frame period such that no packets get lost. This is essential for real-time communication.

For simplicity reasons, it is assumed that traffic shaping time slots posses the time to transmit one packet and that all packets possess the same length. Then, the time slots, as well as the cycle duration of the traffic shaping time frame, can be specified in terms of packets. For the following, the cycle duration will be specified with $t_p$ in terms of packets. Then up to $P$ packets can be generated within the period $t_p$ (that is the time required to send $P$ packets through one link) without losing any packet. This holds true for any feasible network topology. However, if multiple source ports contain packet bursts for the same output port, input queues are required to buffer packets. The size of these buffers will be assumed to be the same for all input ports on all ReCoNodes and it will be denoted with $b$, specified in terms of packets. Additionally, $p$ will state the amount of ports that simultaneously receive bursts (i.e., a sequence of packets).

Figure 2.13 reveals an example where bursts overlap in time. With a larger quantity of active ports the maximum overlapping burst time shrinks. On the other side, multiple active ports receive more packets per time and will fill the input buffer faster. For providing real-time communication, fairness has to be guaranteed when packets from multiple sources are sent to the same output. For this reason, a round-robin send policy is assumed for the following. Then, if input bursts arrive simultaneously at $p$ ports, data from each port will be sent further with a rate of $\frac{1}{p}$ times the capacity of a single link. Consequently, the buffer size depends on the worst case burst length, and hence, on the number of active ports $p$ and the amount of packets $P$ within the time $t_p$. Note that packets have to be discarded if they cannot be stored in the input buffers, because no backlog flow control is assumed. The maximal overlapping length from $p$ source ports is the time required to send $\frac{P}{p}$ packets. During this time, $\frac{p}{p}$ packets will
arrive at a single input port and $\frac{1}{p} \cdot \frac{p}{P}$ packets will be forwarded via the output port. Therefore, at the end of the burst, the buffer requires a capacity of:

$$b = \frac{P}{p} \left( 1 - \frac{1}{p} \right)$$ (2.1)

With this buffer capacity, no packet will be discarded. The maximum of Equation (2.1) points to the worst case scenario with respect to the amount of active source ports:

$$\frac{\partial b}{\partial p} = 0 = \frac{P}{p^3} - \frac{P}{p^2} \left( 1 - \frac{1}{p} \right) \Rightarrow p = 2$$ (2.2)

As a result, the memory consumption in the buffer is maximal when bursts arrive at two active source ports. By using this result in Equation (2.1), the relationship between the traffic shaping period $t_p$ (which is allowed to be up to $P$ packets large) and the buffer size $b$ can be computed: $\frac{P}{t_p} = \frac{4}{7}$. Therefore, no packets will be discarded if the buffer can hold a fourth of the packets that can be sent in one traffic shaping period. Note that real-time transfer is limited to the capacity of a single link. Up to this boundary, no packets will get lost because of any buffer overflow in a ReCoNet. This holds true for any topology of the network.
Numerical Example

If assuming a packet size of 100 byte, a router input buffer, that is implemented using a Xilinx Virtex-II FPGA, can store 20 packets within a single block RAM. In this system, no packets will get lost if all ReCoNodes of the network do not produce more than 80 packets within the time that is required to sent 80 packets through a single link. This holds true regardless to the present network topology of a ReCoNet or the task binding and therefore the location of the packet generation. In other words, when using one block RAM per input buffer, the data rate of a single link is available for the accumulated data rate of any communication demand in a ReCoNet, as long as the number of generated packets in the network does not exceed 80 within the traffic shaping period $t_p$. This holds true, if the packets are periodically generated on a particular node. Any further synchronization between the nodes is not required. Note that different service levels allow to make full use of the network without restricting the real-time capabilities of a ReCoNet. This is considered in the design of the router hardware (see Section 2.7), by providing low and high priority packets.

Worst Case End-to-End Latency

If following the rules for traffic shaping, the upper end-to-end latency bound can be computed for any ReCoNet. The worst case latency $t_{wc}$ for a packet to reach its destination depends on the traffic shaping period $t_p$ (this time specified as a time value), the hop time $t_h$, the number of nodes in a ReCoNet ($|\mathcal{N}|$), and an initial time $t_i$ required in the source and destination nodes to generate and receive a packet:

$$t_{wc} = t_p + |\mathcal{N}| \cdot t_h + t_i$$

As the ReCoNet uses a shortest path routing algorithm, the maximum number of hops, a packet can take, is the diameter of the current network. However, as the ReCoNet topology can change, the number of nodes $|\mathcal{N}|$ is the upper bound of the diameter. The upper bound is reached if the net degrades to a chain.

For the following example, the ReCoNet implementation from Section 2.7 is assumed. With a packet size of 222 bit and baud rate of 12.5 Mbit, the worst case hop time in a ReCoNode is $t_h = 19.2 \mu s$. The maximum traffic shaping period is the time to send 312 packets, which is $t_p = 5.5$ ms. The initial time is about 1 ms (see Section 2.7.6), hence, even for large networks with a diameter of 100 nodes, the worst case end-to-end latency between two communicating tasks is below 10 ms.

This analysis proves that real-time communication within a ReCoNet can be ensured with the help of traffic shaping. This section provides the corresponding design margins. With respect to the examined ReCoNet with 4 ReCoNodes (see Section 2.8), traffic shaping was implicitly provided as the CPUs are not able to generate sufficient traffic. However, for larger networks, traffic shaping should be supported by the communication hardware.
2.6 Operating System Services for Self-adaptive Networks

All methods for establishing self-optimization and self-healing capabilities in a ReCoNet are integrated into the system as OS services implemented in a distributed manner over all ReCoNodes, as illustrated in Figure 2.14. While the reconfigurable network forms the physical layer consisting of the reconfigurable ReCoNodes and communication links, the top layer represents the application that will be dynamically bound on one specific node. This binding of tasks to resources is determined by an online partitioning approach that requires three main mechanisms: 1) Dynamic Rerouting, 2) Hardware/Software Task Migration, and 3) Hardware/Software Morphing. Note that the dynamic rerouting becomes more complex because messages will be sent between tasks that can be hosted by different nodes. These services provided by task migration mechanisms are required for moving tasks from one node to another while the hardware/software morphing allows a dynamic binding of tasks to either reconfigurable hardware resources or a CPU. The task migration and morphing mechanisms require in turn an efficient hardware/software checkpointing such that states of tasks will not get lost. Due to fault tolerance reasons, dynamic rerouting and the opti-
2. Self-adaptive Reconfigurable Networks

mization algorithms of the dynamic hardware/software partitioning have been implemented in a decentralized manner. Task morphing and task migration services are available locally on each ReCoNode.

This section puts special focus on automating the implementation of self-adaptive reconfigurable systems. Consequently, design flows and a novel tool have been developed that will be presented in the following. In addition, for demonstrating the capabilities of the flows and the ReCoNets methods, the OS services have been implemented and evaluated throughout the rest of this section. Section 2.6.1 and Section 2.6.2 reveal a flow and a case study for state dependent software task migration. After analyzing different techniques for hardware checkpointing in Section 2.6.3, the corresponding flow and test system is presented in Section 2.6.4 and Section 2.6.5. Next, Section 2.6.6 and Section 2.6.7 will prove that morphing between hardware and software execution can be efficiently implemented and applied to a real-world test case.

2.6.1 Implementing Software Migratable Tasks

Typical embedded systems for control applications have various resource restrictions, like a missing memory management unit (MMU) or very limited memory. Operating systems for such systems have to respect these restrictions with a reduced range of functionality. In particular, such operating systems do not provide an environment for integrating new tasks at runtime as it is common for desktop computers. For tiny embedded systems, all task object files are linked together with the operating system to one monolithic binary. This is, for example, the case for systems that are based on the MicroC/OS-II developed by Labrosse [Lab98]. This operating system is chosen as the local OS on each ReCoNode. Note that design guidelines for safety-critical applications typically prohibit to generate dynamically linked code. The MISRA guidelines [MIS], for instance, demand static analysis for proving compliance.

The missing runtime environment implies that no new tasks can be added to the scheduler at runtime and that all tasks have to be known during the initialization phase of the OS. Nevertheless, by speculatively starting multiple wrapper tasks, this restriction can circumvented. The wrapper tasks provide a generic part with an API to the OS, including a message passing subsystem, which provides the interprocess communication between the tasks (see Section 2.7.4 for information on the task-to-task communication). This generic part encapsulates the specific functionality of a task that is implemented with the help of virtual functions. The functionality of a ReCoNet task is given by a set of functions that are invoked by a dispatcher. The dispatcher further provides the communication interface for the interprocess communication, which was implemented as a message passing system, as illustrated in Figure 2.15. As the interprocess communication interface is the same for all tasks, only the functions of a ReCoNet task are to be implemented. In case of a task migration, these functions, which means their binaries, are copied to the destination
2.6 Operating System Services for Self-adaptive Networks

Figure 2.15: Model of migratable ReCoNets tasks. All tasks are surrounded by a wrapper task for encapsulating the functionality of a particular ReCoNet task. The wrapper provides an interface to the message passing system and is in charge of starting the particular function of a ReCoNet task. There are no static variables allowed to be stored inside a task function (see also Section 2.2.2). Instead of this, a separate memory region (the data page) is provided for local variable storage in each task. A task is migrated to another ReCoNode by copying the current data page and by overloading the task functions in the target wrapper task.

ReCoNode where they are then integrated with the help of a wrapper task.

When copying a task binary to another ReCoNode, it will be generally located at a different memory address and not only the call addresses to the task functions have to be considered, but also function calls from inside a function of a ReCoNet task. This includes calls to the surrounding system (e.g., requesting the system time) as well as calls to addresses within the ReCoNet task binary. The latter one is solved with a compiler option for generating position independent code that contains only relative branches or jumps within the program binary. For a typical 32-bit CPU, such as the utilized NIOS-II, this is not a restriction, as 32-bit instructions provide in most cases sufficient far relative branches or jumps. Calls to the surrounding systems are accomplished by linking the static system and all library functions to identical addresses on all ReCoNodes, as illustrated in Figure 2.16. This is implemented by guiding the linking process.

2.6.2 Migrating Software Tasks in a ReCoNet

In a ReCoNet, task migration is carried out in three phases. In a first phase, the task binary is copied to the destination node and integrated into the system with a wrapper task, as described in Section 2.6.1. The original task on the source ReCoNode is still active during this phase. After this, the original task takes a checkpoint and transmits it to its copy task in the second phase. This may be issued once in the case of a migration, e.g., for optimizing the present task binding in the ReCoNet, or repeated periodically for seamlessly continuing operation on the occasion of a fault. In the last
Figure 2.16: Mapping of program binaries. a) Without precautions, software tasks are not migratable to another node of a ReCoNet. When task 1 is migrated to node 2, the corresponding task functions (t_function1, ..., 3) are copied and integrated on node 2. When linking the system without constraints, the call to OS_function1 (e.g., for requesting the system time) on node 2 by the task function t_function3 will fail. Note that a wrapper task may host different user tasks at runtime and the binding is not fixed. b) By guiding the linker to map such functions to identical addresses on all nodes in a ReCoNet, tasks can be migrated from one node to another while still being able to call the same functions on all ReCoNodes. The linking is carried out by first mapping the OS kernel followed by all library functions (e.g., printf()) and all wrapper tasks. This code is identical on all nodes and the ReCoNets tasks are located in dynamically allocated memory. For clearness, the local data pages have been omitted.

phase, the original task will be terminated after the replica task has been activated. Note that this includes an update of the task resolution on all nodes of the ReCoNet (see also Section 2.7.4 on Page 70). The task activation can be triggered explicitly for migrating tasks or started implicitly on the absence of keep alive messages.

Keep alive messages are used by the main task to test the availability of the replica task as well as by the replica task to observe the main task. If the main task dies, no further alive messages will be received by the replica task and after a definable timeout value, the replica task will be activated. A tradeoff has to be found between the latency allowed to detect a fault (the period of generating alive messages) and the load on the nodes and network that is required to generate, transmit, and evaluate the alive messages. In the present ReCoNets demonstrator, as explained in Section 2.7.6, keep alive messages are awaited every 100 ms and a task is assumed to be dead after the absence of three consecutive keep alive messages. This results in a worst case
2.6 Operating System Services for Self-adaptive Networks

Figure 2.17: Software checkpointing latencies for different checkpoint sizes. The time is measured between the arrival of a checkpoint request at a task and the finishing of the checkpoint storage operation on another ReCoNode. As the checkpoint latency is in the ms range, the present ReCoNet topology is of minor impact, because of the low hop time within the hardware routers.

For taking global snapshots of a complete sensor-controller-actuator chain, a coordinated checkpointing protocol, similar to the one proposed by Tamir and Séquin [TS84] was implemented. In the protocol, a checkpoint coordinator issues a checkpoint request message to all tasks of a sensor controller actuator chain for signaling the intention to take a checkpoint. Upon reception of this request message, each task...
2. Self-adaptive Reconfigurable Networks

will process all packets from its input message queue but will store the output messages to be sent after taking the checkpoint. After successfully flushing the input queue, each task sends back an acknowledge message to the checkpoint coordinator that will globally trigger the actual checkpoint save process by multicasting a corresponding message after receiving all acknowledge messages. This will transfer the state of each task of the sensor-controller-actuator chain to a corresponding replica task (hosted on a different ReCoNode). After this, all output messages will be passed to the message passing subsystem before processing new messages from the input queues. With this protocol, global checkpoints have been successfully taken from complete sensor controller actuator chains [Dit05]. If a task in the chain detects a fault of another task in the chain (e.g., by not receiving new data), it notifies the checkpoint coordinator for triggering to rollback globally for the whole sensor-controller-actuator chain to the last stored checkpoint.

2.6.3 Techniques for Saving and Restoring Hardware Checkpoints

In this section, different techniques suitable for automatically transforming a module to a checkpointable module will be discussed with respect to the objectives checkpoint latency and hardware overhead. Three different methods for hardware checkpointing will be examined: 1) memory-mapped state access (MM), 2) scan chain based state access (SC), and 2) shadow based scan chain state access (SHC). All these methodologies differ in terms of the latency $t_c$ to extract a checkpoint and the additional hardware overhead $L$ required for saving and restoring the checkpoints. It is presumed that all hardware modules are implemented fully synchronous without any latches. Then, the state of a hardware module is represented by the value of all memory elements inside the module. Consequently, the different approaches provide techniques to read and write all or a subset of the flip-flops inside a module.

Assume that the original hardware module has the following parameters: 1) $K_{LUT}$, $K$ states the number of look-up tables (LUTs) and flip-flops of the module, and 2) $F_{max}$ denotes the modules maximal operating frequency. Additionally, the corresponding variables of the checkpointable hardware module after modification are $K'_{LUT}$, $K'$, and $F'_{max}$. Then, the different methodologies can be compared using the following indicators:

- The Checkpoint Hardware Overhead specifies the amount of additional resources required by a specific checkpoint technique. It will be distinguished between $L_{LUT} = K'_{LUT} - K_{LUT}$ and $L_{FF} = K' - K$, denoting the number of required extra look-up tables and flip-flops, respectively.

- The Checkpoint Frequency Reduction $f_r$ denotes the reduction of the maximal achievable clock frequency due to the increased module complexity ($f_r = \ldots$)
Figure 2.18: Relationship between the checkpoint overhead $t_c$ and the checkpoint latency $t_l$. Checkpoints are stored continuously with the period $t_p$. In order to read a consistent state, the hardware task has to stop its operation for the time $t_c$. The value $t_l$ denotes the time required to save the checkpoint into a secure memory (typically located on another ReCoNode). After a fault occurs, it takes the time $t_d$ to recognize this event. In this case, the redundant CFSM performs a rollback to the last completely stored checkpoint (in this example, the state stored at time $t_0$).

$F_{\text{max}}^* - F_{\text{max}}$.

- The **Checkpoint Overhead** $t_c$ states the time, a module must be interrupted when saving a checkpoint. This interrupt leads to an increase in the execution time.

- The **Checkpoint Latency** $t_l$ specifies the amount of time required until the complete checkpoint data arrives at the device that will take over the task execution in case of a fault.

Based on the overhead $t_c$ and the latency $t_l$, further indicators can be derived for a quantitative comparison of hardware checkpointing techniques. The relationship between $t_c$ and $t_l$ is depicted in Figure 2.18. The example shows a task $T$ sending a checkpoint to another task $T'$ with the checkpoint period $t_p$. Due to the communication to the replica task, the latency $t_l$ is larger than the overhead $t_c$. The checkpoint efficiency, denoting the relative amount of time a module spends on its execution, is defined as $\eta = 1 - \frac{t_c}{t_p}$. Figure 2.18 shows an example where a fault of task $T$ is recognized after a fault detection latency $t_d$. After fault detection, task $T'$ takes over the execution starting with the last completely stored state (here at $t_0$). The maximal loss of computation is $t_{\text{loss}} = t_p + t_l + t_d$.

**Memory-Mapped State Access**

In the memory-mapped methodology (MM), a processor accesses the module’s state by integrating the flip-flops storing the checkpoint into a read/writable memory space of a system. In order to keep the state consistent during read or write operations, the module must not operate until a state extraction or a rollback process has been
2. Self-adaptive Reconfigurable Networks

Figure 2.19: Hardware checkpointing by memory-mapped state access. Flip-flops are clustered together according to the word size of the bus \((B)\), thus, reducing the complexity of the address decoder to \(\frac{K}{B}\) select lines for \(K\) flip-flops.

completed. This is achieved by a multiplexer that feeds back the present value of a flip-flop while saving a checkpoint. As depicted in Figure 2.19, this is implemented by extending each individual flip-flop of the module. For reducing the checkpoint overhead \(t_c\) when storing a checkpoint, flip-flops will be grouped together according to the data-bus size \(B\).

The extra amount of flip-flops \(L_{FF}\) is mainly related to the module interface registers that are marginal as compared to the amount of flip-flops inside the module.

The logic overhead for providing access to \(K\) module flip-flops using memory mapping to a \(B\)-bit wide bus using 4-bit LUTs can be estimated as:

\[
L_{MM}^{LUT} \approx \frac{K}{B} + \sum_{n=0}^{\lceil \log_8 \left( \frac{K}{B} \right) \rceil - 1} 8^n + \frac{K}{2} + B \left[ \frac{K}{2B} - \frac{1}{3} \right] + \frac{K}{\text{feedback mux}}
\]

It must be noted for the address decoder circuit needs a LUT for each output select signal. In case of 4-bit LUTs, it is possible to evaluate three address bits in addition to the enable signal generated by the evaluation of the higher address lines. Thus, the decoder can be modeled as a B-tree with 8 children (selected by three address bits) for each select output wire. The read multiplexer consists of an OR-gate for every bit connected to the read data bus. Wide input OR-gates are mapped as a B-tree with \(k\) children when using look-up tables with \(k\) inputs.

Equation (2.3) states a worst case estimation for the logic overhead resulting from the address decoder, the feedback multiplexers, and the restore logic. However, current architectures and tools support to translate the feedback multiplexer to a flip-flop enable signal, and the restore multiplexer is mapped together with the original circuit.
2.6 Operating System Services for Self-adaptive Networks

Figure 2.20: Hardware checkpointing using a scan chain. Scan multiplexers allow to connect all flip-flops to one large ring shift register. The bus interface contains $B$ extra flip-flops for taking a checkpoint or for performing rollback. A state machine (not shown) keeps track of the current shift position within the scan chain. Furthermore, this automate coordinates the data exchange between the scan chain and the bus interface.

Sharing resources. Therefore, $L_{LUT}$ will be less than the number of flip-flops $K$ in practice.

The checkpoint overhead $t_c$ can be predicted by counting the bus read/write operations necessary to save or restore a checkpoint. With $\tau_{bus}$ denoting the time for a bus cycle, the overhead is about:

$$t_c \approx \frac{K}{B} \cdot \tau_{bus}$$

State Access via Scan Chains

Instead of mapping flip-flop values into the memory address space, the flip-flops can be chained together to a long shift register chain (SC). This is known from established ASIC design techniques. A multiplexer in front of every flip-flop is used to switch between normal operation and the shift register operation where all flip-flops are connected to a long shift register. In general, it is not possible to read the complete chain atomically in one bus cycle. For keeping the state of the chain consistent without additional logic for each flip-flop of the module, the chain ends are connected to a ring shift during scan mode ($t_c$). An additional state machine keeps track of the present shift position of the state data inside the chain. Furthermore, the additional state machine is used to automatically and selectively read and write portions of the flip-flop values through an extra register for accessing the state at the full data bus width $B$, as illustrated in Figure 2.20.

Again, the hardware overhead $L_{FF}$ depends on the module interface $Z$. In addition, some memory is required for the state machine controlling the scan chain access. Beside a constant value $Z_{const}$ (for the control logic), it needs $\lceil \log_2 K \rceil$ flip-flops for a
2. Self-adaptive Reconfigurable Networks

scan shift position counter and \( \lceil \log_2 K / K \rceil \) flip-flops for the word position register for a module with \( K \) flip-flops connected to a \( B \)-bit wide bus. Thus, the flip-flop overhead is:

\[
H_{FF}^{SC} \approx Z_{const} + Z + \lceil \log_2 K \rceil + \lceil \log_2 \frac{K}{B} \rceil + B
\]

In the worst case, the additional amount of look-up tables \( L_{LUT} \) is given by the logic for the scan chain interface plus additional LUTs for the scan multiplexer in front of each flip-flop of the module. This overhead will be less than \( K \) in practice, because of the scan multiplexer may be often integrated into the module logic without additional logic cost. The advantage of the ring-shift is that state consistency can be guaranteed with a low resource overhead by the cost of an increased transfer time for the CPU subsystem. When a specific fraction of the ring shift register is addressed, it takes on average \( \frac{K}{2} \) cycles until the state data has passed the extra register for the state access. \( \frac{K}{B} \) times a value has to be transferred between the extra register and the CPU subsystem. Therefore, the checkpoint overhead \( t_c \) scales with \( O(K^2) \) leading to a poor checkpoint efficiency \( \eta \) for large checkpoints. Instead of using one ring shift register of length \( K \), it is alternatively possible to generate \( K' \) parallel running shift-register chains each of length \( \frac{K}{K'} \). Note that \( K' \) must be chosen to be less or equal than \( B \). Then, the checkpoint overhead \( t_c \) is about a factor of \( \left( \frac{1}{2} \frac{K}{K'} \right) \) \( \cdot \frac{K}{B} \) larger as compared to the memory-mapped approach.

State Access via Shadow Scan Chains

The checkpoint efficiency \( \eta \) decreases dramatically in systems where checkpoints are frequently taken. In this case, it is advantageous to duplicate all flip-flops of the original module to allow copying all values in one single clock cycle, as depicted in Figure 2.21. This permits to continue the execution of a module while moving the state data transfer in the background through the shadow register chain. With respect to the CFSM model, the original module registers store the state \( s \) and the shadow registers the checkpoint state \( s' \). The multiplexers implement the state transition function \( \delta \).

For the shadow scan chain approach, the checkpoint overhead is zero and the efficiency is \( \eta = 1 \). However, \( t_c^{SHC} \) is used to denote the time required to read the shadow chain in the background while the module keeps running. In case of a rollback, the state information is shifted into the shadow chain, then loaded into the working registers and operation will start immediately. If the checkpoint data is stored inside a redundant CFSM, the rollback latency can be reduced to a minimum.

If an FSM refinement with \( K \) flip-flops is transferred to a CFSM, the overhead for the additional flip-flops and LUTs is obviously: \( L_{FF}^{SHC} \approx K \) and \( L_{LUT}^{SHC} \approx 2K \) (without the interface).
2.6 Operating System Services for Self-adaptive Networks

Figure 2.21: Hardware checkpointing using shadow scan chains. By duplicating all flip-flops and arranging them into a chain, checkpoint data transfer can be performed concurrently with the module execution.

### 2.6.4 Automatic Hardware Transformation with StateAccess

This section covers the implementation of hardware checkpointing with the tool StateAccess [KHT07]. StateAccess performs all required transformations to automatically extend a module with the capability of checkpointing. It supports all presented state extraction techniques (see Section 2.6.3) with multiple options. For example, instead of implementing one scan chain, it is possible to generate multiple shorter ones that are used in parallel. The tool works at the netlist level where all types of flip-flops of a module can be directly identified. The original flip-flops will be replaced with checkpoint flip-flops according to the specified state extraction technique. Next, the state extraction interface logic to the system bus will be created and all connections will be automatically generated. The result of this process is again a netlist that will be synthesized together with the remaining system.

The design flow for implementing hardware checkpointing is depicted in Figure 2.22. Starting from an (verified) HDL specification, a netlist is generated in a front-end synthesis step. This uses the Synopsys Design Compiler for generating an EDIF netlist based on GTECH-primitives (Generic TECHnology) [Syn08].

In addition to identifying all flip-flops in the netlist, StateAccess executes design rule checks to detect unsupported constructs like combinatorial feedback paths or latches. The identified flip-flops are listed in a human readable form in a description file. By editing this list, the set of flip-flops and their order within the checkpoint data can be adjusted for guiding the transformation process.

The back-end synthesis step comprises an extended GTECH library with dedicated flip-flop replacement primitives and a library providing templates to interface the state access logic with the rest of the system. The last step does not demand a special synthesis tool. Hence, it is possible to use the checkpoints in heterogeneous FPGA networks. For example, an Altera FPGA-based ReCoNode can directly send a checkpoint to a Xilinx counterpart without any incompatibility or further transformations. This is exemplarily implemented in Section 2.6.5.
2. Self-adaptive Reconfigurable Networks

Figure 2.22: Modified design flow to enable hardware checkpointing with StateAccess.

Further Module Transformations and Limitations

Besides the presented hardware extensions applied at the netlist level, it has to be ensured that during state extraction and state restoration all set or clear signals of the flip-flops are disabled. Furthermore, if flip-flops use enable signals, these have to be forced active. StateAccess supports these signals by adding additional control logic around the flip-flops, as depicted in Figure 2.23. This logic overhead is not considered in the estimates presented above because the amount of additional logic is normally very small and the control signals of the flip-flops can be often shared for complete register banks.

If a hardware module instantiates on-chip memory blocks, a wrapper similar to the memory-mapped approach is automatically generated around the memory. This allows the CPU to randomly access the state stored in on-chip memories. However, StateAccess supports this only for synchronous single port memories that must be directly located inside the module netlist.

2.6.5 Experimental Evaluation

The capability of checkpointing is bought with additional hardware resources and a decreased operating frequency. This section presents quantitative trends on these overheads for each proposed checkpointing mechanism. In addition, the results will be quantitatively compared with checkpointing using FPGA configuration readback.

Table 2.2 lists synthesis results for a set of benchmark modules [KBT07]. The
Figure 2.23: a) Flip-flop providing set, clear, and enable capability. b) adaptations with additional logic to allow checkpoint read or restore operations. These transformations are automatically performed by StateAccess.

checkpointing was implemented following the flow depicted in Figure 2.22 using the Altera Quartus-II design suite and Altera Cyclone FPGAs. The values denote the cost originating from the additional logic for accessing all flip-flops. The relative logic ($L_{LUT}$) and flip-flop overhead ($L_{FF}$) refers to the original module netlist (named "blank" in the table). All module flip-flops have been made accessible. Consequently, the values represent worst case results.

In addition to the synthesis results, the table also reveals estimated values according to the formulas presented in Section 2.6.3. These estimations are based on the number of flip-flops found in the unmodified module (e.g., for the DES56 module, $K = 862$). The data bus width is set to $B = 32$ bits for all experiments. The results point out that the estimates match the measured overheads well. In the worst case, the difference between the measured and the estimated logic overhead is 28% (found for the look-up table overhead of the DES56 module when applying the memory-mapped approach). The amount of additional LUTs for the FIR16 module is 1.44 times larger than the number of flip-flops found in the unmodified module when using a simple scan chain to access the state. This has two reasons: Firstly, the unmodified module has a high degree of fully used 4-bit LUTs (82%), hence, preventing to fit a larger amount of additional logic into the existing LUTs. Secondly, the FIR16 module possesses a very complex control structure with a huge amount of individual flip-flop enable signals.

The impact of inserting scan chains into FPGA designs on the netlist level for testing hardware modules was examined in [WGNH01]. That work focuses on just one approach similar to the simple scan chain presented in Section 2.6.3. But instead of using a ring shifter for keeping the state consistent, flip-flop enable logic was used. The reported results state on average look-up table overhead of 130% that is more than the worst case result for all examined checkpointing techniques in this chapter. Unfortunately, the paper does not list overheads on the achievable clock frequency.
2. Self-adaptive Reconfigurable Networks

Table 2.2: Synthesis results on flip-flop ($L_{FF}$), look-up table ($L_{LUT}$) overheads for the proposed hardware checkpointing methodologies.

<table>
<thead>
<tr>
<th>Method</th>
<th>flip-flops</th>
<th>4-bit LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>synth. results</td>
<td>overhead</td>
</tr>
<tr>
<td>DES56</td>
<td>blank</td>
<td>862</td>
</tr>
<tr>
<td></td>
<td>MM</td>
<td>903</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>928</td>
</tr>
<tr>
<td></td>
<td>SHC</td>
<td>1831</td>
</tr>
<tr>
<td>FIR16</td>
<td>blank</td>
<td>563</td>
</tr>
<tr>
<td></td>
<td>MM</td>
<td>576</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>583</td>
</tr>
<tr>
<td></td>
<td>SHC</td>
<td>1142</td>
</tr>
<tr>
<td>FFT1024</td>
<td>blank</td>
<td>5057</td>
</tr>
<tr>
<td></td>
<td>MM</td>
<td>5651</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>1124</td>
</tr>
</tbody>
</table>

Table 2.3: Reduction of the achievable clock frequency for the proposed hardware checkpointing methodologies.

<table>
<thead>
<tr>
<th>Method</th>
<th>DES56</th>
<th>FIR16</th>
<th>FFT 1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>113</td>
<td>22.7</td>
<td>27.7</td>
</tr>
<tr>
<td>MM</td>
<td>104 (-8%)</td>
<td>21.0 (-7%)</td>
<td>25.6 (-8%)</td>
</tr>
<tr>
<td>SC</td>
<td>111 (-2%)</td>
<td>21.7 (-4%)</td>
<td>26.6 (-4%)</td>
</tr>
<tr>
<td>SHC</td>
<td>94 (-17%)</td>
<td>21.5 (-5%)</td>
<td>24.8 (-10%)</td>
</tr>
</tbody>
</table>

$F_{max}$ or the flip-flops. Tiwari and Tomko reported an overhead of 46% in average for their benchmark circuits that have been extended with simple scan chains [TT03] for debugging purposes. The scan process in that work was controlled by gating the clock on Xilinx Virtex-II FPGAs. On these devices, clock gating requires a dedicated global clock tree and additional care must be taken because of the skew among different clock trees. Opposed to this, all presented techniques in this chapter are not touching any clock net and the overhead of the simple scan chain technique is only 33% on average.

In addition to the logic overhead, the impact on the maximal achievable clock frequency was examined for the benchmark. As stated in Table 2.3, no strong influence was found due to hardware checkpointing.
2.6 Operating System Services for Self-adaptive Networks

Checkpoint Overhead

For determining the checkpoint overhead $t_c$, and consequently, the time to take or to restore a checkpoint, the benchmark modules have been integrated into an SoC containing a CPU (Altera NIOS-II), a memory controller, and some further peripherals. Most commonly, FPGAs are logic bounded and typically not restricted to the amount of flip-flops in a circuit. This observation is confirmed by the benchmark circuits that require more than double the amount of LUTs than of flip-flops. Consequently, focus has to be put on $L_{\text{LUT}}$, when evaluating the different checkpointing techniques.

The measured checkpoint overheads are plotted together with the LUT overheads $L_{\text{LUT}}$ from Table 2.2 in Figure 2.24. For each test module, all results for the different techniques represent a Pareto point, hence, all techniques have their justification depending on the entire requirements (low checkpoint overhead or low logic overhead). The shadow scan chain technique is listed with two different $t_c$ values for the overhead for each module. The first value is zero, because of taking a checkpoint can be performed without stopping the module, while the second value denotes the time for transferring the state (average of checkpoint read and restore).

Checkpointing via FPGA Readback Techniques

As a reference, checkpointing with the help of configuration readback and bitstream manipulation can be quantitatively compared with the results in Figure 2.24. Assuming a Xilinx Virtex-II FPGA, the time required to read back configuration data is related to the number of configuration columns utilized by a module and, hence, to the module width. In the best case, when a module is constrained to a rectangle shape of minimal size and maximal height, the readback data contains 20 times more data than the data required to store just the flip-flop values (refer to Section 4.2 for further details on the bitstream format). This holds, if all flip-flops will be used, and if only the configuration columns containing register values are selectively read.

In the case of the DES56 core, for example, this overhead ratio is in the best case $\frac{1899}{862} \cdot 20 = 44$ because not all flip-flops provided by the FPGA were used. This means that instead of reading just the 862 flip-flop values of the module, it is necessary to read about $862 \cdot 44 = 37$ kilobytes of configuration data which takes at least 1 clock cycle per byte at the configuration interface. When assuming that it takes just 3 times this amount of clock cycles to filter out the flip-flop states in order to store and transfer the checkpoint efficiently, it requires at least 109 000 cycles to extract the state. Note that the readback configuration data needs to be manipulated before it can be used to reconfigure the device again, and that there is no trivial method on Xilinx FPGAs to preset just a submodule with an initial state during partial reconfiguration. Furthermore, the configuration readback approach demands to gate the module clock in order to keep the state consistent when a checkpoint rollback occurs.

In [LWH02] readback techniques are used to implement multitasking on a Virtex
Figure 2.24: Hardware checkpointing logic overhead $L_{LUT}$ over the checkpoint overhead $t_c$ in terms of clock cycles for different benchmark modules.
2.6 Operating System Services for Self-adaptive Networks

Figure 2.25: Two FPGA boards linked for remotely playing Pong. The synchronization between the boards is completely carried out by sending checkpoints between the boards. Note that this is possible despite of using completely different FPGAs.

FPGA. It is reported that it takes 407 ms to read back the configuration (766 kilobytes), 465 ms to extract the state information and another 365 ms to configure the device. These values confirm that the state access via the configuration port is relatively slow. It can be concluded that it is worth to spend some additional logic while reducing the latency overhead by at least one order of magnitude.

Hardware Checkpointing in Heterogeneous FPGA Environments

For demonstrating the capability of hardware checkpointing in heterogeneous FPGA environments, an existing implementation of the video game pong (a table tennis like game) was automatically extended with a simple scan chain using StateAccess. In this example, checkpointing is used to synchronize two gamers playing remotely on different nodes against each other, as illustrated in Figure 2.25. If the ball of the right player hits the right border of the screen or the racket, a checkpoint is sent to the FPGA board of the left player. The left side performs a rollback with this state that contains all necessary information to continue the game (e.g., score, ball direction). The analogous process happens for the left player at the left hand side of the screen.

By omitting all flip-flops from the register description that are used by the video output module of the game, the checkpoint data shrinks from 162 down to 57 flip-flops. The only modification required to the original pong specification is to generate a signal that indicates whenever a ball hits the left or right border. This signal triggers the actual checkpoint saving process. The back end synthesis step (see Figure 2.22)
was performed for FPGAs from different vendors. The synthesis results differ with respect to the amount of LUTs and flip-flops instantiated for the game. However, the checkpoints are still compatible for both FPGA boards. By using an additional RS232 compatible UART at 115,200 baud the checkpoint latency is below 5 ms, hence, allowing real-time gaming.

### 2.6.6 Design Flow for Implementing Morphable Tasks

For implementing morphable tasks for a ReCoNet, a design flow has been developed around the tool StateAccess, which has been introduced in Section 2.6.4. With this tool, hardware modules can be automatically extended with logic for accessing all or a definable part of their internal flip-flops. The tool includes also a FSM into the module to control the checkpointing, which, in particular, allows to trigger the checkpointing with signals from the module itself, which will be used to take checkpoints only if the hardware module is in a morphpoint.

The design flow for implementing morphable tasks is illustrated in Figure 2.26. Based on a C++ specification of the task functionality, the software binary is compiled. In addition, an RTL model is synthesized with the help of a behavioral compiler that uses the same C++ specification. Note that the behavioral compiler is capable to build design alternatives that may be used for morphing between multiple hardware variants. With multiple hardware implementation alternatives, the system may adapt more flexible to environmental changes with hardware modules that are optimized for different scenarios (e.g., throughput demands).

For each RTL model, StateAccess includes additional logic into the module for accessing its internal state. This can be controlled in multiple ways. Besides selecting between various options for accessing the state, the logic generation can be further controlled with a register description file. By modifying this file, StateAccess can be guided to only include a subset of the flip-flops into hardware checkpoint. Furthermore, it is possible to sort the bitvectors such that there is no bit shuffling required during morphing at runtime. Note that the identifiers of the original C++ specification will be conserved within the RTL model as well as in the register description file. The result of the transformation by StateAccess is a modified netlist that is used together with templates (for interfacing the hardware checkpointing logic with the CPU) for generating the final module configuration bitstream.

### Automatic Hardware/Software State-Translation

The final register description file is used to automatically generate morph functions. This is obtained by firstly scanning the file for figuring out the number, names, and sizes of all bitvectors occurring inside the hardware checkpoint ($HW_{CP}$). With this information, a struct storing the software checkpoint ($SW_{CP}$) is generated that combines all different bitvectors such that they are represented by the smallest fitting
2.6 Operating System Services for Self-adaptive Networks

Figure 2.26: Design flow for implementing hardware/software morphable tasks. Starting from a task function, the task binary (software implementation) as well as an RTL model (hardware implementation) is generated. The RTL model is automatically transformed for taking and restoring hardware checkpoints with the help of StateAccess. StateAccess further provides a register description file that is indicating the flip-flop mapping within the checkpoint. An example of such a register description is given in the next section. With this information, morph functions for translating the state encoding between hardware and software execution are generated. Together with some predefined templates for carrying out the interprocess communication, the binary for the hardware task will be generated.

( unsigned ) integer type. The following register description file denotes an example for a hardware checkpoint consisting of two registers ( Reg_A and Reg_B ) that are each two bit wide ( reg_0 to reg_1 ):

```
HDL_hierarchy_Reg_A_reg_1 // LSB of HW_CP
HDL_hierarchy_Reg_B_reg_0
HDL_hierarchy_Reg_B_reg_1
HDL_hierarchy_Reg_A_reg_0 // MSB of HW_CP
```

The generated struct representing the corresponding software checkpoint is:

```
struct SW_checkpoint {
  unsigned char Reg_A;
  unsigned char Reg_B;
```
After this, the actual state translation functions are generated:

```c
SW_CP translate_to_SW(HW_CP) {
    SW_CP.Reg_A = 0;
    SW_CP.Reg_B = 0;
    SW_CP.Reg_A = SW_CP.Reg_A | ((HW_CP & 0x01) << 0x01);
    SW_CP.Reg_B = SW_CP.Reg_B | ((HW_CP & 0x06) >> 0x01);
    SW_CP.Reg_A = SW_CP.Reg_A | ((HW_CP & 0x08) >> 0x03);
}
HW_CP translate_to_HW(SW_CP) {
    HW_CP = 0;
    HW_CP = HW_CP | ((SW_CP.Reg_A & 0x02) >> 0x01);
    HW_CP = HW_CP | ((SW_CP.Reg_B & 0x03) << 0x01);
    HW_CP = HW_CP | ((SW_CP.Reg_A & 0x01) << 0x03);
}
```

The state translation is carried out with mask and shift operations completely in software. The entire bit-masks are adapted to the maximal amount of correctly ordered consecutive bits of a subvector within the hardware checkpoint \( \text{HW}_{\text{CP}} \). This helps, for example, to translate the state of \( \text{Reg}_B \) in just one line. By moving the first line of the register description file to the end (before synthesizing the hardware module) also \( \text{Reg}_A \) can be translated in a single line.

The integration of the translate functions into the software task has currently to be completed manually. A further manual step is required if a bitvector represents an enumerated type. In this case, a state encoding look-up table has to be extracted from the report files, generated by the hardware synthesis tool, and integrated as well. Following these steps, tasks for hardware/software morphing can be implemented almost fully automated.

2.6.7 Morphing Case Study

In this section, the implementation and experimental results of applying hardware/software-morphing to a FIR-filter module will be presented. It will be exemplarily demonstrated that morphable tasks can be automatically implemented starting from the same C++ specification of a function that contains the functionality of a task. A FIR-filter has been chosen, because it contains an internal state that is changed with every invocation of the filter. Furthermore, it can be easily compiled to different implementation alternatives with the behavioral compiler. A hardware task in a ReCoNet comprises a hardware module that is mapped into the address space of the CPU and a corresponding software task that, in particular, provides the interface to the message passing system. For testing hardware/software morphing on a single ReCoNode at runtime, a 16-tap FIR-filter task with constant coefficients and a 32-bit wide data path was implemented. Upon receiving a new input value via the message
2.7 Design, Implementation, and Evaluation of a ReCoNet Protocol

passing system, the task computes the next output value either in software or in hardware. For the latter one, a module is used that is implemented following the flow presented in the last section.

After the behavioral synthesis of the C++ specification, the resulting RTL model uses 563 flip-flops and 1223 look-up tables to implement the filter. After applying a constraint for keeping all tap registers that represent the actual state of the filter and after modifying the netlist to allow hardware checkpointing with StateAccess, the module requires 571 flip-flops (+4%) and 1890 LUTs (+54%). For the experiment, the flip-flops of the module have been made accessible using the simple scan-chain technique (see also Page 2.6.3). The achieved clock frequency is in both cases 20 MHz when implemented on an Altera EP2C20-7 FPGA. The test system provides a NIOS-II softcore CPU that requires 1559 flip-flops (not including the register file that is implemented in block-RAMs) and 2718 LUTs. The CPU and bus clock frequency in this system is 50 MHz while the FIR-filter module runs with a separate clock of 20 MHz. Synchronization between the two clock domains is achieved with a simple handshake protocol.

The measured execution time of processing the filter as a software task is $t_{SW} = 60\mu s$. When the task is morphed to hardware the execution time decreases to $t_{HW} = 2.7\mu s$, which results in a speed-up of 22. The values are acquired with a hardware timer for precisely measuring the processing time for one message. Similarly, the time was measured that is spent for morphing between software and hardware execution, which is $t_{\text{morph}} = 104\mu s$. Most of this time is required to copy the state between the two implementation alternatives. Note that different implementation alternatives of the FIR-filter module would result in the same time, because the number of flip-flops and the position within the checkpoint would be identical.

In this example, it will take two software task activations before the CPU load will be reduced by the accelerator. Note that all variables have been identically specified as 32-bit integer values for the hardware as well as for the software implementation, thus, no translation of data items between the hardware and software domain is required. When omitting any other latencies caused by the operating system, the morphing is performed within $L = t_{\text{morph}} + \max\{t_{HW}, t_{SW}\} = 164\mu s$.

2.7 Design, Implementation, and Evaluation of a ReCoNet Protocol

Traditional distributed control systems use field buses and newly more and more Ethernet variants to link the different control units to a network. These might be applied to implement a ReCoNet, but restrictions on the topology, latency, throughput, or cost may prevent to make full use of the self-repairing capabilities of the ReCoNets approach. Consequently, a dedicated protocol stack, called the ReCoNets protocol
has been developed, as revealed throughout the following sections in a bottom-up manner. This protocol defines multiple layers, starting from the physical layer up to interprocess communication for node-independent task-to-task communication. The protocol is tailored to small wire-linked point-to-point networks and provides efficient low latency communication, especially for sensor or actuator data. As demonstrated in Section 2.7.5, this specification permits a very scalable implementation of the corresponding router hardware.

**2.7.1 Physical Layer**

The following paragraphs reveal the implementation of the physical layer of the ReCoNets protocol, which includes the electrical interface and the synchronization of the serial data streams between the ReCoNodes.

**2.7.2 Electrical Interface**

As many embedded applications are highly optimized for cost, the physical layer of the ReCoNet was specified by avoiding expensive components, like for instance, shielded wires or dedicated line drivers. Consequently, the point-to-point links are directly connected via plugs to FPGA I/O pins. Only additional termination resistors have been integrated for absorbing reflections. Each signal link provides two physical twisted cable pairs of unidirectional links that are simultaneously used in **full-duplex** mode.

**Synchronization**

If data is transferred from one node to another it is necessary to synchronize the communication because there is no global clock assumed that is supplied to all the nodes in a ReCoNet. This requires to synchronize the communication among the different ReCoNodes. For serial data transfer, the following three synchronization techniques are widely used (see also Figure 2.27):

1. **Sending an extra clock** together with the data. This is used mainly in simple communication interfaces (e.g., I²C, SPI).

2. In **constant-weight Coding**, the data is sent such that every code word contains a certain amount of positive and negative levels. Examples for this encoding are SPDIF (Sony Philips Digital Interface) under the name **bi-phase modulation**, or HDLC, or 10Base Ethernet under the name **Manchester-Coding**.

3. **Time driven**. This technique is used, for example, in RS232. It makes use of the fact that the absolute drift between the clock domains of two nodes is very low for short time ranges. Therefore, the clock domains are virtually identical.
2.7 Design, Implementation, and Evaluation of a ReCoNet Protocol

The transfer of a data item starts with a predefined start bit or preamble and data is sent at a fixed defined rate.

The coding in technique 2 can be done in such way that the send signals mean value is zero. This is important if signals should be coupled inductive or capacitive or if the signal is modulated on a power line that is all not the case in the ReCoNet where the Nodes are directly linked from one FPGA-pin to another without additional line drivers. In addition a Manchester-Coding decreases the available throughput dramatically. Whenever sending a sequence of identical bits, two transitions per bit have to be sent to the receiver (1B2B code). Therefore, time driven line coding is chosen for the ReCoNet as it is simple and efficient.

In real life, the signal slopes are not as idealistic as shown in Figure 2.27 c). Therefore, at the receiver, the input data is sampled in the middle of a particular bit, where the signal is most stable. However, as the oscillators may differ between the sender and the receiver, a synchronization detection circuit is used to keep the sample point in the middle of the bit window. Figure 2.28 illustrates how oversampling is used to track the sample point.

The sample point tracking can only be performed on the signal slopes of the input data. Hence, it is important to limit the amount of consecutive equal bits. In the CAN protocol, this is assured by sending a sixth out of order bit after a sequence of five consecutive identical bits. For the ReCoNet protocol, every 16th bit is sent twice, the first one directly and the second one (the synchronization bit) inverted. The synchronization bits are further used for error detection within the receiver.

The output signals SyncTooFast and SyncTooSlow of the synchronization process state a precise indicator value for the relative clock drift between two adjacent neighbor nodes. The firing rate of the signals SyncTooFast and SyncTooSlow at a node $n_i$ with the system frequency $f_i$ is in the case of a communication with node $n_j$ running with the frequency $f_j$ (with $n_i, n_j \subseteq N$):

\[
  f_{(\text{SyncTooFast},i,j)} = \begin{cases} 
    f_i - f_j : & f_i > f_j \\
    0 : & \text{else}
  \end{cases} \quad (2.3)
\]

\[
  f_{(\text{SyncTooSlow},i,j)} = \begin{cases} 
    f_j - f_i : & f_i < f_j \\
    0 : & \text{else}
  \end{cases} \quad (2.4)
\]
2. **Self-adaptive Reconfigurable Networks**

![Diagram](image)

**Figure 2.28:** a) Synchronization detection circuit for tracking sample points in a serial data stream. b) Possible scenarios during sampling the input stream. The table lists the actions for the different scenarios.

<table>
<thead>
<tr>
<th>Synchronization decode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Rx_1 = Rx_2 = Rx_3$</td>
<td>Scan Speed OK</td>
</tr>
</tbody>
</table>
| $(Rx_1 = Rx_2) \neq Rx_3$ | The Scan Speed was too fast  
   $\rightarrow$ include an extra delay cycle until the next scan enable |
| $(Rx_1 \neq Rx_2) = Rx_3$ | The Scan Speed was too slow  
   $\rightarrow$ remove one delay cycle until the next scan enable |
| $(Rx_1 = (Rx_3) \neq Rx_2)$ | The Scan fails  
   $\rightarrow$ trigger exception |

Note that these firing rates belong to one specific port and that every port has its own synchronization detection. If there is a specific firing rate in node $n_i$, it is known that the adjacent node $n_j$ will have a firing rate of the complementary signal:

\[
f(SyncTooFast,i,j) = f(SyncTooSlow,i,i) \\
f(SyncTooSlow,i,j) = f(SyncTooFast,j,i)
\]  

(2.5)

(2.6)

For further network analysis, is it advantageous to combine the firing rates $f_{SyncTooFast}$ and $f_{SyncTooSlow}$ to one combined drift value:

\[
f_{d(i,j)} = f(SyncTooFast,i,j) - f(SyncTooSlow,i,j) \\
f_{d(i,j)} = f_i - f_j
\]

(2.7)

One interesting property of the drift value that may be used for time synchronization in a network is that it is zero along a cyclic path:
2.7 Design, Implementation, and Evaluation of a ReCoNet Protocol

The drift value originates from the variation of the local clock oscillators on the ReCoNodes. A typical crystal oscillator has an error of ±100 ppm, which is ±0.01% [NB97]. Consequently, in the worst case, the drift between two ReCoNodes is ±200 ppm. When assuming that the sample point is not allowed to drift no more than 10% of the bit send period, the longest acceptable sequence of identical all-one or all-zero values is \( \frac{10^6}{200} \cdot \frac{1}{0.1} = 500 \) bits. Therefore, the introduction of a synchronization bit after every 16th bit ensures safe operation also if data transmission is afflicted with an additional jitter or when using less stable oscillators.

2.7.3 Frame Format and Node Addressing

Data within a ReCoNet is transferred in frames. Each frame contains a start bit followed by the node destination address, control flags, the payload, a checksum, and a stop bit, as depicted in Figure 2.29. In addition, a synchronization bit is additionally sent after every 16th bit. The frame format is kept small for enhancing the efficiency, especially when transferring simple sensor or actuator data in a ReCoNet. The efficiency is defined as the payload size divided by total size of a frame. So far, two payload sizes have been defined, one for short packets\(^{10}\) with 48 bit payload (56% efficiency) and one with 176 bit payload (79% efficiency). The synchronization bits limit the efficiency to \( \frac{16}{17} = 94\% \).

Short frames possess only 24% of the amount of bits (including payload) that are required only for transferring the headers of an IP packet embedded into an Ethernet frame, also called cell, contains a header, the payload, and a trailer and is processed by the data link layer. A packet is generated in the network layer and placed into the payload field of a frame.

\[^{10}\text{A frame, also called cell, contains a header, the payload, and a trailer and is processed by the data link layer. A packet is generated in the network layer and placed into the payload field of a frame.}\]

\[2.79\text{ Frame format of the ReCoNet protocol. After a start bit, the header field, containing the destination node address and the flags is sent. Only these values are interpreted by the router hardware.}\]
2. **Self-adaptive Reconfigurable Networks**

frame. When sending at the same gross rate, the ReCoNet protocol is capable to transmit up to four times more packets.

**Flags**

The ReCoNet protocol specifies various flags which are evaluated by the hardware router and the software protocol stack. In particular, the following flags have been specified:

- **Size**: The size flag is used to set the payload size between 6 byte (short packet) and 22 byte (long packet).

- **Priority**: Messages can be sent with two different priorities in a ReCoNet. Whenever available, the router will send high priority frames before low priority messages.

- **Multi Cell Packet**: The ReCoNet protocol provides a packet type for efficiently transferring larger amounts of data that exceed the capacity of a single cell in a ReCoNet. Within the payload field of a long frame, a multi cell packet contains a packet identifier (identical for a complete packet) and a sequence identifier that is incremented for each cell. This protocol is suitable for alternatively transferring large data items (up to 8 MB) directly by the network hardware using Remote Direct Memory Access (RDMA). This flag is evaluated at the network layer.

- **Time-Sync**: For providing precise time synchronization between the nodes of a ReCoNet, time stamps can be directly included by the network hardware. This allows determining offset and drift values with a very low jitter.

So far, only 4 out of 8 possible flags have been specified. The remaining flags can be used for providing more transfer modes and parameters. For example, by using two flags for the priority, four instead of one service levels can be implemented.

**Node Addressing**

With an 8-bit destination address field, more than 200 nodes can be integrated into a ReCoNet. Some addresses have been reserved for special purpose. For instance, instead of addressing a node by its destination address, there exists an alternative method to send frames to a node when the destination address is unknown. This is the case when a new ReCoNode is integrated into the system at runtime. For this reason, the first addresses (0...7) will route a packet directly to the corresponding output port. As illustrated in Figure 2.30, *direct port addressing* can be used to exchange data upon establishing a new link without knowing the target node address of the packets.
2.7 Design, Implementation, and Evaluation of a ReCoNet Protocol

Figure 2.30: Direct port addressing. After establishing a link between Node 21 and Node 22, Node 21 can send a message to the new neighbor using direct port addressing without knowing the destination node address (22). At the receiver, the destination port address is replaced with the own port address, such that higher network layers can identify at which port the message was received.

Checksum

All frames contain a checksum for detecting transmission errors. The computation of the checksum follows the recommendation of the CCITT (Comité Consultatif International Télégraphique et Téléphonique) and is based on a 16-bit Cyclic Redundancy-Check code (CRC). Therefore, bit failures are detected with a probability of $1/(2^{16} - 1)$ or $1.5 \times 10^{-5}$. The generator polynomial is:

$$CRC_{CCITT}: X^{16} + X^{12} + X^5 + 1$$

Many safety-critical embedded systems require high Hamming distances. For automotive X-by-Wire applications, the FlexRay protocol specifies a 24-bit CRC polynomial to generate a code that has a minimum Hamming distance of 6 for code words with up to 2048 bits (data + CRC) in length [Fle05]. This allows to detect up to 5 arbitrary bit errors. In general, the minimum Hamming distance depends on the data word length, the length of the CRC checksum and the generator polynomial used. In the ReCoNet protocol, the data word length consists of a 16-bit header and either 48 bit payload (short packets) or 176 bit payload (long packets). With the 16-bit checksum this results in 80-bit and 208-bit code words. Ray and Koopman analyzed the Hamming distance for multiple generator polynomials in [RK06]. The selected CCITT CRC guarantees a minimum hamming distance of 6 for code words with up to 151 bits, hence, short packets fulfill the same Hamming distance requirement as demanded for the FlexRay protocol in X-by-Wire applications. The hamming distance of the long packets, specified in the ReCoNets protocol, is 5 (see [CGG90] for more details on 16-bit redundancy check codes).

2.7.4 Task-to-Task Communication

The lower level network services are required for providing communication among the nodes of a ReCoNet. This section reveals all further protocol layers that have
2. Self-adaptive Reconfigurable Networks

been specified for the communication among the tasks located on the ReCoNodes. The task-to-task communication in a ReCoNet is fully transparent with respect to the ReCoNodes hosting the tasks. In other words, the tasks are not aware of their own placement position in the network or the locations of their communicating partners. Note that operating system services, running in each ReCoNode, distribute the tasks in the network according to multiple objectives, including the hardware and software load on a node, the available capacity of the ReCoNet links, or the reliability of the total system [Str08].

**Task Resolution**

All tasks in a ReCoNet possess a unique identifier, that is defined at design-time. A task resolution table is used to map the entire task identifier to the corresponding node address. A copy of the table is stored in each node. When a task is migrated from one node to another, this information is broadcasted to all nodes for updating their local task resolution table. The same happens if a replica task becomes active during the fast repair phase in case of a fault. This process takes some time and packets may still be sent to the original node that, if available, relays the traffic further to the new destination. Note that a task migration will not touch the routing tables in the router hardware.

Rerouting, packet loss, or changing the task resolution table can change the receive order of the packets. In all other cases, the packet order will be maintained between sending and receiving ReCoNodes. By adding a sequence number to each packet at a source task, the corresponding destination task can detect and compensate packet disordering.

**Message Passing Interprocess Communication**

Interprocess communication of the tasks in a ReCoNet is carried out via Message Passing. The tasks contain a set of message handlers that implement the functionality. A task processes at most one message at a point of time, and messages are executed in a sequential manner. Each task holds an individual message input queue and a task may generate any desired amount of messages to be sent to other tasks, for example, a value to a set of actuator tasks from a controller task. Communication with the drivers is also carried out via the message passing system. The strict sequential processing of messages makes semaphore mechanisms in most cases unnecessary. For example, if a printer driver receives its jobs from the message passing queue it will automatically ensure that print jobs are handled atomically.

The message passing communication is decoupled from the task execution and the communication as well as the execution is carried out asynchronously. Note that multiple tasks can run concurrently on a single ReCoNode as there exist multiple parallel physical execution units, for example, multiple hardware tasks. If the sending
and receiving tasks are located on the same node, messages between them will be
directly sent to the respective local message queues. If a receiver task is located on a
remote ReCoNode, the message is relayed via the ReCoNet communication protocol
to the destination task with the help of the task resolution table.

The arguments of a message includes a message identifier and a variable amount
of parameter values or pointers to data structures. The message identifier allows to
distinguish between different message sources, as, for example, a trigger message
from a timer source or a message containing a sensor value. This allows to selec-
tively activate corresponding task handlers. If a message containing pointers to data
structures (e.g., a frame of a video stream) is transferred to another ReCoNode, the
data structures will be copied before the message. Then the message will follow and
the pointer addresses will be updated at the destination with the new memory posi-
tions of the structure at the destination ReCoNode. The complete process is handled
by the operating system completely transparent to the tasks of a ReCoNet.

Task-to-Task Communication Protocol Stack

The task-to-task communication is implemented in multiple layers, as revealed in
Figure 2.31. A message generated by a task is transferred to the message passing
system (\textit{T2T message}) running on each ReCoNode. If messages have to be sent
to a local task, this is carried out directly by the message passing system. All other
messages are passed to the network protocol stack. The task-to-task protocol (T2T
is responsible for a reliable message transfer and sends the messages further, either
using the multi cell protocol (MCP) or the single cell protocol (SCP), depending
on the message size. The multi cell protocol is capable to transfer data items of up
to 8 MB. The cell protocol (CP) is responsible to generate the frames to be sent via
a driver to the router hardware. At the receiving ReCoNode, messages are picked
up by the cell protocol, then optionally recomposed by the multi cell protocol, and
passed on the task-to-task protocol. The latter one has to acknowledge the correct
data reception to the sender. On a missing acknowledge, the sender retries to transmit
the message up to three times. The task-to-task protocol generates a message at the
receiver side and passes it to the message passing system. Additionally, the task-to-
task protocol keeps track of the last messages for preventing that the same message
is sent multiple times to a task (for example, if an acknowledge packet got lost).

Figure 2.31 depicts some further elements in the protocol stack. The routing and
task resolution protocols have been revealed in Section 2.5.3. For providing a global
time base in the ReCoNet, a node synchronization protocol is implemented that gen-
erates special time stamp packets that are modified inside the hardware routers with
the corresponding local send and receive time stamps for compensating offset and
drift of the local clock on each ReCoNode.
2. Self-adaptive Reconfigurable Networks

<table>
<thead>
<tr>
<th>Task 1</th>
<th>...</th>
<th>Task N</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2T_message</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2T: Tast-to-Task</td>
<td>TRP: Task Resolution</td>
<td>Route</td>
</tr>
<tr>
<td>CP: Cell Protocol</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.31: Network protocol stack of the ReCoNet protocol.

### 2.7.5 Implementation of a ReCoNet Router

The main objectives in the router design are portability, low latency packet forwarding (*cut-through routing*), and scalability. The latter one includes an adjustable amount of ports, variable FIFO sizes and a tunable amount of routing table entries. All these parameters are defined at design time.

Figure 2.32 depicts a block diagram of the router architecture. All packets are switched with the help of a fully interconnected crossbar, hence, simultaneous transfers are possible to all output ports. By using two input FIFOs, one for high and one for low priority packets for each port, high priority packets can overtake low priority ones.

On arrival of a new packet, a look-up in the routing table is carried out directly after receiving the target node address (located in the beginning of the packet). This may take multiple clock cycles, because all ports share the same routing table and the table look-up is serialized. Depending on the priority flag, the packet will be queued into the corresponding high or low priority scheduling table of the destination port that was determined by the routing table look-up. Sending the packet further may start immediately, if no concurrent packets exist for the entire destination output port (*cut-through routing*). Only if a packet arrives to the network via the local system bus, the transfer is delayed until the full packet has arrived in the input FIFOs of the router. This ensures that a packet can be sent atomically through an output port regardless of whether the local system is interrupted during the FIFO fill process.

**Router Synthesis Results**

The router has been synthesized as part of an SoC for different configurations. Figure 2.33 denotes the implementation cost of the router hardware, when implemented on the main FPGA (a Xilinx Virtex-II 6000) of an ESM [MTAB07]. All queue FIFOs have been implemented with a dedicated memory block each, hence it demands three block RAMs for the connection to the local system and an additional amount of two block RAMs for each port.

The link control, send, and receive logic is instantiated separately for each external port. This causes the majority of the logic cost, thus, leading to a mostly linear increase of the total logic cost with respect to the number of external ports.
Figure 2.32: Block diagram of the ReCoNet router hardware. Each port provides its own send and receive logic, link control logic for connection establishment, and a pair of input queues (FIFOs). The router contains a crossbar for passing data from a receive queue directly to a send port. The local system behaves to the network like an external port. The block main control & routing serializes the routing table look-up, manages the crossbar settings and provides the control and interrupt logic of the router. Additionally, this block contains two independent round-robin schedulers for each external send port and the input receive queue in order to guarantee fairness in the case that multiple sources (receive queues) want to send data to the same output port. CRC checksums are computed in front of the input queue to the local system and in each send and receive port separately. The value of the local clock, which is not part of the router, is connected to the control logic blocks of all receiving and sending ports for low jitter time stamping.
logic cost of a three port router is with 3056 slices roughly about the logic cost of a fully featured Microblaze softcore implementation on the same FPGA or 9.8% of the logic resources available on the main FPGA of the ESM. Logic efficiency was not the driving objective in the implementation of the router. However, if compared with the logic cost of the OPB Ethernet Lite Media Access Controller [Inc06], which is provided by Xilinx as part of the EDK IP-core library, it turns out that one port of the ReCoNet router requires about the same amount of logic as this library core (455 slices).

### 2.7.6 Experimental Evaluation of a ReCoNet

Different experiments have been undertaken to evaluate a ReCoNet. The following sections reveal results on throughput, latency, and on link failure repair times. All results are based on the ReCoNet implementation that is presented in Section 2.7.

#### Throughput

The task-to-task communication as well as the self-organizing capabilities of a ReCoNet require a certain amount of bandwidth within the network. For instance, when transferring video data or in case of a task migration where binaries or configuration bitstreams are sent via the network.

Larger data items are sent with the help of the *multi cell protocol* that fits fractions of a large packet into multiple long cells (see Section 2.7.4). Using this protocol requires altogether 6 bytes header information in the 22 byte payload field. Considering the 79% efficiency when transferring a long frame, this results in a usable

---

**Figure 2.33:** Hardware cost in slices (logic primitive providing two 4-bit look-up tables) and 18 Kb memory blocks over the number of router ports.
Figure 2.34: Measured throughput using the multi cell protocol (MCP). The values are achieved for two tasks that are located on different ReCoNodes. The nodes run a NIOS-II softcore CPU at 50 MHz and host only the respective sending or receiving task.

Figure 2.35: Experimental setup for high throughput measurements. As a single node is not able to fully utilize the capacity of a single ReCoNet link, multiple nodes are used to increase the throughput demand on the link between node 3 and node 4.

peak throughput of $79\% \cdot \frac{22 - 6}{22} = 57.5\%$. With the currently specified 12.5 MBd it is therefore possible to transfer up to 877 KB/s. However, as depicted in Figure 2.34 this throughput is not achievable directly between two ReCoNodes as processing the protocol stack produces a substantial overhead.

When running the throughput experiments, no packet losses have been detected. However, by using six ReCoNodes in a chain, as illustrated in Figure 2.35, for generating three times the 400 KB/s peak throughput demand on a single link, high packet losses have been detected. However, no packet losses have been observed for high priority packets, that constituted 50% of the traffic.

Latency

In Section 2.5.4, the maximum latency for passing a frame through a router was calculated to be 19.2 µs. This value has been verified with an oscilloscope connected to
Figure 2.36: Ping response times. The values are obtained by observing router status signals in the node that is initiating the ping message. Consequently, the time values reveal the ping packet transmit time to the target, the time spent for processing the ping in the target, and the time to deliver the answer but not the time to generate or to verify the answer in the source node.

The measured latency is between 4 and 19 µs with 12 µs on average. Note that these values are obtained on single packets without any other traffic in the network. Other traffic will increase this latency or even result in packet losses in case of buffer overflows.

A long frame requires 18 µs to be fully received or sent via a router port. Consequently, the cut-through routing reduces the average hop latency by \( \frac{18\mu s - 12\mu s}{18\mu s} = 33.3\% \). This benefit will be much higher when sending longer frames.

Furthermore, the plot reveals the latency needed to respond to a ping packet in a ReCoNode. This value can be determined by extrapolating the curve to a distance of 0 hops. The resulting value is slightly below 300 µs and includes the complete time starting from the arrival of the first bit in the router, over processing the protocol stack, up to the transmission of the first bit through the router output.

Rerouting

The time required for processing packets is further important for flooding network state changes among the ReCoNodes. All network state data between the nodes is exchanged using direct port addressing (see Section 2.7.3) because there is no routing
information available at network start and it cannot be assumed that the normal node addressing scheme transfers packets reliably to the destination in case of a link or node failure. Consequently, the network state distribution invokes the CPU on each hop.

A distance vector routing algorithm (Bellman-Ford) as well as Dijkstra’s algorithm have been implemented for the ReCoNet. Both algorithms are only triggered on a particular node if this changes the port cost table (Bellman-Ford) or the adjacency matrix (Dijkstra). The cost table stores the port to be chosen to reach a particular node and the number of hops (the distance) to the destination. The initial implementation of the Bellman-Ford algorithm was unstable when an update of the port cost table had immediately triggered a change of the table entries in the router hardware. In this case, cycles can be temporarily switched which may result in additional delays or packet losses. The problem was solved by synchronizing the route table update process of all ReCoNodes. However, this implies additional overhead as it must be ensured that all nodes have gotten the network state for updating the local cost table before globally updating the route tables.

As an alternative, exchanging full adjacency matrices for computing new routes with the help of Dijkstra’s shortest path algorithm was examined. As available links provide symmetric full duplex operation, only the upper triangular matrix has to be sent. The size of this matrix scales quadratically with the number of ReCoNodes: $|V|(|V|-1)/2$, and is 620 bytes in a ReCoNet with $|V| = 100$ nodes. Before computing the shortest path on arrival of a new adjacency matrix, the new network state is flooded to all adjacent nodes except to the one that is the source of the new network state. Each ReCoNode keeps track of received state messages. Multiple receptions of the same state message are discarded.

This protocol works stable and is able to set up a new route faster than the distance vector alternative. Figure 2.37 depicts the rerouting time as a function of the network diameter. In the experiments, most time is spent on flooding the new network state over the ReCoNet. This is because of the direct port addressing scheme that can send the state at maximum one hop further, the rerouting time scales linear with the network diameter. The measured time per hop is 0.88 ms [Str08].

2.8 Chapter Summary

In this chapter, it was demonstrated that the unique capability of partial runtime reconfiguration provided by certain FPGAs can be used for self-adapting and self-repairing a ReCoNet at runtime. This was shown at the network level by migrating hardware and software tasks among multiple nodes as well as at the node level by providing the ability of dynamically integrating hardware and software tasks on a single ReCoNode [HKT03a, SSK07, SKHT06]. Additionally, with the introduction of hardware/software morphing, the flexibility was further enhanced by allowing to
2. Self-adaptive Reconfigurable Networks

Figure 2.37: Rerouting time over the network diameter. Taking the experimental setup, illustrated in Figure 2.38, the rerouting time has been determined for different network diameters. For each diameter, 25 measurements have been performed and the figure plots the average values together with the best case and worst case values. The results are taken from [Str08].

Figure 2.38: Experimental setup for measuring the time required for rerouting in a ReCoNet. In this example for a network diameter of $D = 3$, a link failure is introduced between the first two nodes. The rerouting time is measured between node 2 that detects the link failure and the last node in the chain (here node 4) by observing status signal pins on the ReCoNodes.
change the execution style of a task between different implementation alternatives at runtime [KHST07]. These novel techniques have been introduced based on a formal model, that, in particular, provides state dependent migration of both, hardware and software tasks, as well as a state dependent swapping of the execution between hardware and software implementations of a task. All these techniques can be transparently applied to the application tasks that may be migrated or morphed without losing internal states, hence seamlessly providing their service even when balancing load between different nodes or optimizing the partition between hardware and software load (used FPGA area or CPU time). This is supported with sophisticated operating system services for managing the migration, morphing, and load balancing autonomously without any interaction by a user. Furthermore, a dedicated network protocol was developed, analyzed, and implemented that is tailored to the requirements of a wide range of distributed embedded control system such as automotive, avionics, or building automation [KSD06]. This includes low latency cut-through routing, a raw bandwidth of 12.5 MBit, real-time and quality of service characteristics, efficient support for event and periodically driven data, and low cost physical connections. Additionally, this protocol is customized to the special demands of a fault-tolerant ReCoNet for allowing fast link state detection, self-rerouting, and for providing a task-to-task communication API that abstracts the application from the underlying network topology, thus, providing communication among the user tasks independent to the present location of the tasks in a ReCoNet [Dit05, HKT03a, HKT03a].

All these techniques have been integrated in the ReCoNets demonstrator for proving the feasibility of the self-adaptive and self-repair methodologies based on an automotive application example [SKHT06, CS1]. As revealed in Figure 2.39, each ReCoNode of the demonstrator provides an Altera Cyclone FPGA. These devices do not support partial runtime reconfiguration and reconfiguration was emulated by implementing multiple design alternatives in parallel that are activated according to the present task binding in this ReCoNet.

For providing partial self-reconfiguration at runtime, the Erlangen Slot Machine (ESM) [MTAB07] has been integrated into the demonstrator, as depicted in Figure 2.40. The ESM platform is a modularized design consisting of a baby-board – providing a large FPGA, six banks of fast SRAM, and configuration logic – and a main-board containing the periphery. The large main FPGA is a Xilinx Virtex-II 6000 device featuring 6 million system gates [Inc05b] The Virtex-II family supports dynamic partial self-reconfiguration and multiple configuration options are available on the ESM including serial JTAG, parallel select map via a dedicated configuration controller and the internal configuration access port (ICAP).

The original ESM main board was developed with emphasis on multimedia applications and a new main board tailored to the demands of a ReCoNet has been developed. This board includes multiple connectors for the ReCoNet point-to-point network and the peripherals required for the automotive application. Due to the modularized design of the ESM, the complexity of integrating an FPGA device possessing
2. Self-adaptive Reconfigurable Networks

Figure 2.39: The ReCoNets demonstrator. Four control nodes, each equipped with an FPGA, host multiple hardware and software tasks from the automotive domain in a ReCoNet. Among other things, this system features a driver assistance application for acoustically notifying the driver in case of an unintended lane change. For this reason, the current lane on the street is detected by evaluating a video stream after segmentation and by combining the result with the state of the direction-indicator control. The application is partitioned in multiple sensor, controller, and actuator tasks that are distributed to the ReCoNodes. Note that some actors can be controlled redundantly from two nodes, e.g. the speakers can be controlled by node 21 and node 22. In this system, link defects can be automatically compensated by rerouting traffic over alternative paths. Additionally task or node defects can be self-repaired with the help of task migration and hardware/software morphing. While sensor and actuator tasks are bound to the corresponding connected nodes, all control tasks can be arbitrarily migrated within this ReCoNet.
1517 pins was encapsulated by the baby board and the PCB of the main board for the ReCoNet has been implemented with just two layers.

With this ESM, the integration of hardware tasks as well as task morphing between hardware and software execution can be implemented on a ReCoNode while keeping the rest of the system active. However, this supposes the existence of 1) adequate methods for integrating hardware modules into a system at runtime and 2) a design flow including sophisticated tools for building according systems. In particular, the first aspect comprises techniques to provide communication with partial reconfigurable modules that should be efficient (low resource overhead), high performance (high throughput, low latency), and that maintain the flexibility of the underlying FPGA architecture to the application. All these aspects will be covered in Chapter 3 that reveals among other things a novel communication architecture that is capable to integrate dozens of reconfigurable modules into a system at the same time. Accordingly, Chapter 4 will cover the design flow and tool aspects required to build runtime reconfigurable systems. Such architectures and tools pave the way for suitably exploiting runtime reconfiguration in future practical systems.
2. Self-adaptive Reconfigurable Networks

Figure 2.40: ReCoNets demonstrator with an ESM. The ESM consists of a mother boards that was especially designed for this demonstrator. The board provides additional 8 MB SRAM, a compact flash card slot for storing the firmware and configuration data of partial modules, a video capture device, and connectors for the ReCoNet links and for the sensors or actuators. This hardware platform provides the same functionality as the original ReCoNodes but provides in addition the capability of partial runtime reconfiguration. The top right corner depicts the partition of the main FPGA into a static region and an area for hosting reconfigurable modules.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

This chapter is devoted to communication architectures required to design and implement the reconfigurable nodes of a ReCoNet. A special communication architecture is required whenever using partial runtime reconfiguration. It must provide all communication demands (e.g., bus connections or point-to-point links) between the static part of the system and the reconfigurable modules. The communication architecture has a large impact on the achievable performance in terms of throughput, the flexibility of the system, and the resource overhead. Note that sophisticated communication architectures may also be used for implementing a complete ReCoNet on a single device. This allows to apply the ReCoNet fault detection and repair methodologies for masking resource defects of future high density FPGAs.

The goal of this chapter is to propose and to analyze new on-chip communication architectures for FPGAs. This includes a novel bus-based architecture, called ReCoBus, for directly connecting runtime reconfigurable modules to a backplane bus and a further communication architecture, called Connection Bars, that can be used to provide point-to-point links to partially reconfigurable modules. The proposed architectures provide the following properties:

- **Direct interfacing** Reconfigurable modules can be directly hot-plugged into an on-chip backplane bus with the help of partial runtime reconfiguration. This includes integration of reconfigurable bus masters into the system. In addition, modules can connect to I/O pins, other on-chip resources or other reconfigurable modules within the system.

- **Module relocation** Instead of binding modules to a fixed reconfigurable island, they can be placed freely within the reconfigurable area. This area may be tiled into a one-dimensional slot style layout or a two-dimensional grid.

- **Variable module sizes** This allows integrating multiple modules with different resource requirements into the system without wasting large resource areas for smaller modules. This permits a system to replace a large module by multiple smaller ones or vice versa.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

- **Fine placement grid** The module placement grid is a quantity for the granularity of how fine a module bounding box can be adjusted to just fit the area demands of a particular module. In addition, the module grid defines how flexible these can be placed within a reconfigurable area. The module grid also defines the possible placement style of the modules within the reconfigurable resource area. The placement style ranges from an island style placement with fixed resource areas over one-dimensional slot based placement styles to free two-dimensional module placement techniques (see also Figure 3.1).

- **Multiple instances** This attribute denotes that multiple instances of the same module could be integrated into the system at the same time. Using multiple instances allows a system to scale more flexibly on changing load demands at runtime.

- **Low logic overhead** to implement the communication architecture.

- **High performance**: The communication bandwidth and the latency of the reconfigurable communication architecture can compete with traditional static only systems.

- **Hot-swapping** Modules can be replaced without interfering with the rest of the system. The configuration process itself can be managed completely transparent to running communication cycles on the communication architecture.

This chapter puts focus on communication architectures for reconfigurable FPGA-based systems that optimize internal fragmentation as well as the communication logic overhead. The impact on the overhead will be discussed in the following section, Section 3.1. For clarifying all geometrical aspects, an FPGA architecture will be introduced in Section 3.3. After this, in Section 3.4, related work on such communication architectures will be summarized. Next, in Section 3.5, the novel ReCoBus architecture for reconfigurable on-chip buses will be presented that is in particular tailored to FPGA-based systems. This communication architecture can be efficiently implemented in one-dimensional slot based systems as well as for systems following a two-dimensional grid placement style, as revealed in Section 3.5.6. Afterwards, in Section 3.6, architectures for point-to-point communication will be proposed. These architectures provide dedicated reconfigurable connections of partial modules with I/O pins or other modules within the system. The introduced architectures will be experimentally evaluated in Section 3.7. Finally, the contributions and the progress against related approaches will be summarized in Section 3.8.
3.1 Preconsiderations on Flexibility and Module Placement

When utilizing partial runtime reconfiguration in a system, a structural partitioning step for dividing the FPGA resources into a static and one or more reconfigurable regions is required. The partitioning massively affects the placement flexibility and results into a specific placement style as illustrated in Figure 3.1. As it is unlikely to manage the reconfigurable area in plenty of small look-up tables, it is common to group multiple look-up tables into one or more rectangular tiles. Such a tile is the smallest atomic area that can be assigned to a module, and, depending on the system, multiple adjacent tiles may be occupied by a module. In order to emphasize that these tiles provide the logic and routing resources for implementing some modules that are dynamically plugged into the system by partially reconfiguring an FPGA, such a tile will be named resource slot. As revealed in the figure, the resource slot grid may follow a) a simple island style, b) a one-dimensional tile style, or c) a two-dimensional grid style. The placement style is massively influencing the configuration overhead that consists of two factors: 1) the logic required to provide communication with other modules or the static part of the system and 2) the internal fragmentation that arises by fitting modules into resource slots. In this sense, a resource slot has some similarities with a sector on a hard disc drive. A sector is the smallest chunk of memory that is directly accessible and a file may occupy multiple sectors for storing its data, while one sector is not shared among multiple files.

In order to quantify the flexibility of different reconfigurable systems, the placement flexibility is defined. This value denotes the sum of all possible starting points to place a partially reconfigurable module on an FPGA-based system. This is regardless of the question whether a particular module could start in all the provided slots.

Figure 3.1: Different placement styles of reconfigurable modules. In all examples, the total reconfigurable area and the individual module sizes are the same. With smaller tiles, internal fragmentation can be minimized and the number of simultaneously placed modules maximized.
With respect to the example in Figure 3.1, the island style case provides a placement flexibility of 2 and the grid style case a flexibility of 20, respectively.

### 3.2 Efficiency Preconsiderations

When assuming that one resource slot provides $\sigma$ look-up tables, then, if omitting communication cost, half of the amount of resources available in one slot ($\sigma/2$) will be wasted in average per module just because of internal fragmentation. This is because in the best case, all resource slots will be completely filled and in the worst case, a module will leave almost the amount of logic provided in one resource slot unused. Consequently, the resource slots should be small. On the other side, the cost for implementing the communication with the reconfigurable modules that are located inside the resource slots may unlikely increase. For the following, it is assumed that the communication cost for the inter-module communication within each resource slot is $c$ look-up tables and that a resource slot provides altogether $\sigma$ LUTs. Then, all modules $m_i \in \mathcal{M}, \forall i = \{1, \ldots, |\mathcal{M}|\}$ possess the following average module overhead $O$ in terms of look-up tables:

$$O = \frac{1}{|\mathcal{M}|} \sum_{i=1}^{\mathcal{M}} \left( \left\lfloor \frac{|m_i|}{\sigma - c} \right\rfloor \cdot \sigma - |m_i| \right).$$

(3.1)

Where $|m_i|$ denotes the size of a module in terms of LUTs and where $\left\lfloor \frac{|m_i|}{\sigma - c} \right\rfloor$ states the amount of resource slots required to implement a reconfigurable module $m_i$. The average module overhead $O$ allows a first rough estimation of an optimal resource slot size as revealed in Figure 3.2. Here, a synthetic scenario is assumed with modules ranging from 300 to 10000 LUTs, which roughly corresponds to 10% to 300% of the logic required to implement an average 32-bit softcore CPU. In this example, the amount of modules is chosen in such a way that all modules sizes contribute with the same overall amount of look-up tables to the set of modules $\mathcal{M}$. At this point, a scenario is assumed with $|\mathcal{M}| = 9700$ different modules ranging from 300 to 10000 LUTs in increments of one LUT. As can be seen in Figure 3.2, the communication cost is dominating for smaller resource slots, while for larger slots, the internal fragmentation will dominate. The curve points out that the size of a resource slot should be around a few hundred LUTs. In the case that $|m_i| \gg \sigma$, the placement overhead converges to $\frac{\sigma}{\sigma - c}$. In other words, for large modules requiring many resource slots, the internal fragmentation becomes negligible and the communication cost $c$ will dominate.

For a given set of modules $\mathcal{M}$, the optimal resource slot size $\sigma_{opt}$ against the communication cost can be computed by differentiating Equation (3.1) with respect to the slot size $\sigma$. As the ceiling function is discontinuous, its bounds are considered:

$$\left\lfloor \frac{|m_i|}{\sigma - c} \right\rfloor \leq \frac{|m_i|}{\sigma - c} < \left\lfloor \frac{|m_i|}{\sigma - c} \right\rfloor + 1.$$
3.2 Efficiency Preconsiderations

Figure 3.2: The impact of the resource slot size $\sigma$ and the communication cost $c$ on the average module overhead $\bar{O}$ for a synthetic module distribution.

fits into the resource slot without internal fragmentation, the upper bound denotes the case of maximal fragmentation where one resource slot is almost unused. For the lower bound, internal fragmentation is optimized by reducing the resource slot size to $\sigma = 0$. However, in the case of a communication cost $c > 0$, this results in an explosion of the of the average overhead (see the results in Figure 3.2 for small resource slots). For the upper bound, the average fragmentation over all modules $m_i \in \mathcal{M}$ can be minimized as follows:

$$\bar{O} = \frac{1}{|\mathcal{M}|} \cdot \sum_{i=1}^{|\mathcal{M}|} \left( \left( \frac{|m_i|}{\sigma - c} + 1 \right) \cdot \sigma - |m_i| \right)$$

$$\frac{\partial \bar{O}(\sigma)}{\partial \sigma} = 0 \Rightarrow \sigma_{opt} = c \pm \sqrt{|\mathcal{M}| \cdot c \cdot \sum_{i=1}^{|\mathcal{M}|} |m_i|}. \quad (3.2)$$

The average overall resource slot size is then approximately the mean value between the optimal slot sizes, determined for the two bounds: $\sigma_{opt} \approx \sigma_{opt}/2$. Figure 3.3 plots the relationship between the communication cost $c$ and $\sigma_{opt}$. The results match the minima depicted in Figure 3.2.

The geometrical view on a concrete system allows to state whenever runtime reconfiguration is suitable or not. The geometrical view is equivalent to a view on the partitioning of the FPGA resources. Let $\tilde{\mathcal{M}} \subseteq \mathcal{M}$ be the module subset requiring the largest amount of resource slots in a reconfigurable system at a point of time, then the following inequation must be met for achieving an overall benefit when applying
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.3: Average optimal resource slot size $\sigma_{opt}$ for a given communication cost $c$ in each slot. The results are obtained for the same synthetic module distribution as assumed for the plot in Figure 3.2.

Here, $c_{const}$ summarizes all additional resources required for allowing the system to exchange modules by partial reconfiguration at runtime, for example, a reconfiguration interface controller. Inequation (3.3) is a necessary condition but not a sufficient one, because other important factors including reconfiguration time, external fragmentation, or design issues (e.g., tools) are omitted in this consideration. Figure 3.4 reveals an example where Inequation (3.3) is not met, because a static only implementation without using runtime reconfiguration would require less FPGA resources. In order to achieve an overall benefit by using runtime reconfiguration, the internal fragmentation could be reduced by 1) tiling the resource slots in a finer grid, by 2) reducing the communication overhead per resource slot, and by 3) optimizing $c_{const}$.

Inequation (3.3) and the example in Figure 3.4 points out that runtime reconfigurable systems have to be designed with ultra low resource overhead in order to justify this approach. As this efficiency seems not to have been demonstrated so far, runtime reconfiguration has not been introduced widely in industrial applications and is still very exotic, despite its potential benefits.
3.3 FPGA Architecture Model

Independent from the utilized FPGA architecture, FPGAs are based on 1.) functional units such as logic cells, I/O blocks or memories and on 2.) an interconnect network for wiring these functional units. The goal of this section is to define a generic FPGA architecture model that reflects all commonly used FPGA architectures with respect to the implementation and analysis of runtime reconfigurable systems. The resulting model has to provide a geometrical view rather than a logical view on the functional units and the interconnect resources of different types of FPGAs. This is essential for examining the additional steps required to enhance a static only system to a reconfigurable system. This includes floorplanning and tiling an FPGA into resource slots, as well as all communication issues.

In common, one or more functional units and the according switching elements of the interconnect network can be combined into one basic element (BE). Multiple BEs, each consisting of equivalent logic and routing resources, are placed on the FPGA in a highly regular fashion. On Xilinx FPGAs, a BE would be equivalent to a so-called configurable logic block (CLB) [Inc08b] that consists of multiple look-up tables (LUT) to implement logic functions. In the Xilinx Virtex-II architecture, for example, eight four-bit LUTs share one so-called switch matrix. The equivalent to a CLB on FPGAs from Altera Inc. is called a logic array block (LAB) [Inc08a]. The architecture of a BE may vary even between different FPGA architectures from the same vendor, but in general, the routing within a BE is extensively faster when compared to the routing among different BEs. In summary, all commercially available FPGAs follow an FPGA architecture model with regular tiled BEs that are also
3. **Intra-FPGA Communication Architectures for Reconfigurable Systems**

connected by regular structured wires, as shown in Figure 3.5\(^1\). Besides FPGAs, this model may also fit to other regular structured logic devices including CPLDs, PLDs, PLAs, or even structured ASICs. The geometrical parameters for partitioning the FPGA into a resource slot grid for hosting multiple reconfigurable modules are as follows:

- **W**: resource slot width in BEs
- **H**: resource slot height in BEs
- **R**: resource slots in total as the product of horizontal and vertical resource slots: 
  \[ R = R_H \cdot R_V \]
- **M**: maximum amount of simultaneously integrated modules
- **w**: module width in resource slots
- **h**: module height in resource slots

With respect to the example in Figure 3.5, module 1 has a width of \( w = 1 \), a height of \( h = 2 \), and the placement position \( S = (0, 3) \).

The BEs of a few currently available FPGA architectures are not identical over the complete device. For example, in Xilinx Virtex-V FPGAs, only every second column of logic blocks can be configured as distributed memory. Similar irregularities also apply for the Stratix-4 [Inc08a] architecture from Altera. However, by grouping a set of CLBs (or LABs, respectively) together, it is still possible to construct a completely homogeneous BE layout on such FPGAs that fits the model in Figure 3.5. For instance, it is possible to group in each case two horizontal adjacent CLBs of a Xilinx Virtex-V FPGA together into one BE that will then possess the same logic and routing resources throughout the FPGA. A further concern are dedicated resources like embedded memory blocks or hardwired multipliers. By examining the Xilinx Spartan-3, Virtex-II, Virtex-IV, and Virtex-V FPGA architectures, it was discovered that the dedicated resources provide exactly the same routing resources as available within all logic tiles. Hence, these tiles cannot provide module connections as compared to the CLBs, but the routing architecture allows to implement homogeneous structures throughout the FPGA.

Typically, FPGA architectures have routing resources that span different lengths. In Xilinx Virtex FPGAs, for example, there exist so-called *double lines*, spanning over a width of two CLBs and *hex lines* spanning over six CLBs, respectively. As revealed in Figure 3.6, some of these wires can be tapped by multiple CLBs which is typically utilized for efficiently implementing multicast signals, such as enable or reset signals. Each output of a basic element forms a *channel* that contains as much wires as its span \( S \). The different channels will be named by their span and their

\(^1\)Inhomogeneous elements, including distributed memory, dedicated memory blocks, or hardwired multipliers will be covered later in this section.
3.3 FPGA Architecture Model

Figure 3.5: Simplified FPGA architecture model. An FPGA consists of a two-dimensional regular grid of basic elements (BE) containing the switch and logic resources. Wires of different span (S) between the BEs carry out the distribution of signals. For each span, there exists in each BE a number of output wires (counted by i) per span forming a channel $C^S_i$. To simplify the figure, only some horizontal routing wires are illustrated, although they may be arranged in any desired horizontal, vertical, or diagonal fashion. The reconfigurable system will be built onto such architecture. A system offers $R_h$ horizontal and $R_v$ vertical resource slots that are each $W$ BEs wide and $H$ BEs high. Modules can have different sizes that are a multiple of $W$ BEs wide and a multiple of $H$ BEs high. The total number of resource slots is $R = R_H \cdot R_V$. The maximum number of reconfigurable modules that may be integrated into the system at the same time is $M$. In addition to the resource slots, the reconfigurable SoC will contain a static part to control the system. The static part can communicate with the modules via a special communication architecture.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.6: FPGA Editor print screen of a single configurable logic block (CLB) with highlighted hex (blue) and double lines (red) on a Xilinx Virtex-II device. Only outgoing wires are marked. On this FPGA, all CLBs provide exactly the same interconnect network with 10 hex and 10 double lines starting in both, horizontal and vertical directions. Each of these wires can be tapped in the middle after three or one CLBs, respectively. As depicted enlarged, a CLB provides four so-called slices, connected to one switch matrix on this FPGA architecture. Each slice contains two 4-bit look-up tables, some further logic, and two flip-flops.
3.4 Related Work on Communication for Reconfigurable Systems

index: $C_7^5$. (see Figure 3.5 that gives examples for triple and single wires). The direction of a channel is obvious by its usage and will be consequently omitted. If, for example, a channel is used for a link from the static part of the system towards the reconfigurable area, it must drive a signal easterly, if the system is floorplanned as shown in Figure 3.5.

The communication resources between the static part and the reconfigurable area have a preferential direction. This means that in the case of one dimensional, vertically aligned resource slots, the reconfigurable modules are located in a vertical manner beside the static part of the system. In such a system, the communication resources will be horizontally aligned and will cross all resource slots starting from the static part of the system. A resource slot may provide some logic and routing resources for the access to a bus and/or a point-to-point link. Such an access to the communication infrastructure is named a module connection point. The BEs of a module connection point must have the same relative position within all resource slots and must further contain the same internal logic functions. This allows connecting different modules to the communication architecture at the same position over time without interrupting the communication of other reconfigurable modules that may be carried out over the currently reconfigured resources. Among multiple different modules, the bus and point-to-point interface signals are allocated to fixed logic and routing resources within a resource slot.

All following figures that show some details of this communication architecture will be arranged in such a way that the static part is located at the left hand side of the system, while the resource slots will be horizontally aligned. This scheme matches best to the FPGAs offered by Xilinx Inc. that feature a column-based configuration scheme where the smallest piece of configuration data is a complete column spanning over the complete height of the device\(^2\). However, the presented techniques can be transferred easily to build reconfigurable on-chip buses towards any direction. At the end of the two subchapters, Section 3.5 and Section 3.6, it will be described how the communication architecture for one-dimensional reconfigurable systems can be extended towards more flexible two dimensional systems.

3.4 Related Work on Communication for Reconfigurable Systems

All approaches for on-chip communication architectures can be classified into the following three classes: 1) circuit switching techniques (Section 3.4.1), 2) packet

\(^2\)The newer FPGA architectures Virtex-IV, Virtex-V and Virtex-VI from Xilinx Inc. are configured sub-column wise. These devices possess still a preferable column wise module layout, because of restrictions in the local interconnect between the logic elements (e.g., interconnects for fast carry-chain logic is only available between vertically aligned logic elements).
Figure 3.7: Taxonomy of on-chip communication architectures (from [MSCL06]).

From a system level perspective, the ReCoBus communication architecture is a hierarchical shared bus, because it connects the bus segment of the reconfigurable subsystem with the backplane bus of the static system. With respect to the reconfigurable subsystem, it is a fully homogeneous structure with identical interfaces to all reconfigurable modules. The Connection Bar architecture is a custom point-to-point interconnect network, because it allows to set connections when there is a necessity. This is sometimes also referred as ad-hoc interconnect.

switching mechanisms (Section 3.4.2), and 3) on-chip buses (Section 3.4.3). Circuit switching is often referred to point-to-point links. Such a classification into different structures and protocols has been refined by Mak et al. [MSCL06] as revealed in Figure 3.7 mainly for implementing static only systems\(^3\).

With respect to communication architectures for runtime reconfigurable systems, it is common to link the modules via a uniform, homogeneous interface that is equivalent in all resource slots for allowing module relocation. For the same reason, hierarchical buses have not been proposed for runtime reconfigurable systems, despite of the fact that such buses are often used in static only systems. A segmentation of routing wires is not only bound to buses (see Figure 3.7) and has been demonstrated for circuit switched networks, as well [WJ06, ABD+05]. In the following, the different communication architecture classes and associated protocols will be separately discussed in more detail. After this, an overview on physical implementation techniques will be outlined.

### 3.4.1 Circuit Switching

Circuit switching is a technique where physical links are established between two (what is also known as point-to-point communication) or more module terminals for a certain amount of time. To be more concise, after establishing a channel of fixed bandwidth between a source and one or more destinations, communication can take

\(^3\)The classification is primary based on physical aspects. With respect to the OSI Reference Model, higher layers, e.g., the data link layer, are not considered at this point.
3.4 Related Work on Communication for Reconfigurable Systems

**Figure 3.8:** a) Reconfigurable Modular Bus (RMB) example. The first established link is channel 1 from module 1 to module 3. After this, channel 2 was established from module 2 to module 4. During link construction, the state machine in each crosspoint (starting from source to destination) will select the most top wire segment that is currently available. b) The crosspoints feature a full crossbar that is implemented with multiplexers.

place with a fixed delay (as opposed to packet switching), and the channel is locked until all modules agree to release the channel. Circuit switching is advantageous if the channels are used at high bandwidth, because almost all wires of the channel may carry data, thus allowing a good interconnect utilization of the routing resources. This is typically the case for video or network processing systems. However, the interconnect utilization may be enhanced by time multiplexing different channels or by serializing the individual channel signals [BT93, DT01, MH04].

Typically, the module terminals are implemented by more or less complex crossbar switches. The switching state of the internal multiplexers can be controlled by the static system in a centralized manner [UHB04] or distributed by some logic in the crossbar switches itself [ESS+96, ABD+05].

**Reconfigurable Modular Bus (RMB)** In [ABD+05] a circuit switching architecture is presented where the switching state is determined by the crossbar switches (called crosspoints) with respect to the currently used routing resources, as illustrated in Figure 3.8. This process is completely carried out in hardware and does not require additional software assistance. Setting up a channel starts from the source towards one destination. In each channel, the currently topmost available wire segment is chosen, until the destination crosspoint is reached. With this policy, a valid route is always being found if there are sufficient wire segments between the crosspoints available. The approach requires fully equipped crossbars, hence, it is unlikely scalable with the number of wire segments (quadratic), the bit width (about linear, as shown in Table 3.2), and the number of crosspoints (linear). However,
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

In [FvdVMT06, FvdVA+07] ILP based solutions are presented for finding module positions to minimize the wire segments between the crosspoints, as well as shortening the longest path and to reduce the total amount of required wire segments in a system. With respect to the example in Figure 3.8, all of these objectives will be enhanced by swapping module 2 with module 3.

Time Multiplexing In [MH04], a system allowing time multiplexed circuit switching is presented. Here, multiple channels are periodically switched in a time triggered manner. This is implemented by a modulo counter (located in the static system) for incrementing the connection state. The crossbar switches are set according to the entire counter state. The assignment of the switch positions to the connections states can be changed, thus, allowing to adjust the schedule of the communication architecture. The authors claim this structure being a Network-on-Chip, but the switching elements are still controlled centralized without performing any packet processing. As a consequence, this approach is classified as a circuit switching technique.

Virtual Wires Originally, the virtual wire [BT93, BTD+97] concept was proposed for systems containing a large design that is partitioned into multiple FPGAs. This is highly relevant for building FPGA-based logic emulators, as the amount of logic on one FPGA scales quadratically with the length of the die (assuming a quadratic chip) and also quadratically with enhancing the technology (transistor shrinking), while the number of I/O pins scales only linear with the die length and even less than linear with the process technology. Consequently, there is a much lower I/O bandwidth per amount of logic resources available on larger FPGAs. When the amount of I/O pins between the FPGAs is insufficient for linking all signals, some or all signals are serialized at the source FPGA and transmitted to the destination FPGA where the signals are deserialized again. Therefore, a much lower amount of I/O pins is required to transfer some signals between the partitions located on different FPGAs.

Beside logic emulation, the authors propose to also use virtual wires for integrating reconfigurable modules into a system at runtime. This is an interesting option to be combined with circuit switching as the crossbars can be implemented with much less logic, as the amount of logic resources for the multiplexers scales linear with the bit width of the switched data channels. However, this is only applicable in situations, where the throughput allows a serialization of the data.

Two-Dimensional Circuit Switching Hübner, et al. proposed in [HSKB06] a two-dimensional circuit switching system that is based on templates for connecting, turning, forking, crossing, or routing through a set of signals. By attaching these templates together through partial reconfiguration, it is possible to set a fixed routing path between some modules at runtime. Consequently, communication with mod-
3.4 Related Work on Communication for Reconfigurable Systems

Figure 3.9: Circuit switching network proposed by Hübner, et al. in [BHB+07]. Multiple modules can be integrated in a daisy chained fashion within horizontal slots. The system includes multiple one-dimensional slots that can be linked in any order with the help of a crossbar switch.

Wires on Demand Athanas et al. proposed to freely route signals between some reconfigurable modules. They call this concept wires on demand [ABD+07] and it is based on reserving routing channels exclusively for performing fine grained online routing. In [SAP08] the authors demonstrate that this routing can be performed with low memory requirements (96 KB for the routing resource database) within an embedded system. The approach cannot guarantee to fully route all signals for any placement scenario at runtime. Furthermore, modules are typically integrated incrementally into a system, hence, any rerouting of an existing routing path will lead to an interruption. The routing channels are not shared with the modules but constitute obstacles, and are thus restricting the module placement.
3.4.2 Networks on a Chip

The motivation for packet switching for on-chip communication comes from the ASIC domain where more and more functional units are integrated. In order to deal with multiple clock domains, modularity for IP-reuse, and to support parallelism in communication and computation, Networks on a Chip (NoC) [BM02] have been proposed. In [DT01], Dally and Towles suggested packet routing for ASIC design. This is justified by the authors because in ASICs some key portions of a chip are often limited by the wire density and another observation that an average wire is used (toggling) less than 10% of the time. By spending about 6.6% more in chip area in an example implementation, the wire utilization could be materially enhanced without increasing latency because critical communication was pre-scheduled.

NoCs in FPGA-based Reconfigurable Systems  An early work on implementing a NoC on an FPGA was introduced in [MBV+02]. The network implements a two-dimensional torus topology and uses wormhole routing. The authors report a throughput of 77.6 MB/s on 16-bit wide data channels between adjacent routers when the system is clocked at 40 MHz. The reported synthesis result for a $2\times2$ torus is 3227 slices which is 9.5% of an Xilinx Virtex-II-6000 FPGA. The system was implemented by folding the two-dimensional torus in a one-dimensional slot based structure with the help of tristate-based bus macros.

Another FPGA-based NoC implementation [SBKV05] consisting of a $3\times3$ mesh, requires 3934 slices on a Xilinx Virtex-2Pro 30 FPGA which is 28.7% of the total logic resources. The achieved clock frequency was 32 MHz and the width of the channels was chosen to be 8 bit. The routers employ an XY-routing strategy with store and forward routing. The target router is accessed by its coordinate. With the proposed 8 bit wide address field, the network is limited to a maximum of $4\times4$ routers. A router has four ports, connecting adjacent routers, plus a fifth port for connecting a module to the network. All ports can simultaneously receive and send data.

Another very similar NoC for FPGAs was previously presented in [ZKS04]. In that work, instead of a complete NoC, a single router that occupies 570 to 1830 LUTs on an Altera Flex 10K FPGA, depending on the data channel width (8 to 32 bit), has been implemented.

To achieve larger networks and to scale down the grid without a dramatic logic overhead, Bobda et al. [BMK04, ABM+05] propose to detach routers from the network for alternatively implementing the module logic in the resources of the detached routers. As depicted in Figure 3.10, the routers are arranged in a mesh structure and larger modules may replace routers by reconfiguring the FPGA. By removing routers from the network and replacing them with module logic, modules become obstacles. The XY routing protocol is capable to deal with obstacles by routing packets around them. As this will increase the distance to the destination, the packets are temporar-
3.4 Related Work on Communication for Reconfigurable Systems

Figure 3.10: Example of a DyNoC. Modules can be implemented in one or more consecutive processing elements (PE). With the help of partial runtime reconfiguration, routers inside a module can be detached from the network and their logic resources will be used for implementing the module. After terminating a module, the routers can be included again. In a DyNoC, modules must be placed such that it is always surrounded with routers at all four edges.

ily stamped for forcing the routers not to send the packet back (which would reduce the distance to the destination) but to route the packet around the obstacle. However, this XY obstacle-routing requires that modules are always surrounded by routers at all four edges, which implies that there is a ring of routers around the reconfigurable area.

The largest implemented DyNoC contains a $4 \times 4$ mesh [BMK04]. A single router requires 819 (1229/2150) look-up tables for implementing routers with 8 (16/32) bit data path width and the maximum clock frequency is reported with 75 MHz [ABM+05].

The listed NoC implementations point out that these network structures are not appropriate for FPGA implementations. The logic overhead is enormous while the achieved data rates are relatively low. Furthermore, the implementations are very restricted and the here mentioned NoCs cannot synchronize between multiple clock domains. In a nutshell, packet-switching is not suitable for integrating partial modules into a reconfigurable system. However, packet-switching may be directly included into the architecture of future FPGAs for providing backbone connectivity between multiple larger logic islands.

99
Packet-Switching versus Time-Multiplexing on FPGAs

Both, packet switching and time multiplexing aim at enhancing the utilization of interconnect resources. While packet switching allocates a bandwidth to modules with respect to the current demands of the modules, time multiplexing is based on schedules for virtualizing interconnect links. In [KMD+06], the authors examined tradeoffs (area, throughput and latency) between time-multiplexed and packet-switched networks. For time-multiplexed networks, schedules have been computed offline with two different algorithms: a greedy heuristic and Pathfinder [ME95]. The greedy heuristic allocates the shortest paths and the earliest possible time slots for a piece of data first, while avoiding congestion. Pathfinder is an iterative algorithm where in each iteration the shortest path is computed for all routes independently. For example, by using Dijkstra’s algorithm [Dij59] or an A* variant [Tes98]. After each iteration, Pathfinder adapts the cost function with respect to the usage of a particular wire segment and to the available slack of the nets passing a particular wire segment. Consequently, Pathfinder minimizes congestion and latency.

The packet switching network contains simple routers that use a round robin policy for selecting between the input queues of a router. The time-multiplexing is implemented with a simple state machine, but requires some extra storage for the scheduling table. For providing a fair comparison, the authors of [KMD+06] optimized design parameters, such as buffer size or channel widths according to a synthetic benchmark. It was found that packet-switching requires less logic than time-multiplexing with increasing traffic, because of the scheduling tables will become unlikely large.

3.4.3 Buses

Buses are the native way to link communicating modules together within a system-on-a-chip (SoC). All major FPGA vendors offer tools that allow easily integrating a set of user-defined modules or IP cores into complete systems by the use of on-chip buses. For example, Xilinx offers the Embedded Development Kit (EDK) that allows to compose systems based on the CoreConnect bus [Int99] and Altera respectively provides the System on a Programmable Chip Builder (SOPC) for designing systems based on the AVALON bus [Alt07]. Apart from these two commercial bus specifications, the WISHBONE specification [Sil02] is public domain, issued by Silicore. For this bus standard, SoC design tools have been developed by academia [KKH06] and as open source projects (for example the WISHBONE Builder [ope]). Because of the prevalence of buses in SoC design, buses are good candidates for also integrating partially reconfigurable modules into a system at runtime.

The following paragraphs will first of all compile an overview about on-chip buses for runtime reconfigurable systems that are physically implemented with tristate drivers. After this, related work on buses that physically link reconfigurable modules with logic resources will be introduced.
3.4 Related Work on Communication for Reconfigurable Systems

On-chip Tristate Buses for FPGAs  Plenty of papers [WP04, KJdlTR05, KPR04b, BMG06, PdMM+02] that were published in this field are based on older Xilinx Virtex FPGA architectures that provide wires, spanning over the complete horizontal width of the devices and that can be used to build buses with tristate drivers (see also the paragraph about Xilinx bus macros on page 109). However, newer FPGA architectures have no support for tristate wires and none of these approaches deal efficiently with dedicated signals, e.g., interrupt lines and chip select signals.

In [WP04], [KD06b], and [KD06a], systems are proposed where fixed resource areas with also fixed connection points to a bus interface are used for the integration of partially reconfigurable modules into a runtime system. With these solutions, all modules have to fit into a resource area of fixed size and one resource area cannot be directly shared by multiple modules, even if the logic of multiple modules would fit into the same resource area. This will typically result in an enormous overhead due to internal fragmentation, as demonstrated in the introduction to this chapter. A suitable bus infrastructure for integrating reconfigurable modules should be able to connect modules of different sizes efficiently with the rest of the system.

A more flexible approach for integrating partially reconfigurable modules into a system at runtime has been published before by Wirthlin and Hutchings. In the dynamic instruction set computer (DISC) [WH95, WH96], modules spanning over the full width of the FPGA (a CLAy31 device from National Semiconductor) can be integrated at runtime with the help of a vertical aligned on-FPGA tristate backplane bus. Different resource requirements of the modules are fulfilled by adjusting their height. The module positions can be varied in a one-dimensional fashion.

Hagemeyer et al. [HKP06] propose a similar tristate driver based bus architecture for Xilinx Virtex-II FPGAs. In that system, module sizes may vary in terms of one-dimensional aligned and adjacent resource slots. Another contribution presented in [HKP06] are four different approaches for distributing dedicated write signals, such as enable signals to reconfigurable modules that are connected to the bus. The approaches are illustrated in Figure 3.11 and include the following techniques:

a) The position-based dedicated signal distribution routes the enable signals in a non-uniformed manner within the resource slots. Consequently, this approach prevents module relocation and does not allow multiple instantiations of the same module.

b) The module-based dedicated signal distribution uses a uniform horizontal routing for the enable signals within the resource slots. The entire vertical routing to the enable terminal of the tristate driver is part of the module routing. This allows module relocation but still prevents multiple instances of the same module.

c) The slot-based dedicated signal distribution uses a so-called adaptor for connecting the tristate enable terminal in vertical direction. The adaptor bridges
the different lengths in vertical direction between a fixed connection to the trisate driver and one of the uniform routed internal enable signals. By adjusting the adaptor, multiple instances of the same module can be integrated into the system. In [HKP06], the authors present an approach to use a dedicated logic column (a CLB column in Xilinx FPGAs) for implementing the adaptor which results in an enormous resource overhead for this variant.

The last approach in [HKP06] uses a serial *shift register* with parallel load that is distributed over all resource slots with one bit in each slot. The enable signal to a specific slot is distributed by shifting in a one hot encoded enable stream combined with a following load enable for the output flip-flops which store the tristate driver control signals. This technique provides the same flexibility as variant c) and entails a reasonable implementation cost but requires $R + 1$ clock cycles for setting select signals in $R$ resource slots. Furthermore, the approach requires some additional logic within the static part of the system for connecting the shift register interface.

With the above listed approaches, a tradeoff between placement flexibility or overhead in terms of logic or latency is provided. The same group published an improved dedicated write signal distribution scheme in [HKKP07b] with reasonable overhead and low latency that allows free module relocation, as well as multiple instances of the same module. As depicted in Figure 3.12a), comparators that are identically

---

**Figure 3.11:** Different techniques for distributing a select signal to reconfigurable modules (from [HKP06]). a) Position-based dedicated enable signals, b) Module-based dedicated signals c) Slot-dedicated enable signals with adaptor, and d) Enabling/Disabling via shift registers.
3.4 Related Work on Communication for Reconfigurable Systems

Figure 3.12: Improved dedicated signal linking from [HKKP07b]. a) Module enabling generation by a comparator that is decoding a binary encoded slot address. b) Module request signal linking (e.g., a module interrupt) by decoding a one-hot encoded slot address.

placed and routed within the resource slots are used for decoding a slot-dependent address that is stored in some extra flip-flops. This approach exploits a feature of Xilinx Virtex FPGAs that allows to set the initial values of the flip-flops during the initial configuration of the device. Further partial reconfiguration steps will not touch this initial flip-flop state that represents the particular slot addresses. Beside enable signal distribution, [HKKP07b] reveals how individual slot addresses can be used to send request signals via dedicated signal wires from a module to the static part of the system. This is achieved while maintaining relocatability and the capability of multiple module instantiation as depicted in Figure 3.12b). The disadvantage of these techniques is that the logic overhead scales with the amount of resource slots which is reasonable for the proposed system with 16 resource slots but unlikely if slots sizes are reduced in order to minimize internal fragmentation. Following the considerations on internal fragmentation in Section 3.2, the resource slot size of the proposed system should be about an order of magnitude smaller. Consequently, the amount of resource slots will rise by an order of magnitude, and therefore, prevents the depicted approaches in Figure 3.12 that is based on fixed slot addresses.

For multimedia processing systems, Sedcole et al. designed the Sonic-on-a-Chip communication architecture [SCCL04] that consists of a global bus similar to the one proposed by Hagemeyer [HKP06]. In addition, [SCCL04] reveals a chained bus for connecting adjacent modules. This is also implemented with tristate wires and is possible as these wires are segmented on Xilinx Virtex FPGAs.

On-chip Logic-based Buses for FPGAs Tristate wires require multiple drivers to a single wire, hence, leading to a poor chip utilization if such wires are used for static routing. Static means that the drivers to a tristate wire are not changed. The utilization is poor, because each driver requires some area on the die, but only one
driver can be active at a certain point of time. Using unidirectional wires instead would allow multiple independent active wires at the same cost [LLTY04]. Another problem of tristate wires spanning long on-chip distances is their propagation delay which enormously reduces throughput. With respect to the chip area and latency, it is more advantageous to split a wire in segments with small drivers for each segment, instead of using a single large driver [MSCL08]. Finally, tristate wires possess multiple drivers which may produce short circuits that potentially damage the device or may introduce further side effects including supply voltage drops, loose of signal integrity and an increased power consumption.

All these reasons have led to avoiding on-chip tristate wires in ASICs and consequently in FPGAs. Accordingly, all common SoC buses are implemented with logic based multiplexers. Such buses are often implemented hierarchically for reducing the size of the multiplexers and consequently enhancing speed for high performance systems. For example, in a typical system that is communicating via an AMBA bus [Ltd07], a high speed AHB bus (Advanced High-Performance Bus) may directly connect a CPU with the memory while communicating through a bridge with some peripherals connected via a slower APB bus (Advanced Peripheral Bus).

When designing buses for reconfigurable systems, hierarchical buses are unsuitable for connecting multiple modules within one reconfigurable area when these modules should be freely placeable. For such systems, the bus interface has to be uniformed and the challenge in designing on-FPGA buses for reconfigurable systems is to build buses providing low latency at low implementation cost that can integrate modules placed at many different positions.

A straightforward method of connecting modules with the help of logic resources to a bus is to simply bridge the signals from a configuration island into the static part of the system that contains all bus logic. This is implemented by Koh and Diessel with the COMMA architecture [KD06b, KD06a] that provides multiple reconfigurable islands, located in a two-dimensional fashion. By using look-up table bus macros, simple point-to-point bridges are implemented for linking partial modules with the bus logic. This logic is completely located within the static part of the system. The system was implemented on a Xilinx Virtex-IV FPGA that can be configured sub-column wise instead of configuring the device over the full height. By choosing the height of an island to exactly one sub-column, modules can be exchanged without influencing other parts of the system. The bus macro bridges deploy two look-up tables per signal wire that include no additional functionality, thus, resulting in pure logic and latency overhead.

A more advanced communication architecture was developed by Hagemeyer et al. [HKKP07b] as an alternative for tristate buses. In that approach, parts of the bus logic are joined together with the look-up tables used for connecting the bus with the modules. More details on this work are depicted in Figure 3.36.

**Serial Buses for FPGAs** Some systems do not require high speed bus communi-
3.4 Related Work on Communication for Reconfigurable Systems

For example, a FIR filter module may require some register file accesses for initializing and parameterizing the filter, while the entire data for computation may be transferred via dedicated resources for the complete execution time of the module. Then a serial bus may be appropriate to connect the control CPU with the filter module. Such a serial bus for reconfigurable systems is presented in [KKT06] for a cryptography application. In that system, a master sends an address followed by a command and optionally the data to the slaves that are daisy chained. In other words, the address, the control and the data signals of a master are serialized and transmitted to the slaves. In each slave module, the address value is decremented and sent further to the succeeding modules. If the incoming slot address is equal to zero, the module is selected and the command is executed. Commands can include reading or writing a particular register as well as some simple module control signals, such as a selective module reset. Except for the physical voltage levels, the serial interfaces in [KKT06] follow the RS-232 specification. Including a register file, a UART, a control FSM and the address computation, one module interface is reported to cost 830 look-up tables. It must be mentioned that this interface features some fault tolerance against temporal interruptions of the chain as well as against bit errors, because the proposed serial bus was used to link partial modules together that may be spread over multiple FPGA boards. A test implementation demonstrated, that the interface logic can be scaled down to less than 300 look-up tables when omitting fault tolerance. Each module provides one RX-TX (receive and transmit) pair for the upstream from a master to the slave modules as well as a second pair for the downstream link. Therefore, only four wires are sufficient for connecting partial modules via this serial bus with the rest of the system. In [KKT06], modified look-up table-based macros - located at the left and right border of a module - have been used for building a system featuring a chained reconfigurable bus along adjacent modules.

3.4.4 Physical Implementation Techniques

The communication architecture mainly restricts the placement style (see Figure 3.1). While an island style reconfigurable system may be implemented by locating all communication logic within the static part of the system [WP04, KV06, KD06a, LBM+06], more complex systems require a corresponding communication architecture, which may include large portions of the logic transparently within the reconfigurable modules [HKKP07b, KPR04b]. So far, suitable architectures for efficiently supporting a grid style module placement have been rarely proposed (e.g., [OM07, BHB+07]). The challenge in designing runtime reconfigurable systems featuring a two-dimensional grid style is to include variably placeable modules that can be freely scaled in height and width.

Partial runtime reconfiguration implies that modules are integrated into a system at runtime. This means that the module interface signals - for example all signals of a VHDL entity description of a module - have to be connected to the rest of the
system. Hence, partial runtime reconfiguration demands a strict separation of the module computation from module communication. This allows handling the module functionality as a black box while only requiring to adjust the communication for a particular reconfigurable module at runtime after or during reconfiguration. The modules themselves as well as the static system are physically stored as completely routed components in bitstream repositories, as it is virtually impossible to perform full place and route in an embedded system at runtime. As a consequence, physical implementation techniques for providing low level connections are necessary.

For providing links among the interfaces of multiple reconfigurable modules or between the static part of this system, modules must be designed with corresponding connection points acting as fixed transfer points for the module interface signals. Basically, this could be achieved by simply binding specific signals that have to cross the module boundary to dedicated wire segments of the FPGA fabric. Depending on the design tools, such wire segments are typical not directly usable at higher abstraction levels. To overcome this issue, links via I/O pins [AS93, LNTT01, MTAB07] for connecting reconfigurable modules have been proposed, as well as so-called hard macros [LBM+06] that are build upon additional FPGA functional units. The connection points must be bound to predefined resources on the FPGA, that have an equivalent placement in the static system as well as inside the partial modules. Equivalent means that specific signals are located at the same fixed relative locations inside all resource slot, as illustrated in the examples of Figure 3.13.

**Figure 3.13:** Four resource slots assigned logically on top of the logic and routing fabric of an FPGA. Communication is either established via external I/O tiles or via dedicated wire segments of the FPGA fabric. Depending on the tools, the latter one may be implemented directly or with the help of additional logic tiles acting as connection points.
Partial Module Linking via I/O Pins

Using I/O pins is a simple, straightforward method for providing communication to and from reconfigurable modules. This was used in pioneering approaches for exchanging full FPGA configurations. In such systems, a resource slot engages a complete device and partial reconfiguration is achieved on system level by exchanging some full FPGA configurations while keeping other devices operating. However, the progress in silicon industry results in the ability to implement full systems on a reconfigurable chip (SoRC). Hence, partial reconfiguration will be defined as exchanging fractions of the configuration of a single FPGA throughout the rest of this thesis. An early work using runtime reconfiguration was proposed by Athanas and Silverman in the PRISM project [AS93]. PRISM is an acronym for “processor reconfiguration through instruction set metamorphosis”. The proposed system comprises some Xilinx 3090 FPGAs for extending the instruction set of a M68010 processor. In this system, the FPGAs are fully reconfigured for adaptations which was called instruction set metamorphosis. This concept was retained in a further development of PRISM-II [WAL+93], which features Xilinx 4010 FPGAs. In these systems, the coupling with the static part (here the CPU) was completely carried out via the I/O pins of the FPGAs that are individually but only fully reconfigured. The FPGA I/O pins are directly connected to the PCB backplane bus.

Lockwood et al. proposed a system called field programmable port extender (FPX) [LNTT01] for network packet processing with partially exchangeable accelerator modules. In this system, all interface signals of a partial module are routed to I/O pins, located inside the module bounding box. And most of these pins are connected to external memory buses. These buses are designed for high throughput with a width of 64 (SDRAM) and 36 (SRAM) bits. Consequently, a single partial module requires a material amount of external pins, thus, making this approach very costly and unlikely for scaling the number of simultaneous partial modules inside the system. The approach follows an inflexible island style placement of the modules, hence, the external I/O pins are exclusively allocated to one particular module.

Linking partial modules via I/O pins has also been proposed in a current project called the Erlangen Slot Machine (ESM) [MTAB07], as just one possibility of providing physical links to reconfigurable modules. Beside other techniques (listed later throughout this section), the ESM can directly connect modules to external peripherals (for example, a video DAC or an AC97 audio chip) with the help of an external crossbar chip. This allows a fine-grained, and therefore flexible placement grid, while still providing access to the external peripherals. These advantages are achieved by the cost of an extra crossbar device that causes additional latency and is consequently limiting throughput. A further restriction is that the module size is related to the maximum of the resource requirements for implementing the logic and for providing
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

the external I/O pin interface. The ESM follows a one-dimensional placement style with columns spanning the complete height of the device. Consequently, the size of a reconfigurable module is directly related to its width; and the used main FPGA of the ESM (XC2V6000) provides only three I/O pins at the bottom side of the device per logic column. Whereas one column contains 768 4-bit look-up tables on this device. These three bottom I/O pins are linked to the crossbar chip. In other words, the I/O-pin density is three pins per 768 LUTs that is 12 pins for an average 32 bit softcore CPU with about 3000 LUTs.

In the ESM, I/O pins are not only used to link peripherals to the reconfigurable modules but also for directly connecting dedicated external SRAM memory banks. The ESM platform provides 6 memory banks that are connected to the top I/O cells located above a logic column. A distinctive feature of this scheme is that modules may be relocated to different positions (start columns) while still being capable to access a private external memory bank.

Partial Module Linking via Tristate Buffers

While modern FPGA architectures are no longer supporting internal tristate drivers\(^4\), this was common in many older FPGA architectures. Such tristate signals have been mainly used for implementing wide input functions, such as wired-OR or wired-AND gates, as well as for global communication purposes.

A pioneering project using internal tristate buffers for integrating partially reconfigurable modules in a single FPGA is the dynamic instruction set computer (DISC) [WH95, WH96] proposed by Wirthlin and Huhtings. Based on two National Semiconductor CLAy31 FPGAs - one for control and one for actually accelerating computations - the system is able to partially exchange simple instructions with special configuration commands (as part of the instruction set). The accelerators span the full width of the device for connecting tristate wires distend over the full height of the device. The modules height is adjustable to the entire logic demands. As the tristate drivers are located in a fine vertical grid, modules can be placed freely into the system following a vertical one-dimensional style, while still being able to communicate with the rest of the system. Consequently, the system features module relocation. Furthermore, the device could accommodate multiple reconfigurable modules at the same time. In this approach, the tristate drivers are actually used as connection points for linking the signals of the hardware accelerators to specific long lines.

\(^4\)The last FPGAs with internal tristate drivers to internal wires are the Xilinx Virtex-II and Virtex-II Pro FPGAs that have been released in 2000 and 2002 respectively for the II Pro series. Tristate signals for partially reconfigurable systems come along with additional place and route restrictions, as typical FPGAs provide only very few of such routing resources, and require timing parameters that must be met to turn buffers on-and-off. This leads typically to lower clock speeds as compared to logic based approaches.
Xilinx Bus Macros  Most projects using tristate drivers for implementing a module interface to partially reconfigurable systems on Xilinx FPGAs are based on so-called Bus Macros [Inc02]. These macros define the positions of a set of tristate drivers acting as connection points to predefined horizontal long lines, hence, following the same idea of the vertically aligned DISC approach. The reason for using horizontal long lines stems from the fact that Xilinx FPGAs follow a column based reconfiguration style. This means that configurations cannot be written randomly to the device but in larger portions of so-called vertically frames that span the complete height of the device. Ergo, partial modules for Xilinx FPGAs should have thin and tall module shape.

For Xilinx FPGAs that provide internal tristate drivers and according wires, the propagation delay on these resources is higher as compared to other routing resources. On the other hand, these routing resources can be used independently from the look-up tables that implement the logic of a module. As a result, linking partially reconfigurable modules with tristate drivers on Xilinx FPGAs can be implemented with a very low logic resource overhead.

Walder and Platzner proposed a reconfigurable runtime environment [WP04] using tristate driver based macros for Xilinx FPGAs that allows including multiple modules of different size in a one-dimensional slot based fashion into the system. The reconfigurable modules access the bus via a bus access controller for encapsulating the module interface to the bus arbiter. Consequently, the data transfer protocol is transparent to the modules and permits concurrent bus transactions. In this approach, modules are not allowed to have direct connections to some I/O pins or to other on chip modules inside the FPGA.

In [KPR04b] Kalte et al., a further approach of a one-dimensional tristate driver based bus is presented for the RAPTOR2000 [KPR02] prototyping platform. Here, focus is put on tightening the resource slot width in order to reduce the effects of internal fragmentation. In addition, the authors present how the tristate wires can be used for implementing segmented buses. Therefore, multiple bus transactions on the same horizontal wire can be performed. This is possible because the used Xilinx FPGAs provide configurable switches to link tristate buffer lines to longer wires. A major drawback of this approach is that so-called bridge modules must be placed into the system just for activating the switches between two consecutive wire segments.

In this system, the authors propose linking partial modules with the help of tristate drivers not only for bus signals but also for I/O pins. Consequently, partial modules can be relocated to different positions inside the FPGA while capable of directly accessing external peripheral components.

Palma et al. used tristate drivers, called virtual pins [PdMM+02] to implement sockets that are regularly aligned on a Xilinx FPGA. In this approach, the modules are
not spanning the full device height and the static system is occupying resources below the resource slots. This static area hosts the bus logic and provides links to the tristate drivers that act as connection points for linking the modules. In other words, the virtual pins concept splits the resource columns of a Xilinx FPGA into a static fraction and a reconfigurable fraction. This requires merging bitstreams of the static system with the partial modules, because the smallest atomic piece of configuration data accessible in a Xilinx FPGA (here a XCV300) is a so-called frame that includes all resources within the complete height of the device.

The issue was solved with the tool core unifier that merges a master bitstream (static design) with the module bitstream at runtime. The bitstream merging technique has some advantages when compared to [KPR02] and [WP04], where partial modules are required to span over the full height of the FPGA. The merging technique allows heterogeneously routed bus signals, such as chip select or interrupt signals, as well as using the I/O pins located above or below the resource slots that host the reconfigurable modules.

Further approaches Because of a lack of available and easily usable tools, multiple other systems have been developed that are basically variants following the techniques of the three above mentioned papers. For example, Bieser and Müller-Glaser [BMG06] presented an FPGA-based rapid prototyping system containing a simple lightweight tristate driver based system bus for directly linking modules in a slot based manner. Krasteva et al. [KJdlTR05] reintroduced the virtual wire concept and implemented a tool called BITPOS [KdlTRJ06] for merging partial module configurations into the static configuration bitsteam. In [SGS05], Sunwoo et al. propose a reconfigurable system based on the Atmel AT94K FPGA architecture for providing RAM connections for built-in self-testing (BIST).

Partial Module Linking via Logic Resources

As an alternative to the tristate driver approach, it is possible to build hard macros from the look-up tables (look-up table bus macro) that can also be instantiated within some high level design tools (e.g., the Xilinx design tools). By instantiating two LUTs, one at the beginning and one at the end of a wire resource, it is possible to bind a particular signal to a specific wire. The macros are designed and placed such that one LUT is placed inside the reconfigurable module bounding box, and the other one inside the static part of the system or another module. As a consequence, the LUTs act as connection points to the wire inside the macro that passes the boundary of a partial module in a predefined manner.

Xilinx LUT-based Bus Macros As compared to the tristate driver approach, the look-up table bus macros allow a higher density of signals and a lower latency by the cost of two look-up tables just for implementing the communication between
the static system and a module. Look-up table bus macros are obligatory for all modern Xilinx FPGA families including Spartan-3, Virtex-IV, and Virtex-V that are not equipped with internal tristate drivers. A design flow based on LUT-based bus macros was presented in [LBM+06]. These macros are based on a previous project published in [HBB04].

In [KD06b, KD06a] LUT-based bus macros have been used to link partial modules in a vertically aligned resource slot style into a system that is based on Xilinx Virtex-IV FPGAs. The proposed system provides multiple different reconfigurable islands and a tool assists in instantiating and placing the communication macros at the module sides in a regular manner. This allows variable module sizes and relocating modules to different slot positions. The proposed system incorporates that Virtex-IV FPGAs can be reconfigured sub-column-wise in multiples of 128 4-bit look-up-tables. This allows adjusting the module height in multiples of a sub-column. The logic overhead for this communication architecture is enormous with two LUTs for each signal that is passing a resource slot boundary. The approach does not allow direct connections between partial modules and the complete communication is carried out via the static part of the system with multiple wide multiplexers.

By tiling the reconfigurable area in a few rows and in restricting modules into bounding boxes that are allowed to be one or more rows height, Silva and Ferreira designed a reconfigurable system allowing to place modules in a two dimensional fashion [SF08]. As illustrated in Figure 3.14, modules can only receive data from the left side and send data to the right side via LUT-based bus macros. Vertical routing is provided by special switching macros that can route an input stream from one row to another one that may be located in a different plane. To adjust modules of unequal width, route through modules can be instantiated that expand the interface of a module, so that multiple modules end at the same switching macro or to bridge modules with the static system.

LUT-based bus macros can be implemented in all horizontal and vertical directions. This is used in [OM07] for implementing reconfigurable systems with prerouted modules possessing macro connections at the edges for linking adjacent mod-
ules or for connecting the static system. By providing implementation alternatives offering varying shapes and different signal directions at the borders of the modules, the runtime system is able to plug modules together very flexible in a two dimensional grid style. This approach requires that all data streams are routed through a chain of adjacent modules making this approach unlikely for larger systems containing, many modules and many different data streams.

Dynamic Hardware Plugins (DHP) In [HLTP02], Horta et al. published an early concept for integrating partial modules with look-up tables that connect input and output antennas into a so-called “gasket area”. The gasket was used to overcome limitations of early FPGA design tools in strictly following area constraints for routing. Here, the idea is to leave a logic corridor between the reconfigurable islands and the static part of the system unused. This leads to an oversupply in routing resources inside the gasket for supporting the place and route tool not to use routing resources within the static system when implementing a reconfigurable module or not to use resources inside the reconfigurable islands for the static part of the system, respectively. Only the antennas area allowed to completely cross the gasket for linking partial modules with the static system. Obviously, this approach is very costly. Furthermore, with improved tools, the gasket technique has become obsolete.

Alternative Methods for Physically Linking Reconfigurable Modules

For the sake of completeness, alternative approaches for integrating reconfigurable modules into a system at runtime will be presented.

Instead of using the gasket technique to overcome the limitations of early design tools, Luk et al. [LSC97] proposed a solution, applicable to the vendor tools for the Xilinx XC6200 FPGA at that time. Starting from a static design, all modules are built one after another, using an incremental design flow. With their tool ConfigDiff, the authors have been able to generate differential bitstreams that contain just the differences between two full configurations. Obviously, this approach is not scalable, because it requires $|\mathcal{M}| \cdot (|\mathcal{M}| - 1)$ differential bitstreams for a system with $|\mathcal{M}|$ partial modules that can be exclusively loaded to the FPGA, if the modules can be configured in any order.

A similar approach was presented in [KT04] to partially configure any kind of FPGA. In this work, an incremental design flow was used to build a full static configuration bitstream and multiple complete bitstreams containing the static part and one or more exchangeable modules. By correlating the complete module bitstreams with the full static bitstream a resulting bitstream was produced that mostly contains the fraction of a particular exchangeable module. This correlated bitstream was compressed for the runtime bitstream repository. At runtime, a dedicated module bitstream can be decompressed and merged into the full static bitstream. The goal of this
3.4 Related Work on Communication for Reconfigurable Systems

Figure 3.15: a) Static modeling of a system that can behave either as \( P \) or as \( R \) depending on the control blocks \( C \) and \( C' \) (control signals are not shown). b) Cascading \( C' \) and \( C \) should behave like a pair of wires [LSC96].

approach is to reduce the bitstream repository size by preventing to store multiple full bitstreams. This technique and the ConfigDiff approach omit physical connections that are specially designed for partial runtime reconfigurable systems. Consequently, connections to the reconfigurable modules and some routing nets of the static system may be interrupted when swapping a module. This is because of some nets may be routed on different paths from each place and route run for implementing a module.

A straightforward approach for modeling reconfiguration with connection points on an FPGA was presented by Luk et al. [LSC96] based on the Ruby language [JS90]. Opposed to VHDL or Verilog, Ruby facilitates the encoding of layout information to a design. For example, ruby allows to express that function block \( A \) is arranged beside \( B \) as well as \( C \) is grouped beside \( D \) while \( A \) and \( B \) are grouped below \( C \) and \( D \): \( (A \leftrightarrow B) \downarrow (C \leftrightarrow D) \). These capabilities have been used in [LSC96] to guide the physical implementation of a dynamically reconfigurable design. The idea is to include control blocks into the design for switching between multiple function blocks as illustrated in Figure 3.15a). In the model, the control blocks behave like demultiplexers and multiplexers. Within the physical implementation, the control blocks represent plugs at fixed positions that act as connection points behaving like simple wires (see Figure 3.15a)).
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

3.5 Bus-Based Communication for Reconfigurable Systems

Buses provide the most common and cheap communication technique between multiple modules and memory. Therefore, buses can also be used for the communication with runtime reconfigurable modules. Comparable with plugging an extension card (e.g., a network adaptor) into a PCI backplane bus on a PC mainboard, a reconfigurable module should be as easily integrated on a reconfigurable device.

In this subchapter, new concepts of a bus-based communication architecture for implementing reconfigurable SoCs will be discussed and analyzed. The communication architecture includes physical connection techniques and is not restricting the overlaying bus protocol. Consequently, the communication architecture is compatible with most commonly used bus standards found in SoCs, including AMBA [Ltd07], CORECONNECT [Int99], AVALON [Alt07], and Wishbone [Sil02]. The aim of the bus architecture is to allow changes to the system, for example, by loading a new module configuration to the FPGA, without interfering other running modules connected to the same bus. Instead of using tristate bus lines (see Figure 3.16a)) that are often used for buses at board level or for backplanes, this work considers buses that are based on unidirectional signal lines that are typically linked together by multiplexers (see Figure 3.16b)) or logic gates (see Figure 3.16c)) (see also the footnote on page 108).

A bus is a set of signals where different signals or groups of different signals have special purposes, as shown in Figure 3.17. A bus provides a shared medium for the communication between various modules connected to the bus. These modules can be divided into two classes, namely i) master modules and ii) slave modules. Only masters are allowed to control the bus, e.g., by driving an address to the bus, whereas slaves are only allowed to respond to requests from a master. However, by the use of interrupt lines, slaves can notify a master to initialize a communication with a

![Figure 3.16: Common bus implementations: a) tristate driver based, b) multiplexer based, and c) logic-based.](image-url)
master. Table 3.1 presents an overview of signal classes found in typical on-chip buses. The table distinguishes between signals that are driven by masters (master write) or slaves (master read). Furthermore, the table differentiates if a signal is shared among different slaves, or if there is a dedicated (non-shared) connection between a slave or a master module connected to the bus.

If multiple masters are connected to the same bus, an arbitration scheme is required to resolve conflicts. For example, if two masters want to access a slave at the same time, besides masters and slaves, there must be a third class of bus subscriber namely an arbiter, connected to the bus. The communication of an arbiter takes only place with the master modules by using a of dedicated bus signal for each master (e.g., bus_request or bus_grant). However, with respect to reconfigurable on-chip buses, these bus signals form no extra problem class. For instance, the signal distribution of an interrupt request of a slave to a master is basically the same problem as the distribution of a bus request signal to an arbiter. Therefore, the communication with an arbiter will not be examined any further. All approaches presented in the following sections discuss the communication between a master (located in the static part of the system) and one or more slaves (implemented as partially reconfigurable modules). However, the techniques can be easily extended for multi-master operations on a reconfigurable bus architecture.

For the following, it is assumed that partially reconfigurable modules are separable from the overall system. This can be achieved by applying some logic and routing constraints to the interchangeable modules. For example, by applying area constraints in forms of bounding boxes, synthesis tools can be forced to fit modules exclusively within a specific region of the FPGA while further preventing the usage of logic resources located inside the bounding box by other parts of the system. In addition, all bus interface signals have to be routed via dedicated wires, such that a static part of a system can communicate with a dynamic reconfigurable part of the system.
3. *Intra-FPGA Communication Architectures for Reconfigurable Systems*

**Table 3.1:** Classification of bus signals. All bus protocols can be implemented by combining signals from the four problem classes listed in the table. Examples for each problem class can be found in Figure 3.17.

<table>
<thead>
<tr>
<th>master</th>
<th>shared</th>
<th>dedicated (non-shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>address</td>
<td>module_enable</td>
</tr>
<tr>
<td></td>
<td>data_out</td>
<td>bus_request*</td>
</tr>
<tr>
<td></td>
<td>write_enable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read_enable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>byte_select</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>data_in</td>
<td>interrupt</td>
</tr>
<tr>
<td></td>
<td>address_in</td>
<td>wait_request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bus_grant*</td>
</tr>
</tbody>
</table>

* Signal for master-arbiter communication

system over these wires only. The wires will be used for linking together the static and the dynamic part of the system among different modules placed into an area allocated for dynamic partial reconfiguration. This is comparable to connecting some cards via plugs in a traditional backplane based bus. Here, interface signals are connected via dedicated pins inside the plugs and the pin layout must be equivalent if the same card should operate in any plug.

There exists a lack of adequate constraints on wires inside commonly used synthesis tools that prevent the tools to directly map a signal to a specific (user-defined) wire or a set of wires. However, this lack can be avoided by utilizing special macros built of additional resources connected to both sides of the desired wire. With respect to the Xilinx design tools, it is possible to use any look-up-table or the attached flip-flops behind the LUTs for this purpose. These resources provide connection points for such a user-defined wire. Instead of assigning a wire directly within the tools, an associated macro is instantiated and placed to a user-defined position with the help of location constraints. Such macros act as plugs for the module interface signals (the top-level interface of the partial modules) that have to cross the module border. The here proposed ReCoBus and connection bar communication architecture will be physically implemented as monolithic hard macros that can be directly instantiated by the Xilinx design tools. The here used term "'macros'" should not be mixed up with so-called "'bus macros'" used in an alternative design flow provided by Xilinx (see also the paragraph on Xilinx bus macros on page 109).

Each quadrant in Table 3.1 represents a different problem class for a specific set of bus signals that has to be addressed individually when dynamic reconfigurable on-chip buses should be implemented efficiently. Based on an example with a master located in the static part of the system and some partially reconfigurable slaves
located in a set of horizontally aligned resource slots, multiple methodologies for implementing the different bus signal classes will be proposed and analyzed separately in the following four sections.

### 3.5.1 Shared Write Signals

Examples for write signals that are driven by a master and shared among multiple slaves are address lines, write data lines, or read/write control signals. The implementation of shared write bus signals is trivial for static only systems, where these signals can be freely routed on the FPGA. For dynamic reconfigurable buses, most related approaches are based on a macro approach [LBM+06] that requires some resources in addition to the wires inside every module connection point for providing shared signals. Consequently, the amount $S_{SW}$ of shared write signals would occupy the same number of look-up tables (or some other logic resources), multiplied by the amount of resource slots $R$ when the module connection points are explicitly provided within each resource slot. In addition, $S_{SW}$ LUTs are used for the connection to the static system as illustrated in Figure 3.18. Therefore, the resource overhead $L_{SW}^c$ (in terms of look-up tables) for implementing $S_{SW}$ shared write signals using the common macro approach is:

$$L_{SW}^c = (1 + R) \cdot S_{SW} \quad (3.4)$$

The examples in Figure 3.18 comprise a dummy sink that is required to arrange the routing of the last resource slot (slot R-1) just as homogeneous as the other slots. It is possible to avoid this dummy by simply extending the wire into the static area located right beside the reconfigurable area while leaving the end unconnected as an antenna. Alternatively, the dummy can be implemented by instantiating some look-up tables for terminating the shared write signal with the LUT inputs$^6$. This requires $\lceil \frac{S_{SW}}{k} \rceil$ $k$-input LUTs. As a further alternative, the last resource slot will be left with another internal routing as the other slots. As revealed in Section 4.1.3 this will not necessary restrict the module placement as the routing can be adjusted at runtime.

For reducing the overhead of the common macro approach, a so-called interleaved connection scheme has been proposed [KSHT07, KSHT08a, KHT08]. Here, a shared write wire is not tabbed within each resource slot but once every $N$-th slot, as shown in Figure 3.18b). The idea behind interleaving shared write signals is that in typical systems smaller modules (requiring less resource slots) also require smaller interfaces. Analogously, larger modules require a respectively wider interface. Consequently, interleaving allows reducing the resource slot width, and therefore, reducing internal fragmentation, without additional look-up tables inside the resource slots. In order to allow a free module placement to the resource slots, additional alignment

---

5The letters that are also used as indices will be underlined throughout this section.

6Connecting shared write signals to a LUT after the last resource slot will work more stable with the Xilinx design tools as compared to the antenna approach.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

**Figure 3.18**: Shared write distribution. a) Common macro approach requiring a LUT per shared write signal in each resource slot. b) Interleaved connection scheme exemplary for \( N = 3 \) interleaved chains for reducing the amount of connecting LUTs within the resource slots by \( N \). c) Implicit connection scheme without any connecting LUTs inside the resource slots. A dummy ensures a homogeneous routing also for the last resource slot such that it is compatible to all other resource slots.

Multiplexers in the static part of the system will ensure that a particular shared write signal is driven to an interleaved chain throughout the resource slots depending on the placement position of the currently selected module. Note that this approach only works for one selected module, hence it is unlikely to be used for control signals. However, the majority of write signals like write data and address lines will take advantage of this technique, and it may be combined with the common macro approach for distributing the remaining control signals. A more detailed discussion on interleaving bus signals can be found in Section 3.5.3. Let \( N \) be the number of interleaved chains and \( MUX(N) \) be the number of LUTs required to implement one \( N \)-input multiplexer, then the resource overhead (in look-up tables) for implementing \( S_{SW} \) shared write signals using the interleaving approach is:

\[
L_{SW}^{in} = S_{SW} \cdot MUX(N) + S_{SW} + \left[ \frac{S_{SW}}{N} \right] \cdot R
\]  

(3.5)

The logic overhead can be reduced further by connecting the reconfigurable modules directly to the internal bus wires that implement the shared write signals, as shown in Figure 3.18c). For allowing a relocation of modules connected to the bus, the routing has to be constrained, such that, the internal bus signals always occupy the same equivalent routing resources within each resource slot. In the common macro based approach proposed by Xilinx \[LBM^+06\], the routing is not required to be arranged in a uniform manner, because the flow follows an island reconfiguration style, where modules share a single fixed area over time. In slot or grid style placement, multiple modules can be located freely to different positions, hence, requiring a uniform communication architecture throughout all possible placement positions. With respect to the proposed communication architecture, a routing is called uniform if it is identical.
inside all resource slots and also at the slot borders. Using the implicit connection scheme is not directly supported by the Xilinx router. However, in Section 4.1.2 it is revealed how systems can be easily constrained for allowing this technique together with the standard Xilinx design tools. The logic overhead for the implicit connection scheme is one look-up table per shared write signal:

\[ L_{SWS}^{im} = S_{SW} \]  

(3.6)

Note that the logic overhead of the implicit connection scheme is independent of the amount of resource slots. Consequently, this technique allows a very fine resource slot grid without additional logic overhead.

### 3.5.2 Dedicated Write Signals

Besides shared write signals that are connected to all modules at the bus, it requires dedicated write signals for linking a signal to a specific module. These signals are typically control signals. An example for such a dedicated control signal is a module select signal that is not shared among multiple modules. In the following, this example will be used to describe how a master can selectively address a slave that is implemented as a reconfigurable module.

As a straightforward approach, it is possible to use non-uniform routing within the bus as illustrated in Figure 3.19a). In order to allow module relocation this requires implementing the dedicated write signals outside the resource slots. If a module configuration is loaded to the FPGA at runtime, it may be further required to merge the module bitstream into the configuration of the static system. This will connect the module with the heterogeneously routed dedicated write signals located in the static part of the system. Such a bitstream merging is obligatory when using Xilinx Spartan-3 or Virtex-II devices. Newer Xilinx devices such as Virtex-IV and Virtex-V enable sub-column-wise reconfiguration of the device, thus, supporting configuration merging by the FPGA architecture. However, this requires accurately following the device restrictions during floorplanning, where the FPGA is partitioned into a static region and multiple resource slots. If these restrictions do not correspond to the system requirements, this results in a lower logic utilization of the FPGA. However, the main limitation of a non-uniform routing is that it is not usable for a two-dimensional grid style alignment of the resource slots (see also Figure 3.19a)).

To overcome these limitations for module select signals, three novel techniques have been proposed and will be examined in the following: 1) a slot-wise rotated module select distribution scheme where a set of dedicated wires is homogeneously routed through the resource slots, 2) a counter-based module select scheme requiring half the amount of routing resources, and 3) techniques that are based on reconfigurable select generators where the number of routing resources scales logarithmically with the number of modules connected to the bus.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.19: Distribution of module enable signals a) using inhomogeneous routing within the static part of the system and b) homogeneous routing formed by a rotated module select signal distribution scheme.

Slot-wise rotated Distribution Schemes

This technique routes a set of wires, one for each resource slot and dedicated write signal, slot-wise from resource slot to resource slot. In each resource slot, the routing is rotated by one wire position and then passed to the next resource slot. As presented in Figure 3.19b), this scheme allows modules to connect to an exclusive dedicated wire in each resource slot at exactly the same relative position while maintaining a homogeneous routing within each resource slot.

This signal distribution scheme has the predicate of a uniform interconnect structure with respect to the module boundary while still being able to supply an individual signal to each slave by rotating the complete set of dedicated signals inside the module boundary. The technique can be implemented with wires spanning over multiple basic elements to perform the slot-wise rotation scheme, because of such wires, spanning over \( S \) consecutive BEs, will automatically rotate dedicated signals for up to \( S \) resource slots (see also Figure 3.5 that introduces, among others, the phrases wire span \( S \) and basic element BE).

If the number of resource slots \( R \), multiplied by the number of basic elements per slot \( W \), is larger than the span \( S \), this scheme can be cascaded to any arbitrary width by driving the incoming signal from the configurable address decoder to a new unused channel as shown in Figure 3.20.

Figure 3.20a) gives an example where \( R = 12 \) module slots of width \( W = 1 \) are connected with an individual module select signal for every slot by the use of local wires that span over \( S = 6 \) basic building blocks. The fat drawn module select signal points out how a dedicated signal is cascaded after \( S \) resource slots to another channel that spans again over \( S \) resource slots.

Figure 3.20b) reveals a second example where the resource slot width granularity is larger than one basic element \( W = 2 \). In this case, the module select signal has to be cascaded after \( \left\lfloor \frac{S}{W} \right\rfloor \) consecutive BEs, as shown by the fat drawn module select signal. The figure points out that this cascading can be applied within the same channel. Note that the routing from module slot to module slot is equivalent for
3.5 Bus-Based Communication for Reconfigurable Systems

Figure 3.20: Module select distribution by exploiting the regular structure of an FPGA architecture. Wires spanning over multiple basic elements (here $S=6$) automatically perform the slot-wise rotated distribution scheme, as shown in Figure 3.19b). a) Cascading the slot-wise rotating scheme when the number of resource slots $R$ is greater than the span of a channel (i.e., $R > C_S^1$). See also Section 3.3 that introduces the here used notation. Whenever a channel has connected $S$ dedicated signals to $S$ resource slots, we will use a new channel to cascade the scheme. For instance, slot 0 to slot 5 use channel $C_6^0$ to route dedicated signals from the configurable decoder while the last slots are cascaded via channel $C_6^1$. b) Cascading the slot-wise rotating scheme within a channel. If a resource slot is wider than one basic element (BE), i.e., $W \geq 2$, then the routing scheme can be cascaded within the same channel after $\left\lceil \frac{S}{W} \right\rceil$ BEs. Note that both cascading schemes have exactly the same routing structure within each resource slot.

Each possible resource slot, hence making this approach suitable for a free module placement to the bus.

By using implicit connections (see also Section 3.5.2), the slot-wise rotated distribution scheme will not require any logic resources inside the resource slots. However, there are plenty of look-up tables required for the address decoder logic in the static part of the system. Furthermore, the number of routing wires inside the bus scales unlikely with $\left\lceil \frac{R}{S} \right\rceil \cdot S$, thus, preventing this approach for a larger amount of resource slots.

Counter-Based Module Select Distribution

If a master shall be able to access any arbitrary combination of slaves by dedicated signals and if a module can be placed freely to each resource slot, the slot-wise ro-
Figure 3.21: Chipselect generation. In each module slot, the input value is passed through a combinatory Möbius-counter stage. Upon an input value of 000 at a slot’s input, the according module select signal is activated.

A Möbius-code is obtained by a ring-shifter, where the input shift value is the inverted output of the shifter. The longest non-repeating sequence of different encodings of such a ring-shift register will be used for generating the module select signals. An example for a 3-bit Möbius-code is the sequence \{000, 001, 011, 111, 110, 100\}. All Möbius-code words can be decoded with a single two-input gate. For example, the code word containing only ‘0’-bits can be decoded with a NOR-gate evaluating the most left and the most right shift register bit of a particular Möbius-code. Note, that this property holds for any width of a Möbius-code.

A module select generation, using a 3-bit Möbius-Code, is depicted in Figure 3.21 for six resource slots \(R = 6\). In each slot the incoming data is ring-shifted and the value of the wrap around wire is inverted. The result is supplied slot-wise to the next stage performing the same operation, thus forming a combinatorial Möbius-counter stage in each slot. When the input value of a slot is 000, the module select signal is activated. For this purpose, a single NOR-gate allows decoding the code at the input of a slot. This NOR-gate is further used as a macro resource such that high-level tools can easily apply this technique. If this approach is used in Xilinx Virtex FPGAs, the inverter will not demand an exclusive lookup table, because these devices provide dedicated inverters usable inside the logic resources that can be concurrently used with the LUTs. It is not required to decode a 000 inside the resource slots, it is only important that the same element of the Möbius-code sequence is decoded in every
single slot. When the system wants to activate a particular resource slot select signal, it has to apply a Möbius-code value to the bus that has been counted backwards in such a way that the count value reaches $000$ at the slot intended to be activated. For instance, in Figure 3.21 the configurable address decoder has to apply a $001$ at its output to activate slot 1. Note that all bus interfaces are equal. Thus, the modules using this technique are fully relocatable. In this example, there is always one slot selected. If there must be the situation that no module is selected, the last encoding (here $100$) is used for this purpose. A system may be designed in such a way that a master drives addresses directly to the combinatory Möbius counter.

**Reconfigurable Select Generators**

A completely different methodology to generate individual module select signals is based on reconfigurable module select generators that decode an address signal within each resource slot. These select generators are configured at two levels. After loading a module onto the FPGA device through partial reconfiguration (*device level*), it will be in a state where the module is deactivated (e.g., by applying a module wide reset). In this state, the module select generator can be configured at the second level, the *system level*, in order to write some specific data for the module individually. This data may include information for a reconfigurable module select generator that decodes bus signals in order to generate dedicated module control signals.

Figure 3.22 presents an approach tailored to distributed memory FPGA architectures, such as all Xilinx Virtex architectures, where look-up tables can be configured as memory and shift registers. Whenever the FPGA is configured with the logic of a reconfigurable select generator, one look-up table inside this logic is configured to use the shift register mode and this shift register is initialized with the same bit value at all positions. The cascading output of the shift register - normally used to build larger shift registers - is connected to the enable input pin of the same shift register. This signal is further used as the *module_reset* signal supplied to the logic of the module. The preload bit value has to be chosen to enable the shift register. By setting the shift registers enable input to be active high or low, it is possible to select whenever the reset signal to the module should be active high or low, too. After configuring the FPGA with the complete module, the system can configure the reconfigurable module select generator in a second step.

This second step is performed by shifting in a new look-up table value to the shift register by the *config_data* input and the *config_clock* input of this register. The first bit shifted into the register must be set inverted to the shift register init value in order to lock the shift register after finishing the second level reconfiguration process. This first shifted value will be stored at the cascade output of the shift register that is connected to the shift enable input for locking the shift register after logic level configuration. After this, a function table for decoding the *bus_enable* signal must follow. Typically, the function table will contain a one-
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.22: Reconfigurable select generator based on a shift register primitive with a self-locking feedback loop. At FPGA reconfiguration time, the shift register is enabled by configuring a 1 to all LUT values (equal to initializing the shift register with 1 values). In the following system reconfiguration step, a configuration word starting with a 0 followed by a one-hot encoded sequence is shifted into the LUT (via config_clock and config_data). Therefore, the shift register will lock itself when a complete new table has been shifted into the LUT. After finishing this process, module_reset is automatically released, and the register is used in LUT mode, and thus, generating the module_select signal by decoding the bus_enable signal. A module_read can only be activated after the two reconfiguration steps.

hot encoded value. Assuming the example in Figure 3.22 with a module_select signal to be activated only when the bus_enable signal is equal to 0011, then the value 0001 0000 0000 0000 has to be shifted into the look-up table. Note that in the case when bus_enable is equal to the maximum value (e.g., 1111 in the example in Figure 3.22), the module_select signal will take the signal value of the cascading output locking the shift register. Thus, it is possible to generate \(2^k - 1\) individual module select signals when using a \(k\)-input look-up table as a shift register inside the reconfigurable module select generator. As a consequence, this approach requires for \(M\) modules the minimal theoretical amount of \([ld(M + 1)]\) internal bus wires in order to generate individual module select signals for each module. Note that this alternative scales with the maximum number of modules \(M\) and that it is independent of the number of resource slots \(R\) provided by the reconfigurable bus.

In order to protect the shift register against unintended shift operations during reconfiguration of the FPGA, the routing information should be written prior to the init value of the shift register look-up table of the reconfigurable module select generator. Alternatively, the init value can be written another time after the module is written.
3.5 Bus-Based Communication for Reconfigurable Systems

Figure 3.23: System composed of a master and some modules of different widths. Each module requires one module select block that contains the reconfigurable select generator shown in Figure 3.22.

completely to the FPGA.

This module select generation methodology has the predicate that it is completely independent with respect to the resource slot position of a module at the bus. Figure 3.23 presents an example of a system using the module select logic introduced in Figure 3.22. Due to the homogeneity of the bus interconnect resources, it is possible to relocate modules on the bus. Each time, after a new module is configured at the FPGA device level, this module is the only one with an activated shift register in its configurable module select generator logic. This holds true under the assumption that the system also performs the second reconfiguration step at the system level before writing the next module configuration to the FPGA. Note that a selective reconfiguration of the shift register look-up table values can be used anytime to apply module_reset and to change the value of the bus_enable signal that will actually select the connected module.

If the address ranges of the different reconfigurable modules can be chosen freely and within one consecutive address block, then the bus logic may be materially simplified. In this situation, the current master can directly drive all low address lines on the bus via shared write signals for accessing a register file inside each individual module. The bus_enable signal is then connected straight to the next higher set of address lines of the master. Finally, all other higher address lines are used to decode the address range of the modules connected to the reconfigurable bus. Assuming the system, illustrated in Figure 3.24, as an example: The address range for a module’s individual register file is chosen to be 8-bit (256 registers) and a maximum of up to \( M = 15 \) modules may be connected to the bus at the same time, while mapping the complete address space of the reconfigurable modules at the upper limit of a 16-bit address range. Then the address lines \( A_7 \ldots A_0 \) of the master will be connected
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.24: a) Example of a reconfigurable system with a 16-bit address bus, driven by a master. The eight least significant address bits are linked directly to the modules via shared write signal wires. The module select logic blocks contain reconfigurable select generators, (see also Figure 3.22) interconnected by an internal bus_enable signal. This signal contains a 1 at all positions if all four most significant address bits do not contain the base address prefix F---. In the case that $A_{15}, ..., A_{12}$ match the prefix, the reconfigurable select generators will decode the address bits $A_{11}, ..., A_{8}$. b) The corresponding memory map for the 16-bit address space.

directly to the bus via shared write signal wires. The bus internal bus_enable lines will contain the next address lines ($A_{11}, ..., A_{8}$) if all remaining higher address lines ($A_{12}...A_{15}$) contain a 1. In the case that the remaining higher address lines ($A_{11}, ..., A_{8}$) are not in the upper limit address space (F---) that is evaluated by the shown NAND-gate, the bus_enable lines will contain a 1 for every line, thus preventing the selection of any module. This is because of the highest look-up table entry is reserved to encode that no modules are accessed.

The reconfigurable select generator approach, depicted in Figure 3.22, can also be used for multicast bus transfers, where more than one reconfigurable module can be accessed at the same time. In this case, the shift registers of the reconfigurable select generators of two or more modules will contain a 1 at the same position. For instance, assuming that we have a module 1, mapped to the bus_enable address 0001, and a module 2, mapped to 0010, and that both modules should further be accessible at the multicast address 0100. In this case, we have to shift the value 0100 1000 0000 0000 into the look-up table of the select generator in module 1 and the value 0010 1000 0000 0000 into the LUT in module 2. Note, that both modules can be accessed individually or simultaneously. This multicast operation can be used for write and read modes. In the latter case, the data outputs of all selected modules are bitwise linked by OR-gates (see also Section 3.25).
3.5 Bus-Based Communication for Reconfigurable Systems

Online Bitstream Manipulation. It is possible to omit the second configuration step at the system level for setting the reconfigurable select generator (see Figure 3.22) by manipulating the configuration data at runtime. If multiple instances of the same module are intended to be connected to the bus at the same time, different look-up table values are required for each module. Consequently, the LUT cannot be hard coded directly in the partial module bitstream, if multiple instances of the same module have to be loaded into the same reconfigurable system. This can be performed during the reconfiguration phase by replacing the fraction of the reconfiguration data that is carrying the look-up table values of the module select comparator.

To determine the exact positions of the configuration bits describing the select generator, a bitstream parser was written that provides (among plenty other information) the exact positions of the configuration bits of the select generator LUTs. This tool supports all Xilinx Spartan-3 and Virtex-II/IIPro FPGAs. Some configuration data details for Virtex-II FPGAs including the positions of the look-up table configuration bits have been presented in [ZAT06].

As compared to the original reconfigurable select generator approach, the online bitstream manipulation avoids the logic overhead required in the static part of the system for shifting in the comparator values into the select generator look-up-tables. Furthermore, the bus internal wires for distributing the configuration clock and the configuration data signals are now useable for implementing the module itself.

Regardless of whether the LUT-values are set by shifting in a new table or by manipulating the configuration bitstream at runtime, the logic overhead $L_{DW}$ for providing $S_{DW}$ dedicated write signals in a system with up to $M$ simultaneously running modules is:

$$L_{DW} = (R) \cdot S_{DW} \cdot \left[ \frac{M}{2^k - 1} \right] + S_{DW}$$

This summarizes the logic overhead in the resource slots and the connecting LUTs that is required to connect $S_{DW}$ dedicated signals in each resource slot. One $k$-bit LUT that evaluates an internal bus enable signal to a module select can decode $2^k - 1$ different modules. One codeword is required for encoding that no module is selected. If more individual modules have to be accessed ($M \geq 2^k$) it is possible to combine multiple LUTs to a larger one on all Xilinx FPGAs. However, the amount of 15 individually addressable modules in the case of $k = 4$ bit look-up tables will be sufficient for most practical systems. It is also possible to share the same encoding of a dedicated write signal among multiple modules. For example, by mapping the register files of two modules into disjoint address ranges. Consequently, the logic overhead will be $(1 + R) \cdot S_{DW}$ LUTs in practical systems.

Figure 3.23 depicts that one reconfigurable select generator is sufficient per module and not per resource slot. This can be used when a ReCoBus integrating modules in a static only system. However, as presented in the following section, the outputs of the module select logic will be used to control access among different module outputs.
within the bus. This control is required within all slots for allowing modules to be exchanged without interfering any transaction on the bus.

### 3.5.3 Shared Read Signals

As a counterpart to shared write signals, a bus also has to provide signals in the opposite direction. Therefore, selected modules must be able to drive signals to a set of shared wires of the bus. An example for shared read signals is the read data bus from several slaves to a master. As already mentioned in the introduction, typical on-chip buses use unidirectional wires in order to avoid on-chip tristate buses as depicted in Figure 3.16a). Figure 3.16b) and Figure 3.16c) present examples of common on-chip bus implementations.

Reconfigurable buses have to be designed in a uniform and modular manner for allowing to include the bus structure directly within the modules connected to the bus. This was proposed before for read signals in [HBB04] and [HKKP07b]. There, the large OR-gate in Figure 3.16c) is split into small chunks of two-input OR-gates for arranging the signal routing between the resource slots in a uniform and modular manner, as illustrated in Figure 3.25. The output of the last OR-gate represents the shared signal supplied to the destination in the static part of the system (here a master). Such a regular construct will be called a **distributed read multiplexer chain** in the following.

Under all circumstances, the bus logic has to prevent modules to drive a logic value to a shared signal if it is not selected. As shown in Figure 3.25, this can be achieved by an additional AND-gate that fits into one look-up table together with an OR-gate. For this purpose, the enable signals \( \text{en}_i \) shown in Figure 3.25 are connected to the module_read output signal of a reconfigurable select generator (see 3.22) where each resource slot contains its own select generator. As a consequence, the output data of an individual module can only be applied to the bus if 1.) the corresponding module select generator has been properly configured at system level, 2.) the module is selected by an according bus_enable assignment from the static system, and 3.) a read operation is requested by activating the bus_read signal (see also Figure 3.22). Note that all approaches presented in Section 3.5.2 are suitable to distribute a dedicated write signal to each \( \text{en}_i \) signal, but the reconfigurable select generator is the most adequate one. If not only a single shared read signal is connected to the master but many, it requires one look-up table for each shared read data line in each resource slot. In this case, the \( \text{en}_i \) select signal inputs of the LUTs, used to implement the read multiplexer chains within one resource slot, are connected to the same bus_read signal of the same reconfigurable select generator. Then, the logic overhead \( L_{SR}^{SC} \) (in terms of LUTs) for implementing \( S_{SR} \) shared read signals by simple read multiplexer chains is:

\[
L_{SR}^{SC} = (1 + R)S_{SR}. 
\]  

(3.8)
3.5 Bus-Based Communication for Reconfigurable Systems

Figure 3.25: Implementation of a shared read signal via a distributed logic-based bus. The uniform structure of the distributed read multiplexer chain allows a flexible connection of modules to the bus.

**Performance Considerations**

Besides the logic overhead, shared read multiplexers increase the combinatorial path by one look-up-table for each resource slot. This may considerably limit the maximum amount of allowed resource slots in a system when implementing this structure. Before discussing further details of the ReCoBus structure, the relevance of this effect will be quantified. The impact of increasing the amount of resource slots was determined for a Xilinx Virtex-II device (XC2V6000-6) for one horizontal distributed read multiplexer chain of different sizes. In order to obtain realistic timing values, the propagation delay between two I/O pins was measured. In the test circuits, a signal is routed directly to the look-up table which represents a connection to the resource slot with the most critical propagation delay. On the way back, this signal is propagated through \( R \) LUTs in every \( W \)-th logic column. The results, presented in Figure 3.26, quote the propagation delay between the two I/O-pins.

The width \( W \) of a resource slot was varied between one and eight basic elements called CLBs for Xilinx FPGAs. Figure 3.26 indicates how the pin-to-pin propagation delay increases with the amount of resource slots \( R \) provided by the bus. This delay provides a realistic estimate of the propagation delay for a master accessing a module. The figure also allows determining possible parameters of the reconfigurable bus if a certain clock frequency is required. A 100 MHz system clock, for example, is equivalent to 10 ns of propagation delay allowed for the combinatorial paths, including routing delays and the flip-flops (setup-hold times and clock to output delays). In this situation, a reconfigurable bus that is based on a distributed read multiplexer chain is limited to a size of \( R = 8 \) resource slots when width of the slots is chosen to \( W = 2 \) CLBs, or in the case of \( W = 6 \) CLBs, the bus is limited to just \( R = 4 \) resource slots. Note that in the first case, the complete bus is \( R_{BE} = 8 \cdot 2 = 16 \) CLBs wide, while in the second case, the bus spans over \( R_{BE} = 4 \cdot 6 = 24 \) CLBs, with \( R_{BE} = R \cdot W \).

For a sake of completeness, the pin-to-pin propagation delay for an alternative bus implementation using a dedicated multiplexer provided in the CLBs of the Virtex-II architecture was measured, too. The performance of this alternative was slightly
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.26: The impact of the bus size on the propagation delay. The graphs show how the propagation delay increases with the number of resource slots \( R \) and by the width \( W \) of the resource slots. The values are determined for Xilinx Virtex-II XC2V6000-6 FPGA.

below the look-up table approach presented in Figure 3.26.

The long combinatory path of the read-multiplexer chains is inapplicable for pipelining, as this would lead to variable latencies depending on the module placement position at the bus. However, by duplicating the amount of shared read signals while using them as multi-cycle paths, the throughput can be enhanced to a wide range of speed and elongation requirements. This is illustrated in Figure 3.27b) for enhancing throughput and in Figure 3.27c) for extending the length of a read multiplexer chain.

In order to increase flexibility and reduce the internal fragmentation in case that a slot is not completely filled with the logic of a module, it is advantageous to keep the width of the resource slots small to reduce internal fragmentation (see Section 3.1). Nonetheless, it is possible to apply the read multiplexer methodology to large scale systems with small slot widths, while still keeping the bus operation frequency high. Instead of using one distributed read multiplexer chain that links one slot after the other, multiple chains can be used in an interleaved fashion, as illustrated in Figure 3.28. As a consequence, buses implemented with a wire span of \( S \) basic elements
3.5 Bus-Based Communication for Reconfigurable Systems

Figure 3.27: a) Critical path of a read multiplexer chain, passing four look-up tables, and the according waveform. b) By duplicating the read multiplexer chain, the throughput is doubled, while each chain holds the same propagation delay as in a), by setting a multi-cycle path to the chain. c) Instead of increasing the clock frequency, the additional read multiplexer chain could alternatively be used for extending the length of the chain while allowing the same clock frequency as in a).
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

and a module width of $W$ BEs can save up to $S/W$ times the propagation delay of a single look-up table. Furthermore, the usage of wires, spanning over a larger amount of elementary building blocks, leads to a lower overall signal propagation delay. This is because less switch matrices have to be passed for implementing the read multiplexer chain. Note that in case, the FPGA architecture does not offer the desired wire span $S$, it is easily possible to cascade multiple wires to route a signal in a uniform manner over longer distances by using the approaches presented in Figure 3.20. For connecting a module in each resource slot with this technique, the module must provide access to all bus signals within each resource slot. However, this flexibility for the dynamically reconfigurable bus comes along with some unused logic resources in the case when modules span over multiple slots, as usual in a fine-grained resource slot layout. The examples in Figures 3.25 and 3.28 show how a single shared read signal is connected through chains of look-up tables. In order to build buses of any arbitrary bit-width, multiple chains, one upon the other, have to be instantiated. Alternatively, slots may be more than one elementary basic block (BE) wide, thus allowing to cascade multiple chains side by side or in any mixed combination.

**Interleaved Bus Signal Distribution**

Different modules may have individual interface widths in terms of data bus widths. The amount of address signals and modules may require various amounts of resources, in terms of slots, to implement the logic of a module. However, modules of higher complexity typically demand a wider interface and vice versa. For example, although a simple UART is typically connected via an 8-bit bus interface, a complex Ethernet core will be linked in most cases with a 32-bit wide interface. In order to benefit from this observation, a *multi-slot read technique* is proposed, as revealed in Figure 3.29. In this signal distribution scheme, not all of the $S_{SR}$ shared read signals are connected in each resource slot, but onto subset of $\lceil \frac{S_{SR}}{N} \rceil$ signals. The signal subsets form $N$ individual read multiplexer chains that are connected once per $N$ resource slots. Furthermore, the $N$ chains are interleaved along the bus by dis-

---

**Figure 3.28:** Interleaved read multiplexers. Shared read signals from different slots (data out) are linked by interleaving multiple distributed logic-based bus chains.
3.5 Bus-Based Communication for Reconfigurable Systems

placing the logic of the chains among each other in N adjacent slots. This leads to a
repeating connection scheme with an identical logic and routing footprint throughout
all resource slots. Interleaving reduces the logic cost per slot to \( \frac{S_{SR}}{N} \) as compared to
\( S_{SR} \) for the original bus implementation depicted in Figure 3.25.

In order to establish a better comparison with existing systems, it is assumed that
each signal shown in Figure 3.29 represents one byte and that the data size depth
the bus is up to 32 bits wide. Then module 0, possesses a 16-bit interface while
module 1 uses the complete interface width of the bus. The bus is arranged by
four interleaved logic-based bus chains for the shared read signals. In this example,
modules that are 1, 2, 3, or 4 consecutive resource slots wide can connect a module
containing an interface widths of up to 8, 16, 24, or respectively 32 bits. These slots
can be as narrow as the width of one basic element with respect to assumed FPGA
model (see Section 3.3).

When a master reads data from a module, multiplexers inserted between the differ-
etent independent multiplexer chains and the master will switch the different bus chains
in such an order that the master can read a correctly aligned 32-bit word independent
of the position of the start resource slot of the currently selected module. The select
signals for these alignment multiplexers can be generated by a tiny look-up table.
This table is written by the operating system whenever a new module is included
into the system and it delivers the alignment select signal according to the current
module select address. Consequently, this look-up table requires as much registers
as the maximum amount of simultaneously integrated modules \( M \) in the system. The
interleaving factor \( N \) affects the width of the registers. The total amount of flip-flops
in the table is then: \( M \cdot \lceil \log_2 N \rceil \). Note that all Xilinx FPGAs allow using some or all
look-up tables alternatively as small registers, thus allowing to implement the align-
ment select generation efficiently without dedicated flip-flops. If the module is placed
at another slot position, the multiplexer select signals have to be adjusted by the start
resource slot index, modulo the amount of interleaved distributed logic-based bus
chains \( N \), as it is illustrated in Figure 3.30. The register file is directly addressed by
the same address lines that are decoded within the bus to produce the bus_enable
signal (see Figure 3.23). Note that the overhead for the alignment multiplexers and
the select look-up is not related to the amount of resource slots provided by the bus.

3.5.4 Dedicated Read Signals

The last class of bus signals not considered so far are dedicated (non-shared) read
signals that allow a module to transmit state information to the static part of the
system. Examples for this class of bus signals are interrupt lines and wait request
signals\(^7\). These two examples have one main difference: as interrupt lines can be

\(^7\)Wait request signals are used by an accessed module (e.g., a slave) to request extra idle cycles for
a bus transfer. The wait request concept allows including a variable amount of wait states for the
3. *Intra-FPGA Communication Architectures for Reconfigurable Systems*

**Figure 3.29:** Multi-slot read technique. The wider the required module bus interface is, the more slots are used to connect shared read signals by independent chains of interleaved distributed logic-based bus chains. A set of multiplexers that are inserted in front of the master ensure that the data signals are aligned in the correct order regardless of the respective start position of the currently selected module at the bus.

**Figure 3.30:** Alignment multiplexing for the multi-slot read technique. Whenever a module is accessed at the bus, the alignment multiplexer select signal must be set to the resource slot index modulo the interleaving factor $N$. In this example, the start resource slot index of the currently selected module is 1 and the interleaving factor is 4. By setting the alignment multiplexers to input 1, the module data ($d_{31}, \ldots d_{0}$) is send correctly adjusted to the destination port ($D_{31}, \ldots D_{0}$) within the static system.
activated in any combination, all used interrupt lines form a signal that is binary encoded. In contrast to this, a wait request signal may be activated only by a single selected module, thus leading to a one-hot encoding over all wait request signals of the modules. Therefore, wait request signals can be connected to a master using the distributed logic-based bus implementation as presented in Figure 3.25. By enabling a specific module by a master, it is clear that only this activated module may enable the wait request signal.

For dedicated read signals, such as interrupt lines, the approaches presented in Figure 3.19 and Figure 3.20 for dedicated write signals could be applied with the change of the signal direction. These techniques, however, are not applicable to large scale systems that are combined with a small slot width $W$. This is because of the required routing and logic resources to implement these techniques that unlikely scale with the amount of module resource slots. The same holds for related approaches that use a demultiplexer for connecting a dedicated read signal per resource slot to the static part of the system (see Section 3.4.3). In the following, alternatives will be examined where the amount of routing wires that are required for connecting dedicated read signals is constant or scaling with the number of modules connected to the bus.

**Time-Multiplexing**

In many systems, it may be acceptable to handle dedicated read signal events (e.g., an interrupt request) with a delay of a few clock cycles. Then, the approach presented in Figure 3.31 can be applied, where such signals are captured using time-multiplexing via a reconfigurable bus. Whenever a module has a dedicated read signal output, this signal has to be connected to a distributed logic-based read multiplexer chain (see Figure 3.25). The particular dedicated module signal is chosen by a select generator, analogous to the one presented in Figure 3.22. The select generator can be configured to send state information from a specific slot to a particular flip-flop and, consequently, to a particular dedicated read signal line, connected within the static part of the system. If required, modules may interconnect multiple dedicated read signals via several resource slots towards the static system.

Implementing time multiplexed shared read signals requires one LUT per resource slot for the distributed read chain and an amount of LUTs related to the total number of modules $M$ (assuming that each module requires one request signal to be transferred to the static part of the system). Beside the logic in the resource slots, it further requires some logic in the static part of the system for the flip-flop select decoder and the state counter. The logic overhead for implementing shared read signals by
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.31: Time-multiplexed connection scheme for dedicated read signals. A counter enables cyclically a set of flip-flops and simultaneously an according select generator within a definable resource slot for writing the request state (e.g., an interrupt or a bus request) of a module to a flip-flop in the static part of the system.

\[
L^{TM}_{DR} = R \cdot \left\lfloor \frac{M}{2^k} \right\rfloor + M \cdot \left\lfloor \frac{M}{2^k} \right\rfloor + \log_2 M. \tag{3.9}
\]

Logic-Based Demultiplexing

While it might be applicable to distribute an interrupt signal in a time-multiplexed manner, this kind of distribution is disadvantageous for bus request signals. For example, this may be disadvantageous if multiple masters at the reconfigurable bus can request control over the bus only by an elongated time required for arbitration. As a consequence, each master should have an individual connection to the bus arbiter. One solution for this problem is to use a single distributed logic-based bus signal (see Figure 3.28) for each possible bus request instead of one for all in a time-multiplexed fashion. A solution using this technique is proposed in Figure 3.32. In order to gain flexibility for module placement at the reconfigurable bus, the approach allows configuring a bus_request signal to a connection point within a specific resource slot at the system level (by the operating system). The logic in each slot provides a demultiplexer and a configuration register for the select signal of this demultiplexer. Figure 3.32 points out that such an approach comes along with a considerable logic overhead that is at least \(M \cdot R\) look-up tables for \(M\) modules connecting a bus request signal in each resource slot. A related approach [HKP06] uses one chain of look-up tables per resource slot \(R\) to link one dedicated signal per slot to the static part of...
3.5 Bus-Based Communication for Reconfigurable Systems

Figure 3.32: Configurable bus request demultiplexer. In each resource slot, with a slot bus_request signal output, this signal is connected to multiple distributed logic-based read signals (like three in this example). Each read signal is selected on a different select value that is stored in a set of flip-flops for each resource slot driving a bus request signal. Such a set of flip-flops can be written individually on the system level.

As already mentioned for the multi-slot read technique, modules typically span over multiple elementary resource slots. This holds true especially for masters connected to the bus, as they consume a considerable amount of logic just to implement the master interface. Based on this observation, a multi-slot dedicated read technique, as presented in Figure 3.33, was proposed, where \( N \) demultiplexers are built upon \( N \) interleaved, horizontally aligned bus request chains. Consequently, the resource overhead can be reduced by \( N \). After device reconfiguration, the multiplexer in each resource slot is configured to pass through the preceding incoming signal. In a second system level configuration step, the bus request signal of the module may be connected in one particular slot to one of the horizontal bus request chains.

The interleaved horizontal chains can be directly implemented with routing wires, spanning \( N \) resource slots (like four as shown in the example). However, the routing may be cascaded to any desired span by using the approaches presented in Figure 3.20. The chosen span determines the number of interleaved horizontal chains and consequently the number of dedicated read signals for all modules together. If this span does not accommodate all required dedicated read signals, multiple inter-
Figure 3.33: Multi-slot dedicated read technique, exemplarily for connecting bus request signals. a) General structure. Multiple distributed logic-based bus request signals are interleaved according to the routing infrastructure offered by the FPGA architecture (in this example, quad lines spanning over four resource slots). In each resource slot, a reconfigurable multiplexer allows selecting between the local bus request signal or the incoming preceding request signal. Each distributed logic chain can be configured individually. b) Configuration example for two modules connected to the reconfigurable bus. Note that all local bus request signals within a module are connected to the same bus request source of the module.

leaved channels can be instantiated, one above the other.

The configurable multiplexers shown in Figure 3.33 and Figure 3.34a) are used for a second reconfiguration step at the system level. This means that after writing the configuration to the FPGA, the values of the shown LUT can be adjusted by a driver without using the FPGA configuration port again. Here, the same approach, as presented in Figure 3.22 for the reconfigurable select generators, is used. This approach can be efficiently implemented on all current Virtex or Spartan FPGA devices, shipped by Xilinx Inc., within a single look-up table per configurable multiplexer. As shown in Figure 3.34b) exemplary for a 3-bit look-up table, this look-up table also has to be used in the shift register mode. In the following, it is exemplarily assumed that $N = 4$ horizontal traces are interleaved with a span of 4 resource slots. Then the con-
3.5 Bus-Based Communication for Reconfigurable Systems

Figure 3.34: a) Switching element for a configurable multi-slot demultiplexer chain that is suitable of implementing dedicated read signals. b) Implementation alternative tailored to distributed memory architectures, when look-up tables can be used as a shift registers (like in all Xilinx Virtex FPGA families). The table in the right gives the values to be shifted into the look-up table in order to set the switching element to connect to the bus_request signal or to route through bus requests from preceding resource slots.

Configuration data output config_data from the configuration unit is connected to the first $N$ configurable multiplexers in the first $N$ resource slots (see Figure 3.33). The configuration logic is located in the static part of the system. Each cascading output of the first $N$ shift registers is connected to a further configurable multiplexer, located $N$ resource slots further. This is repeated over all resource slots, and thus, each interleaved multiplexer channel forms one shift register over the look-up tables that are all instantiated in shift register mode. Therefore, there exist $N$ interleaved shift register chains. Each shift register chain comes along with its own clock signal. This allows to shift data from the config_data terminal into a selectable shift register chain, without interfering the other chains and consequently, the present signal state in all other chains. The look-up table in Figure 3.34b) is configured either to pass through the incoming request signal from a preceding look-up table or to connect the chain to the local bus request signal. The according entries for the example in Figure 3.34b) are given in the adjoining table. In order to configure the bus request signals as shown in Figure 3.33b), the values from the table have to be shifted into the corresponding shift register chain. For example, to connect the signal bus_request_0 in resource slot 4 to the local bus request signal of module 1, the sequence 0101 0101 has to be shifted with config_clock_0 followed by the sequence 0000 1111 for passing through the signal in resource slot 0. Analogous, bus_request_1 is connected to module 0 via resource slot 1 by shifting in the sequence 0000 1111 0101 0101 into chain 1. Note that this will also set the reconfigurable multiplexer in slot 5 to route through potential bus requests from preceding slots. Whenever the resources of
Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.35: Dedicated read signal distribution by FPGA configuration data manipulation. A request output from a module is driven by a connecting LUT vertically over a set of horizontally routed wires. The routing resources are chosen such that a single multiplexer within the switch matrices can be configured for either routing through the horizontal wire or connecting vertically to the request signal.

When a module at the bus are removed, the system can shift in the sequence 0000 1111 into all look-up tables of the utilized bus request chain for disconnecting the local bus request signal from the bus. Instead of using individual clocks for each interleaved chain, a single clock can be used in combination with enable logic to individually activate a specific shift register chain. Note that the configuration clock signals are distributed by using local wires instead of using global clock trees.

Demultiplexing through FPGA Configuration Manipulation

All presented approaches for connecting a dedicated read signal from a particular reconfigurable module to a destination in the static part of the system have in common that they omit a modification of the configuration data written to the FPGA. The only exception, where the configuration data is required to be modified during the reconfiguration process, is the address information that determines the position of the reconfigurable module on the device. For all Xilinx devices, the required documentation for changing address information inside the configuration data can be found in the respective user guides [Inc07].

Instead of implementing demultiplexers that connect a request output signal to a horizontal dedicated routing resource towards the static system, it is also possible to use the switch matrix resources of the FPGA fabric. An example for this approach is illustrated in Figure 3.35. Here, the amount of horizontal routing tracks for implementing dedicated read signals scales with the amount of modules located at the bus. This is an important improvement against all other presented approaches that require at least one LUT per resource slot and typically many more in most practical systems. Figure 3.35 reveals that one connecting look-up table is sufficient per module and dedicated read signal within the resource slots. This connection may be carried out by sharing resources that are used to implement other signal classes, such
3.5 Bus-Based Communication for Reconfigurable Systems

as dedicated read signals. The maximum number of concurrent, horizontally aligned, dedicated read lines has to be specified at design-time and is related to the maximum amount of simultaneously integrated modules.

Each module is constrained to drive its outgoing dedicated read signal (typically an interrupt or bus request signal) straight vertically across a set of horizontal wires that are routed in a homogeneous manner from resource slot to resource slot towards the static part of the system. Within the FPGA fabric, there exist switch matrices at the crosspoints. Horizontal and vertical wires for implementing dedicated read signals have to be selected such that the horizontal output wire towards the static system can either be connected with its preceding horizontal input wire from the preceding resource slot or with the vertical request signal output of a particular module. The crosspoints are part of the FPGA routing resources and require no logic in terms of look-up tables. The required configuration data to modify the setting of a crosspoint can be generated on-the-fly by exchanging fractions of the original module configuration data at runtime The necessary bitstream information is not published but could be easily figured out by reverse engineering. The configuration data of Xilinx FPGAs was identified to be highly regular and very suitable for manipulations at runtime. Alternatively, a set of configuration data templates, one for each dedicated read signal crosspoint, can be built offline and stored in a repository for the runtime system.

3.5.5 Module-to-Module ReCoBus Communication

For all proposed bus-based interconnect techniques, communication with a reconfigurable module takes only place always with the static part of the system and two reconfigurable modules cannot directly communicate with each other. Some related approaches \cite{LCK07, HKKP07b} propose buses, allowing direct links between two reconfigurable modules with the help of a bidirectional read multiplexer chain (see Figure 3.36). However, these approaches require three times more logic as compared to simple read multiplexer chains and they do not allow to use the write and read signals concurrently for simultaneous read and write operations by the static part of the system. Furthermore, communication structures for direct bus transfers within the reconfigurable area are unsuitable to be used for interleaved buses, because this would demand an extra alignment multiplexer in each module.

If a ReCoBus is implemented on Xilinx-FPGAs, it is possible to share the same look-up tables for read and write data signals. Furthermore, these signals can be used concurrently, hence data can be sent from one reconfigurable module to the static part of the system and directly back to another reconfigurable module connected to the bus. With this procedure, it is possible to communicate between two reconfigurable modules connected to the same bus. This is illustrated in Figure 3.37 for a reconfigurable master module that accesses a slave module, which is also located in the reconfigurable area. In this example, the signal path is propagated three times towards the static part of the system: i) for the arbitration (step 1 and step 2), ii)
Figure 3.36: a) Unidirectional read multiplexer chain versus b) bidirectional multiplexer chain [HKKP07b]. The unidirectional multiplexer chain a) requires one look-up table per resource slot and signal bit, while the bidirectional variant b) requires three times more LUTs. Beside looping back the output of the unidirectional chain to the reconfigurable modules (e.g., for transferring data from one partial module to another one) it is possible to use the read and write signals concurrently, hence, allowing the static part of the system reading one partial module while sending data to another one at the same time. This is not possible in case of the bidirectional multiplexer chain. Interleaving is only applicable to the unidirectional chain and allows faster operation despite the routing in backward direction. Note that the signal in backward direction can be pipelined in the static system.

for driving the address from the master to the slave (step 3 and step 4), and iii) for sending the data from the slave to the master (step 5 and step 6). By interleaving the address and data bus signals, it is possible to reduce the combinatorial depth (in terms of look-up tables), and hence, the overall propagation delay. The throughput can be further enhanced by pipelining the signal path within the master, the slave, or in the static part of the system. Experimental results on the propagation delays and on pipelining will be presented in detail in Section 3.7.1.

3.5.6 Buses for Two-dimensional Placement

In the introduction to this chapter, it was demonstrated that the resource slots shall be dimensioned small in order to reduce internal fragmentation. The waste of logic - due
Figure 3.37: Example of a reconfigurable master module reading a register from a reconfigurable slave. In the first step, the master requests bus access from the arbiter. Next, the arbiter grants bus control to the master (step 2). Upon this grant, the master drives the slave address to the static part in step 3. In the case of an interleaved bus, this address has to be firstly aligned according to the start position of the master that is currently selected by the grant signal. The resulting address is then evaluated inside an address comparator that will activate a module select for the slave in step 4. Within the same step, the address data will be aligned by a demultiplexer according to the start position of the selected slave. With respect to these two signals, the selected slave will drive its data onto the bus in step 5. Finally, in step 6, the data is transferred to the master. If the data signals are interleaved, it requires an alignment multiplexer for adjusting the slave data as well as an alignment demultiplexer for ordering the data to the master. The figure does not show the alignment (de) multiplexer look-up tables.

to internal fragmentation and because of the communication cost - is minimal for resource slots that provide just a few hundred look-up tables (depending on the communication cost as shown in Figure 3.3). Consequently, partial runtime reconfiguration includes a considerable logic overhead if complete logic columns on larger FPGAs are exchanged. This is the case in many systems [MBV+02, MTAB07, SBFP03] that are based on Xilinx Virtex-II FPGAs. For instance, a Xilinx Virtex-II 6000 device provides 768 LUTs per logic column (CLB column) and the best reported systems have been able to reduce the resource slot width to four columns, hence providing about one order of magnitude more LUTs than the optimal value.
Apart from the logic, FPGAs further provide dedicated resources, such as multipliers and dedicated memory. On FPGAs from Xilinx or Altera, these special resources are located in columns, spanning the complete height of the FPGA conjoined with the logic columns. The dedicated resources are very valuable and should therefore be highly utilized by the system. For example, an 18 by 18 bit multiplier would require nearly 400 LUTs for an implementation with the help of logic tiles. And a single block RAM can substitute more than 2000 LUTs (also in the case that the LUTs are used as distributed memory) on Xilinx Virtex-II or Virtex-IV FPGAs.

In [BLC07] it is proposed to pay this price for enhancing module relocatability. The idea is omitting the usage of multipliers or BRAMs such that the same module (equal to the same configuration bitstream) can be placed to different resource slots that provide once a BRAM column or once a multiplier column at the same relative position (proposed for Virtex-IV devices). The goal of this approach is to reduce the number of configuration bitstreams. However, this is bought by a poor device utilization and it must be mentioned that in some cases two individual modules - one for the BRAM resource slot and one for the multiplier slot - require the same amount of memory in the bitstream repository as the relocatable one. In other words, modules should also achieve high device utilization in the case of specialized resources provided by the FPGA fabric.

The only way to tackle these issues is to tile the reconfigurable area into a two-dimensional resource slot grid. This holds many advantages including that multiple modules can share the same multiplier column or memory column and that bounding boxes can be better adjusted to fulfill different requirements with respect to the logic or some dedicated resources as exemplary shown in Figure 3.38. A further benefit is that the FPGA I/O pins can be used more flexible as it is possible to assign logic and I/O pins above or below the reconfigurable area. The main penalty of a two-dimensional placement is a rather complicated resource and configuration management as revealed in Section 4.2.

The ReCoBus interleaving scheme, where the module interface can grow in relation to the module size, can be extended for two-dimensional systems as illustrated in Figure 3.39 for shared read signals as well. Each row of resource slots is interleaved, as revealed in Section 3.5 for the one-dimensional case. The interleaving scheme in each new row is horizontally shifted one position further as compared to the preceding row, hence interleaving the bus also in vertical direction. Note that all rows are exactly the same and that the vertical interleaving is achieved by cyclically rotating the connections in the static part of the system. The bus does not require any vertical connections between resource slots, and the logic overhead of the resource slots is the same as in the one-dimensional case. Also, the alignment multiplexer is the same as in the one-dimensional case. The only additional logic overhead is a set of OR-gates for implementing shared read signals. In the case of shared write signals, another set of unidirectional wires will be used in opposite direction as the one shown in Figure 3.39, hence the alignment multiplexer will become a demultiplexer and the
OR-gates will be removed. Consequently, the resource overhead for shared write signals is the same for the one-dimensional and for the two-dimensional case. The same holds true for dedicated write signals, because all internal enable signals for the reconfigurable select generators within one row are connected to the same source within the static part of the system without any further logic.

The last signal class, required for implementing buses suitable for a two-dimensional placement of reconfigurable modules, are dedicated read signals. As two-dimensional reconfigurable systems require advanced configuration bitstream manipulations when implemented on Xilinx Virtex-II FPGAs (see also Section 4.2), the dedicated read signal distribution that is based on the manipulation of the configuration bitstream is most suitable. In this case, the techniques proposed for one-dimensional slot based systems, depicted in Figure 3.35, can be duplicated for each row of resource slots. As it is possible to share the connecting resources with other bus logic, this approach does not include additional logic cost within the resource slots.

It can be summarized that the interleaving technique can also be efficiently applied to systems that allow placing modules in a two-dimensional resource slot grid. Using a grid style configuration instead of a one-dimensional slot arrangement will not increase the logic overhead per resource slot. This allows reducing the size of a resource slot in general and enhancing the utilization of the dedicated resources, like, for example, dedicated memory blocks.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

3.6 Point-to-Point Communication

Besides buses for directly accessing registers inside a reconfigurable module (slave mode) or for addressing system memory by a reconfigurable module (master mode), a reconfigurable module may require some further signals to carry out the communication with the environment. These could be other reconfigurable modules, I/O pins, or some dedicated modules in the static part of the system. The module environment communication should preserve the flexibility provided by a ReCoBus, such as a fine resource slot grid or module relocation.

By analyzing related work on point-to-point communication or circuit switching for runtime reconfigurable systems, it was observed that in all presented systems the channels are established directly after exchanging or loading a reconfigurable mod-

Figure 3.39: Interleaved ReCoBus (only shared read data) implementation for two-dimensional module placement. All modules provide 32-bit access via four resource slots that are vertically (m1), horizontally (m2), or in an 'L'-shape (m3 and m4) aligned. The highlighted top left resource slot is the start resource slot, connecting the most significant byte within a reconfigurable module. The numbers in the resource slots denote the multiplexer setting for the alignment multiplexer if the slot is the start point of a reconfigurable module. The value is the sum of the x and y start point coordinates modulo the interleaving factor. For example, if module m4 is selected, the alignment multiplexers must be set to position \((5 + 2) \mod 4 = 3\).
3.6 Point-to-Point Communication

ule [BHB+07]. These channels will then be retained until the module terminates or if it is replaced. Therefore, there is only switching activity in the point-to-point terminals upon a module swap with the help of partial reconfiguration. As reconfiguration is typically performed rarely, the multiplexers in the point-to-point terminals should be directly implemented within the switch matrix resources of the FPGA fabric, rather than in the look-up-table resources. Despite the fact that the FPGA look-up tables are basically multiplexers (with SRAM cells at the inputs and the select inputs being connected to the routing fabric of the FPGA), look-up tables are not very efficient for implementing wide input multiplexers as listed in Table 3.2.

Table 3.2: Relationship between the look-up table count and the input width of a multiplexer implemented on a Xilinx Spartan-3 or Virtex-II FPGA.

<table>
<thead>
<tr>
<th>Mux inputs</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit LUTs</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

For example, a simple one-dimensional RMB-like (see also page 95) circuit switching architecture with two horizontal channels requires four multiplexers for the left and right channel outputs that have three inputs each. This allows selecting between one of the two preceding channels or the module output, as illustrated in Figure 3.40. For this example, a switching element requires 4 times two 4-bit look-up tables, which fits exactly into one CLB on a Xilinx Virtex-II FPGA. This is the logic overhead per bit within a channel. The total logic overhead in CLBs within the reconfigurable area is therefore the overhead per switched bit, multiplied by the channel width, multiplied by the amount of switching elements (equal to the number of resource slots). For instance, the simple example with two times two horizontal channels requires for 50 slots and 16 bit wide channels 800 CLBs on a Virtex-II FPGA. This is 9.5% of a Virtex-II-6000 device if implemented within the LUTs of the FPGA fabric or two to three times the amount of logic required for implementing a 32 bit softcore CPU. Note that this is pure overhead that originates only from using runtime reconfiguration. And this overhead contributes to more overhead resulting from the internal fragmentation, for the bus communication, and for the reconfiguration interface, hence, making it difficult to achieve an overall benefit by using partial runtime reconfiguration.

In [GBS+08], a concept for implementing the switching multiplexers with look-up tables that are configured in route through mode is proposed, which requires about half of the amount of LUTs when compared to the values listed in Table 3.2. The idea of this concept is the same as the one used for the switching elements for dedicated read wires (see Figure 3.34): the multiplexer inputs are routed to the inputs of a look-up table and the LUT function is the identity for the selected input. The multiplexer setting is exchanged by modifying the LUT function and not by changing the state.
3. **Intra-FPGA Communication Architectures for Reconfigurable Systems**

![Figure 3.40](image)

**Figure 3.40:** a) Two simple circuit switching elements implemented with multiplexers (LUTs) and b) alternative implementation directly with the help of the FPGA switch matrices. For the latter case, only the important possible switch settings are shown that are selected by a multiplexer within the FPGA routing fabric. Direct implementation of circuit switching inside the switch matrices allows using the LUT resources by the reconfigurable modules. The module output is marked by dashed lines.

of some select signals. In this approach, the multiplexer state, determining which particular input is connected to the output, is stored in the internal look-up table SRAM cells (the LUT function table) and not within some user flip-flops. This allows reducing the overhead against classical multiplexer implementations to roughly one half, which will still be substantial in many systems with a fine-grained resource slot grid. For example, a 4-input LUT can directly implement a 4:1 multiplexer when changing the LUT configuration according to the selected input. When implementing the same 4:1 multiplexer in the common technique, two 4-input LUTs are required (see also Table 3.2).

Alternatively, it is proposed to carry out the switching directly within the switch matrices of the FPGA fabric that are basically reconfigurable multiplexers. This is depicted in Figure 3.40b) for the simple two channel one-dimensional circuit example discussed above. The figure only shows the FPGA routing resources relevant for implementing the switching elements. However, there are plenty of additional wiring resources available on typical FPGA architectures, thus only a relatively small amount of the available routing resources are required for implementing the switching elements. According to the Xilinx Virtex-II architecture, there exist 20 outgoing wires (called hex lines and double lines) towards each horizontal direction as plotted in Figure 3.6. Consequently, a switching element of the assumed example would occupy only 10% of the horizontal routing resources while leaving all look-up tables and the remaining routing resources for implementing the reconfigurable modules.

The following two sections will discuss how point-to-point links can be implemented between a reconfigurable module and the environment, directly within an FPGA routing fabric. In Section 3.6.1, a one-dimensional circuit switching architecture will be proposed that is suitable for a fine resource slot grid and wide data
3.6 Point-to-Point Communication

channels. An architecture for switching dedicated signals in a two-dimensional fashion for reconfigurable systems on an FPGA will be presented in Section 3.6.2.

3.6.1 One-dimensional Online Circuit Switching

In Section 3.6 it was assumed that all channels between the circuit switching elements are equal and that each channel may be able to transfer any data from or to a reconfigurable module. In related approaches [ABD⁺05, UHB04] this model was used and it requires full crossbar switches for relatively wide data paths. Regardless of whether the multiplexers of the switching elements are implemented with the help of look-up tables or directly within the FPGA switch matrices, wide input multiplexers are inefficient to implement in both cases. Consequently, a suitable circuit switching architecture for runtime reconfigurable systems should minimize the amount and the size of the multiplexers inside the circuit switching elements.

Circuit switching is used for switching a channel from one source to one (point-to-point) or more (multicast) destinations. The channel must match the entire application and different channels with different requirements may be used within one system. For example, a multimedia system may require a video channel and an independent audio channel. While a video channel typically requires a throughput of some tens to a few hundreds of megabytes per second, an audio channel demands just 1.4 Mbit in CD quality. Consequently, diversely adjusted channels for the independent data streams are in most cases much more efficient to implement than using a homogeneous communication architecture with identical channels for any type of stream. Another reason for using independent channels is that the switching elements require a lower degree of freedom, hence, simplifying online routing algorithms. Finally, a system should always provide feasible switching for all allowed placement positions of the reconfigurable modules. In most applications, it cannot be accepted to reject a module or a service because of insufficient online communication resources for integrating a partial module into the system.

These observations have been taken into account for the design of so-called connection bars that feature an adjustable set of wires, which create switched connections from resource slot to resource slot in a homogeneous manner. In order to allow a module to connect to some resources outside its boundary, multiple adjusted connection bars are allocated at design time along all resource slots. Within each resource slot, a bar may be tapped to receive a down stream or ripped for driving an upstream towards the bar. Note that in the latter case, the module may still receive a downstream, hence, allowing to receive data, process it, and send it to some other modules or the static part of the system further along the bar in a read-manipulate-write manner.

Figure 3.41 presents an example of such a system, where three modules communicate with the environment using the connection bar approach. A bar can utilize any arbitrary amount of input or output wires. All resource slots that do not write
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

A bar have to pass through all internal bar wires. Regardless of whether a bar is accessed, the input and output wires have to be equivalent in all resource slots and for all possible settings of the connection bar switching elements. Consequently, modules can be placed freely to the resource slots while still being able to communicate with dedicated resources or modules outside the module bounding box. The communication bar approach does not require any online routing algorithm and the switching elements are part of the module and can be automatically set during module configuration. This is possible by selectively instantiating connection bars that pass through a set of wires or provide access to these. The entire bar instance is related to a particular module. For example, a video module will access the video connection bar with an associated connection bar containing connection primitives, while passing through the bar for audio data. As an alternative for setting the connection bar switching primitives directly during module reconfiguration, the switching element setting can be pushed before or after the entire module configuration. For example, if a video filter is exchanged by another, the associated video bar may be firstly configured in feed-through mode for masking disturbances related to the reconfiguration process from the video stream. Later, after writing the configuration of the new module to the FPGA, the connection bar switch may be changed from feed through to read-manipulate-write operation.

The connection bar approach is ideal for many applications processing data streams. The same set of horizontal wires of a connection bar can be segmented into multiple point-to-point links or for streaming data from one source to multiple destinations or for a mixture of both. This matches a wide range of networking or multimedia processing systems. A video processing system, for example, may send the video stream to a reconfigurable filter module first, from where the preprocessed data can be distributed to multiple parallel working pattern recognition modules. In this example, the connection bar is segmented in the preprocessing module, allowing a point-to-point link from the video source and, at the same time, a multicast link to the recognition modules, using the same horizontal connection bar. The only restriction in such a system is that the preprocessing module must be placed before the recognition modules in the data stream. Note that a connection bar can be alternatively used for providing connections between both sides of a static system that is located at the left and right neighborhood of the reconfigurable area.

The latency of a connection bar becomes large in systems with many resource slots. For reducing the combinatory path of a connection bar, the internal wires can be interleaved, as discussed for the bus architecture in Section 3.5.1. By interleaving a connection bar, the $S_C$ signals of a particular bar are distributed via $N$ wire subsets, containing $\lceil \frac{S_C}{N} \rceil$ wires each. The wire subsets provide a subset of switching elements after $N$ resource slots and each subset is displaced against the preceding subset by one resource slot position. Hence, each resource slot is connected to a subset of $\lceil \frac{S_C}{N} \rceil$ internal bar wires. As the wire subsets skip $N - 1$ resource slots, routing may be carried out faster. For example, when implementing a connection bar on a Xilinx Virtex-II

150
3.6 Point-to-Point Communication

Figure 3.41: Example of a system with an audio module and two video modules linked together by a ReCoBus and three connection bars. The top bar allows the video filter module to connect to on-chip memory located outside the reconfigurable area, while the other connection bars are used to connect the audio and video streams to the I/O pins. A module can pass through the signals of a particular connection bar (box a), access these signals in a read-manipulate-write fashion (box b), or tap the stream without manipulation (box c). The settings of the switching elements inside the resource slots are part of the FPGA routing fabric and controlled by configuration of SRAM cells.

XC2V6000-6 FPGA with double lines (a wire spanning a distance of two basic elements), the propagation delay for three lines in a row is $3 \cdot 0.282 \text{ ns} = 0.846 \text{ ns}$. Interconnecting the same distance with interleaved hexlines allows the same resource slot grid, while reducing the propagation delay on the same routing distance to 0.356 ns or 42%. Note that interleaving allows to use routing wires, spanning over a wider distance than the width $W$ of a particular resource slot. For full connection bar access, a module must be at least $N$ consecutive resource slots wide.

Free module placement in systems using interleaved connection bars requires a set $S_C$ alignment multiplexers (each possessing $N$ inputs) once per module, if the data is streamed to the module (and not bypassing it). In case the module sends data to the connection bar, $S_C$ alignment demultiplexers are necessary. There is no further alignment required in the static part of the system. On Xilinx Virtex-II or Spartan-3 FPGAs, the alignment multiplexer can be directly implemented within a CLB switch matrix. For example, a system with one CLB column wide resource slots ($W = 1$) that is twofold interleaved ($N = 2$), will use double lines for implementing connection bars. Such double lines span a distance of two CLBs and can be tapped in the middle,
after one CLB. The same holds respectively true for hex lines that connect to CLBs, located three and six columns away from the source CLB. The end points and the middle connections of these routing resources are connected to the same configurable input multiplexer within a CLB switch matrix. This allows switching between the first and the second interleaved wire subset by changing the associated switch matrix multiplexers between end point or the middle connection. More details on online bitstream manipulation are revealed in Section 4.2.

### 3.6.2 Two-dimensional Online Circuit Switching

Connection bars are very efficient for distributing wide circuit switching channels in a one-dimensional fashion at low latency. Multiple connection bars located in different rows of resource slots aligned in a grid style can also be used for a two-dimensional placement. As it will be shown in Section 3.7.3, many hundreds of individual horizontal wires of the FPGA fabric can be allocated for implementing a communication architecture capable to integrate reconfigurable modules before dropping connectability or the achievable clock frequency. In the proposed resource slot based reconfigurable systems (tailored to Xilinx FPGAs), this only holds for horizontal routing resources, because modules typically possess a narrow but high form factor. Therefore, many horizontal wires are not usable for implementing the reconfigurable modules, as these resources can be driven within the module boundary but only sensed outside the bounding box. In other words, the required modular design style of strictly forcing modules into bounding boxes will disallow using some interconnect resources at the module boundaries. Employing these resources in order to provide a communication architecture for integrating partial modules into the runtime system comes for free. However, any vertical routing between the different multiple horizontal communication bars must be performed within the static part of the system.

As an alternative to two-dimensional online circuit switching with multiple connection bars, a mesh topology for free two dimensional online routing was examined. Such an approach is only suitable for narrow channels with very few wires, because almost all vertical wires are used for implementing the interconnection of a partial module, hence, conflicting with the online routing architecture. On current FPGAs, this is possible with a reasonable performance. A Xilinx Virtex-II 6000-6 FPGA, for example, allows to establish a signal path composed of up to 35 double lines (or 26 hex lines) with a 10 ns latency budget. This equates to a Manhattan distance of 70 (156) CLBs or 73% (163%) of the device height or width. A single wire channel at this speed is sufficient for Fast Ethernet, 70 CD audio streams or 41 uncompressed CIF (Common Intermediate Format) video frames per second (24 bit RGB).

For free two-dimensional online circuit switching, an interconnect network is allocated logically on top of the FPGA fabric, whereas the switching elements are allocated within the switch matrices. This can be implemented by blocking a set of...
3.6 Point-to-Point Communication

wires, for example, one for each direction (north, east, south, west) within each ba-
sic element (see also Section 4.1.2) of the reconfigurable area that will be used for performing the circuit switching. These wires are permitted for implementing the re-
configurable modules or any other hardware of the static system. The blocked wires will represent a separate interconnect layer for module-to-module and module-to-
environment communication. Using disjoint interconnect resources for the module implementation and the online circuit switching network, routing can be performed regardless of the module placement positions, and therefore, the modules are not ob-
stacles for the circuit switching. The blocked wires are directly linked together by the routing resources supplied by the BE switch matrices of the FPGA, thus, not requiring any extra logic. Only at the source and the destination(s) of a connection bar there may be some logic required for connecting the path with the modules. The blocked wires should be chosen in such a way that they form a regular two-dimensional rout-
ing grid. In each cell of the grid it should be possible to route strait through, or to turn left or right. In addition, an incoming signal may be connected to multiple outputs of a cell for allowing multicast operations on the same path.

Figure 3.42: Examples of some switch positions of a routing cell that is obtained by blocking one routing wire in each direction (north, east, south, west) of the device. When each output can be connected with three different inputs by partial reconfiguration, there exist $3^4 = 81$ different switch permutations of such a routing cell.

Figure 3.42 depicts an example of how two-dimensional circuit switching could be implemented directly within the switch matrices of an FPGA. For supporting this approach efficiently, a set of wires must be chosen that allows a replicated routing by directly configurable connections from the input wires of a cell to the allocated output wires, without using additional resources. A replicated routing scheme allows a free rectangular circuit switching on the device by using the same equivalent routing resources in each routing cell. For example, when reserving one dedicated wire in all four directions for providing a two-dimensional online circuit switching network on an FPGA, then it should be possible, to establish any path within the network. This implies that it must be possible to set a zigzag path (like towards east then towards north and in east direction again), such that only the reserved wires are used without additional wiring resources. Most common, this is not possible in a trivial manner, because the approach requires that connections can be arbitrarily set. In general, this is not possible, because FPGA switch matrices cannot provide full crossbar switches.
3. **Intra-FPGA Communication Architectures for Reconfigurable Systems**

Consequently, online circuit switching is possible, if the allocated wires fulfill the following three properties:

1) incoming wires must be extendable to the same corresponding channel index for allowing straight forward routing,

2) there must exist a cycle with four hops in clockwise direction for turning to the right, and

3) a cycle with four hops in anticlockwise direction for turning to the left.

The switch matrix settings of the circuit switching network can be computed with standard algorithms such as Dijkstra [Dij59], Belman-Ford, or A* [Tes98]. Congestion can be prevented with iterative techniques such as proposed for VPR [BR97] or Pathfinder [ME95] where cost functions are adapted over the iterations such that congested paths become more expensive in order to force the search algorithms to use alternative resources.

**FPGA Limitations**

Implementing two dimensional online circuit switching on existing FPGAs may restrict the communication architecture because of an insufficient degree of routing freedom or because of some inhomogeneous resources found in the otherwise regular FPGA fabric. With respect to the amount of wires connected to an FPGA basic element, there can only exist a relatively small subset of all possible connections within a configurable switch matrix, as fully connected crossbars would be too expensive. Therefore, the approach presented in Figure 3.42 may not be applicable to all available FPGA architectures. In Figure 3.43, the connectivity of a switch matrix is exemplary illustrated by an adjacency matrix in form of a bitmap for a Xilinx Virtex-II FPGA. A black dot denotes a possible connection between a switch matrix input and an output. While a fully connected crossbar would lead to one black plane in the figure, the Virtex-II FPGA adjacency matrix is sparely populated and the routing multiplexers inside the switch matrix are connected to just 6.25% of the available inputs, in average.

By examining the Xilinx Virtex-II switch matrices in more detail, no set of routing wires can be found, allowing to directly implement the approach in Figure 3.42, because of not all of the above mentioned properties can be fulfilled at the same time. While property 1) for straight routing tracks can always be met on Virtex-II FPGAs, only one further property can be directly implemented for making a turn in the routing path. A replicated routing scheme, however, can also be obtained, if some connections between the set of blocked wires is not available. For example, if there is no possibility to turn left at a routing cell, it is feasible to implement a loop by routing three times in turning right direction. This leads to a *clockwise* or in turn to an
3.6 Point-to-Point Communication

Figure 3.43: Adjacency matrix of a Xilinx Virtex-II CLB switch matrix, illustrated by a bitmap graphic. Each black dot represents a possible configuration from one source (column) to one particular sink (row). Between the 327 inputs and the 157 outputs of a CLB switch matrix, there exist altogether 3210 configurable connections, which is 6.25% of a full crossbar switch. A row is a reconfigurable multiplexer and the amount of ones (dots) within a row indicates the number of multiplexer inputs.

anticlockwise routing scheme. Examples for these routing schemes tailored to Xilinx Virtex-II FPGAs (also suitable for Virtex-IV, or Spartan-3 FPGAs) are illustrated in Figure 3.44. Here, each output wire of the network can be directly connected with a wire, starting in a straight preceding switch matrix or with a wire starting in one bended cell. Therefore, each wire has two switch positions and a complete cell that is providing one signal per direction can be described with just four bits. The two switch positions are set by configuring the output multiplexer for a particular wire with one out of the two corresponding configuration encodings.

Apart from the routing freedom, the online circuit switching architecture should further be capable to respect inhomogeneous FPGA resources, such as dedicated memories or multipliers. Furthermore, the architecture and the resource slots should be selected with respect to the available routing resources of the FPGA fabric. As depicted in Figure 3.45, the resource slot dimensions should be set to a common multiple of the distance of the used wire resources and the size of possibly existing, dedicated FPGA resources. With these restrictions, modules can be freely placed in a two dimensional fashion while being able to carry out the communication via a regular mesh type network at runtime.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.44: Two-dimensional on-line circuit switching scheme tailored to Xilinx Virtex-II, Virtex-IV, or Spartan-3 FPGA. On these devices it is possible to arbitrarily route within a switch matrix straight ahead and either turning a) clockwise or b) anticlockwise directly by allocating one wire per direction (north, east, south, and west) but without any additional routing resources. a) and b) are exclusive as there exists no set of wires that allows directly implementing a zig-zag path, like turning right, then left, and right again.

Figure 3.45: Example of partitioning an FPGA into resource slots. In all examples, (vertical) online circuit switching is carried out via one fat drawn wire of a channel of span $S = 3$. a) A height equal to the span, $H = S$, allows accessing the online circuit switching architecture at the same relative position inside each resource slot but may prevent the usage of dedicated resources (here the memory blocks). b) Selecting the resource slot height to a multiple of the memory height provides full memory utilization but may restrict access to the circuit switching architecture. c) By setting the resource slot height to a common multiple of the memory height and the span, it is possible to use the memories as well as to access the online circuit switching network at exactly the same relative position within the resource slots.
3.7 Experimental Results

For benchmarking the proposed ReCoBus and connection bar communication architecture, several tests have been performed that are organized in the following sections:

Section 3.7.1: Measuring the timing and performance characteristics of the ReCoBus architecture.

Section 3.7.2 Examining the logic cost for implementing the communication architecture.

Section 3.7.3: Evaluation of the impact of restricting the module routing on the achievable clock frequency and on the capability to fully route a module.

Section 3.7.4: Investigation of transient effects on the communication architecture due to a reconfiguration process.

Section 3.7.5 Implementation example of a two-dimensional communication architecture featuring ReCoBus and connection bar connections, as well as a two-dimensional circuit switching network.

During these experiments, it was found that the original ReCoBus and connection bar architecture may be interrupted during the complete reconfiguration process of a module when implementing reconfigurable systems on Xilinx Spartan-3 FPGAs. Therefore, Section 3.7.4 presents techniques to circumvent these interruptions such that the communication architecture can also be implemented on these FPGAs.

3.7.1 ReCoBus Timing Characteristics

The maximum throughput $T$ of a ReCoBus is given by the data word width $B$ of the bus, multiplied by the clock frequency $f$ that can be achieved in the system ($T = B \cdot f$) and measured in the unit bits/second. Here it is assumed that a new bus transfer can start at any clock cycle and that the bus does not use multi-cycle paths in order to reach higher clock rates by the cost of multiple clock cycles per transfer. While $B$ is typically set to the word width of a processor or memory, the value for the maximum clock frequency $f$ is related to the reciprocal of the propagation delay of the critical path $t_{prop} = \frac{1}{f}$ to evaluate the combinatorial logic. For simplicity reasons, setup and hold times of the flip-flops will be omitted in the following. With respect to the proposed ReCoBus architecture, the critical path determined on data read operations, in the case of a bus containing only slaves. Depending on the used features, the critical path consists of a forward path with the module select logic (see Figure 3.23) and a backward path consisting of a distributed read multiplexer followed by a final multiplexer for aligning the correct byte order (see Figure 3.29). A timing model for this path can be specified based on the bus architecture presented in
3. **Intra-FPGA Communication Architectures for Reconfigurable Systems**

Figure 3.46: Critical path of a transaction of a ReCoBus. Data read operations have the longest critical path consisting of a forward path including a reconfigurable select generator and a backward path containing a distributed read multiplexer. The path is measured between the master connection terminals driving the address signals and receiving the data through the final multiplexer. The example follows the presented approach in Figure 3.29 with four interleaved distributed logic-based bus chains ($N = 4$).

Figure 3.46 that is independent of the final FPGA implementation. The architecture is mainly parameterized by the resource slot width $W$ and the number of interleaved read multiplexer chains $N$. Both values are given in terms of basic elements found on the reconfigurable target architecture (see also Section 3.3 for details on the assumed FPGA model). Then, the annotated delays in the architectural view have the following meanings:

- $t_{BE1}$ is the time required for the most critical address signal to pass a connecting resource (e.g., a look-up table) plus the propagation delay for the wiring of this address signal to the reconfigurable select generator, having the maximum signal propagation delay within the first $N$ module connection points. These module connection points are located in the first $N$ resource slots, located next to the static part of the system. In the case that the address signals are interleaved, the propagation delay of an additional alignment demultiplexer has to be considered. Note that all available reconfigurable select generators are connected to the address wires in order to take the delay for an increased fan-out into account.

- $t_{BE2}$ specifies the signal propagation delay for routing the most critical address signal to the next chained module connection point that is $N$ resource slots further away.

- $t_{EN}$ gives the propagation delay for the reconfigurable select generator (see Figure 3.22) plus the chunk of a distributed read multiplexer that feeds in the module data to the bus. The read enable signal ($read\_en$ in Figure 3.22) is con-
connected to \([B/N]\) chunks of a distributed read multiplexer, because it requires \(N\) resource slots to connect \(B\) individual data signals to the bus. The value of \(t_{EN}\) represents the most critical propagation delay over all \([B/N]\) paths.

\(t_{DRM}\) specifies the time to pass the look-up table of a distributed read multiplexer stage.

\(t_{D2D}\) denotes the propagation delay on the wiring between two chunks of a distributed read multiplexer.

\(t_{D2M}\) denotes the propagation delay on the wiring between the last chunk of a distributed read multiplexer chain and the multiplexers for adjusting the alignment of the \(N\) subwords. The value gives the largest delay over all \(B\) data lines.

\(t_{MUX}\) specifies the time for the alignment multiplexers. These multiplexers have \(N\) inputs and ensure that the data signals are aligned in the same order, regardless of the position of a module at the bus.

These time values allow extrapolating the propagation delay to any size of the bus. The combined worst case propagation delay of the bus is calculated as follows:

\[
t_{\text{prop}} = t_{BE1} + t_{EN} + t_{D2M} + t_{MUX} + \lambda \cdot (t_{BE2} + t_{DRM} + t_{D2D})
\]

(3.10)

The factor \(\lambda\) specifies the width in addition to the first \(N\) resource slots of the bus as a multiple of the number of interleaved distributed read multiplexer chains \(N\). As shown in Equation (3.10), the worst case propagation delay consists of a fixed and a variable part that can be used to extrapolate small reconfigurable buses to any size. The complete reconfigurable bus offers \(R = (\lambda + 1) \cdot N\) resource slots, each one with an interface width of up to \([B/N]\) bits. With this timing model, the design space of a particular ReCoBus implementation can be accurately predicted without the need to fully implement the whole bus architecture on a particular FPGA.

As presented in Section 3.3, FPGAs offer routing resources, spanning over different numbers of basic elements that can be combined to build different regular bus structures. The resource slot width \(W\) may be chosen to a multiple of the grid of the basic building blocks of the respective FPGA architecture. Let \(H\) be the height, counted in basic blocks then the total amount of logic resources, a reconfigurable bus can provide to some modules, is \((\lambda + 1) \cdot N \cdot W \cdot H\) basic blocks.

For determining the impact of combining different wiring resources and resource slot widths, multiple variants of the bus architecture shown in Figure 3.46 have been implemented on different FPGA devices. The experiments were carried out for different FPGAs from Xilinx Inc. that are all partially reconfigurable. As mentioned before in Section 3.3, a basic element of these FPGAs is called a CLB. And all examined FPGAs offer the same two different wire spans for building the local module...
Table 3.3: Measured propagation delays for the proposed timing model. All values are in nano seconds. The table lists results for one Xilinx Virtex-II FPGA, two Virtex-IV devices of different speed grade, and a Spartan-3 FPGA. The values have been determined by a timing analysis of the different bus implementations.

<table>
<thead>
<tr>
<th>FPGA variant</th>
<th>t_{BE1}</th>
<th>t_{BE2}</th>
<th>t_{EN}</th>
<th>t_{DRM}</th>
<th>t_{MUX}</th>
<th>t_{D2D}</th>
<th>t_{D2M}</th>
</tr>
</thead>
<tbody>
<tr>
<td>V-IV-6 double+double</td>
<td>1.095</td>
<td>0.486</td>
<td>1.988</td>
<td>0.347</td>
<td>0.92</td>
<td>0.639</td>
<td>1.148</td>
</tr>
<tr>
<td>hex-double</td>
<td>1.095</td>
<td>0.486</td>
<td>1.988</td>
<td>0.347</td>
<td>0.92</td>
<td>0.706</td>
<td>1.082</td>
</tr>
<tr>
<td>hex+double</td>
<td>1.581</td>
<td>0.972</td>
<td>1.988</td>
<td>0.347</td>
<td>0.92</td>
<td>0.706</td>
<td>1.179</td>
</tr>
<tr>
<td>V-IV-10 double+double</td>
<td>1.178</td>
<td>0.445</td>
<td>1.46</td>
<td>0.195</td>
<td>0.779</td>
<td>0.581</td>
<td>1.025</td>
</tr>
<tr>
<td>hex-double</td>
<td>1.178</td>
<td>0.445</td>
<td>1.46</td>
<td>0.195</td>
<td>0.779</td>
<td>0.614</td>
<td>0.83</td>
</tr>
<tr>
<td>hex+double</td>
<td>1.673</td>
<td>0.89</td>
<td>1.46</td>
<td>0.195</td>
<td>0.779</td>
<td>0.624</td>
<td>1.345</td>
</tr>
<tr>
<td>V-IV-11 double+double</td>
<td>1.828</td>
<td>0.374</td>
<td>1.244</td>
<td>0.166</td>
<td>0.664</td>
<td>0.492</td>
<td>0.867</td>
</tr>
<tr>
<td>hex-double</td>
<td>0.997</td>
<td>0.374</td>
<td>1.244</td>
<td>0.166</td>
<td>0.664</td>
<td>0.517</td>
<td>0.702</td>
</tr>
<tr>
<td>hex+double</td>
<td>1.415</td>
<td>0.748</td>
<td>1.244</td>
<td>0.166</td>
<td>0.664</td>
<td>0.525</td>
<td>1.137</td>
</tr>
<tr>
<td>S-3-4 double+double</td>
<td>1.637</td>
<td>0.335</td>
<td>1.239</td>
<td>0.608</td>
<td>1.793</td>
<td>0.955</td>
<td>1.376</td>
</tr>
<tr>
<td>hex-double</td>
<td>1.637</td>
<td>0.335</td>
<td>1.239</td>
<td>0.608</td>
<td>1.793</td>
<td>0.927</td>
<td>1.376</td>
</tr>
<tr>
<td>hex+double</td>
<td>1.972</td>
<td>0.67</td>
<td>1.239</td>
<td>0.608</td>
<td>1.793</td>
<td>0.927</td>
<td>2.305</td>
</tr>
</tbody>
</table>

interconnect. Double-lines connect a CLB with another one that is two CLBs far away. Analogously, Hex-lines connect a CLB with another one that is six CLBs far away. For all cases, the interleaving factor was set to $N = 4$ to match the interface size of a $B=32$ bit data bus by using 8 look-up tables per resource slot. For a resource slot width of $W=1$, two cascaded wiring schemes have been manually implemented. In a first variant, two cascaded double wires (named double+double) have been used for routing a distance of $W \cdot N = 4$ CLBs. In another assembly, one hex wire in forward direction was cascaded with one additional double wire in backward direction (named hex–double). For a ReCoBus, featuring a resource slot width of $W=2$ CLBs, one hex wire and one double wire, both cascaded in forward direction, have been selected (named hex+double). This leads to the required routing distance of 4 \cdot 2=8 CLBs. The resulting timing values are listed in Table 3.3.

According to Equation (3.10), the measured propagation delays have been extrapolated to larger bus sizes. The results are visualized in Figure 3.47. It can be seen that the double+double approach leads to similar results as the hex-double one for all examined FPGAs. By defining a specific time budget for the bus propagation delay, and consequently defining the desired clock frequency of the bus, the possible parameters for the complete bus width $(\lambda + 1) \cdot N \cdot W$ and the resource slot width $W$ of the ReCoBus can be determined. For example, if the specification demands a clock frequency of 50MHz ($t_{prop} = 20\text{ns}$) on a Spartan3 -4 FPGA, a ReCoBus can be
3.7 Experimental Results

**Figure 3.47:** Extrapolated propagation delays for different FPGAs and different cascading schemes over the number of attached interleaved resource slots $\lambda$. The resource slot width is $W = 2$ for the hex+double cascading scheme and $W = 1$ for the other schemes.

implemented that is up to $(5 + 1) \cdot 4 \cdot 2 = 48$ CLB columns wide. This bus would be organized in 24 resource slots, each of them two CLB columns wide. Alternatively, $(7 + 1) \cdot 4 \cdot 1 = 32$ CLB columns, organized in 32 resource slots, would allow higher flexibility in smaller systems. Note that these values are best case values. Additional propagation delay may be added by some of the reconfigurable modules or the static part of the system.

So far, a discussion about heterogeneities found in the FPGA architecture was omitted. Such a heterogeneity can be a memory column or columns with dedicated IP blocks, like multipliers. For example, on Xilinx Virtex-II FPGAs a RAM column is provided in the range of every 4 to 25 CLB columns. For all investigated devices, the additional propagation delay, found for bypassing a RAM column, was less than the delay required to bypass a CLB logic column. In addition, all RAM/multiplier columns provide the same routing resources as the logic columns. This allows to feed signals through heterogeneous columns without losing the regular interconnect structure of a ReCoBus. As a consequence, a RAM/multiplier column can be treated like any normal logic column with the exception that it cannot provide interface logic to a connected module.

Figure 3.47 points out that high system frequencies (with a 100 MHz or more) are only allowed for relatively narrow buses. Note that also static designs require ad-
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

traditional effort for the implementation of high speed systems. In order to overcome this limitation, pipelining the bus communication was examined. This still allows starting a communication transaction at any clock cycle, but data will arrive with a latency of one additional clock cycle at the output, while potentially transferring a new data word each cycle. Larger portions of data can be sent over the bus as one continuous transfer by using a burst protocol. Alternatively, split transaction protocols may be used that allow overlapping multiple bus requests in time. Both protocols are supported by the embedded hard IP processors (a PowerPC-405 Core) available on Xilinx Virtex2-Pro FPGAs. On these cores, a sequence of multiple bus transfers to different addresses can be started before the first transfer has been finished. It is preferable that the latency of the reconfigurable bus is independent of the placement position of a module. Therefore, pipelining is not suitable within a distributed read multiplexer. By analyzing the timing model in Figure 3.46, it was determined to place a pipeline register between the forward path and the backward path, containing the distributed read multiplexer chains. In this case there exists the choice to place the pipeline register after or before the block, containing the reconfigurable select generator. As a consequence, $t_{EN}$ will be considered either in the forward path or in the backward path. The worst case propagation delay of the reconfigurable bus with one pipeline register is then:

$$
t_{BE} = t_{BE1} + \lambda \cdot t_{BE2} \\
t_{RM} = t_{D2M} + t_{MUX} + \lambda \cdot (t_{DRM} + t_{D2D}) \\
t_{wc}^{pipe} = \max \{ \max \{t_{BE}, t_{RM} \}, \min \{t_{BE}, t_{RM} \} + t_{EN} \}. 
$$

The combined worst propagation delay when using pipelining $t_{wc}^{pipe}$ is the maximum of the faster path plus the enable time $t_{EN}$ and the slower path (in both cases the forward path and the backward path). The pipelined worst case propagation delay $t_{wc}^{pipe}$ against the width in terms of $\lambda$ is extrapolated as plotted in Figure 3.48. The results reveal a material improvement as compared to the unpipelined version in Figure 3.47. Now the Spartan-3 device allows implementing buses that are up to $(10 + 1) \cdot 4 \cdot 2 = 88$ CLB columns wide at a speed of 50 MHz. These 88 columns would be 8 CLB columns wider than the amount that is offered by the biggest device within the Spartan-3 family and nearly double the amount, achieved for the unpipelined version.

**Fan-out Impact on the Signal Latency**

The signal propagation is not only related to the selected routing resources and the distance of a signal path, but also to the number of inputs connected to a particular
3.7 Experimental Results

Figure 3.48: Extrapolated worst case propagation delays for buses with one pipeline register in dependence of the number of attached interleaved resource slots $\lambda$.

wire. This number of destinations of a particular wire is called fan-out\(^8\). A high fan-out will lead to a considerable increase of the propagation delay on the critical path of a specific net. This increase must be known when the implicit connection scheme is used, where the module is connected to a ReCoBus or a connection bar without additional logic, and hence, without an additional driver. Connecting a module via an extra driver will decouple the module from the communication architecture, such that the module logic and routing has no influence on the signal timing of the communication architecture. The fan-out of internal bus signals can become very high, for example, in the case the partial modules contain a large register file that is directly connected to the ReCoBus.

When taking the impact of the fan-out into account, timing can be guaranteed, regardless of the current combination of reconfigurable modules, integrated into the system at runtime. Note that a full timing analysis is impossible to be computed at runtime for an embedded system, such as a ReCoNet and that these effects have to be considered at design time. The impact of the fan-out on the signal propagation delay of a single signal wire was examined with the test circuit depicted in Figure 3.49. The experiments have been carried out for a signal that is once routed regularly with the help of hex lines and once with double lines, as it would be for an implicitly

\(^8\)In general, fan-out states the amount of inputs a logic gate output can drive while still being able to reach the technology specific signal levels at the inputs. For FPGAs produced in CMOS technology, reaching signal levels is of less concern for static signals and the fan-out denotes the number of sinks a signal is connected to. Its value is related to the dynamic behavior of a signal.
3. *Intra-FPGA Communication Architectures for Reconfigurable Systems*

**Figure 3.49:** Mapped test circuit for measuring the impact of the fan-out on the signal propagation delay of a Xilinx XC2V6000-6 FPGA. Starting from a regularly routed signal path (shown in the top for a hex line path) with one source at the left hand side of the FPGA and one sink at the opposite site, the design was successively extended with additional sinks. The figure shows the initial path and the path after extending it with 1120 additional sinks. The time between the source and the first included sink was determined with the Xilinx timing analysis tool each time a new sink was included. The results are displayed in Figure 3.50.

connected shared write signal or the internal signal of a communication bar. The increase of the propagation delay on the initial path has been measured each time, after including and connecting a new sink to the test circuit. The results are plotted in Figure 3.50 and point out that in the case of hex lines, the propagation delay can rise by up to 40%. For double lines, the increase was found to be 8%. For interpreting the results in Figure 3.50, it must be mentioned that the additional sinks have been included consecutively one row after the other and that one row includes altogether 80 sinks. As can been seen, the propagation delay increases linear with strongest slope for the first row and runs into a saturation after two to three rows in this experiment.

It was found that each wire segment has its own driver, thus electrically decoupling the wire output from the input. As the amount of possible destinations is limited for each wire segment, the impact on the propagation delay is limited, as well. While the first rows of sinks can directly connect to the initial signal path, this is not possible for all following sinks, because of the limited amount of direct connection terminals. Consequently, connections have to be provided via additional (decoupled) wire segments. As the sinks are regularly distributed among one row, the amount of connections to a particular segment of the initial prerouted signal path is about the same for each segment which explains the linear slopes in the result.

With this knowledge, an upper bound for the fan-out impact can be specified by determining the propagation increase after connecting all possible destinations of one particular wire segment to some sinks. For double lines, an increase of up to 12.7% in the propagation delay was found and an increase of up to 97% for hex lines on a Xilinx XC2V6000-6. In the latter case, the propagation delay increased from 0.596 ns to 1.175 ns when connecting all remaining taps of a hex line to some sinks.
3.7 Experimental Results

Figure 3.50: Impact of the fan out (amount of connected look-up tables) on the propagation delay of a signal spanning the complete width of a Xilinx Virtex-II XC2V6000-6 FPGA.

These values are upper bounds that will typically not be reached in practical designs. However, by reducing the slack on shared write signals with the maximal possible delay increase, partial modules can be implemented that will never violate the timing specification at runtime. Note that different wire segments of the same span differ with respect to their timing, because a higher driver strength requires more FPGA die area, while only the minority of the wire segments are used on timing critical signal paths [LAB+05]. This may be used for also implementing the communication architecture in a timing driven manner.

High Throughput Test System

For demonstrating upper throughput limits on a ReCoBus, a test system was built, containing a stimuli pattern generator, an unpipelined 32-bit ReCoBus, and some reconfigurable test modules, as it is depicted in Figure 3.51. The set of test modules contains only simple modules, such as Boolean functions. This allows keeping the achievable clock frequency of the system high. Whenever a particular reconfigurable module is written by the stimuli generator, an equivalent counterpart module is written at the same time inside the stimuli generator that is located in the static part of the system. This allows to verify proper bus operation by comparing all return values of read operations with the expected results from the internal test modules of the stimuli generator. The test pattern generator (a linear feedback shift register) and the error checker is pipelined such that the overall clock frequency is mainly related to propagation delay of the ReCoBus macro.

For this system, a requested clock frequency of 100 MHz was achieved for a ReCoBus with eight resource slots that are each $W = 2$ CLB columns wide on a Xilinx
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

![Figure 3.51](image)

Figure 3.51: Test system with a 32-bit ReCoBus connected to a stimuli pattern generator. In this example, each resource slot is $W = 2$ CLB columns wide and the bus is neither interleaved nor pipelined. The enlarged view in the right hand side of the figure shows a simple test module that is connected to the bus. The measured throughput of the ReCoBus is 800 MB/s on a Xilinx Virtex-2 XC2V6000-6 FPGA. The exact reported frequency was 104 MHz. By using only the address signals for controlling write operations to the test modules and respectively only the reconfigurable select generators for controlling read operations, read and write operations can be performed concurrently. This is possible as all modules contain only a single output register that is directly connected to the read data multiplexer chains. As a consequence, the bus is capable to transfer data at a throughput of 800 MB/s between the stimuli generator and the reconfigurable modules.

The stimuli generator contains two round robin schedulers, one for determining a module for the next read operation and another one for the next write operation. The schedulers ensure that modules are only accessed after proper module configuration. The partial runtime configuration and the schedulers are controlled by an external host PC. For this test setup, no errors have been found on the bus, while the system was continuously performing read and write operations at full speed, as well as exchanging test modules by partial runtime reconfiguration. Note that the configuration interface controller uses its own independent clock domain. This helps to detect transient effects due to the reconfiguration that may be masked otherwise.
because of synchronizing the system with the reconfiguration process (also consider Section 3.7.4 that discusses issues on glitch-free reconfiguration).

### 3.7.2 Logic Cost of the ReCoBus Architecture

As revealed in the introduction of this chapter, communication architectures for reconfigurable systems must be implemented with a very low resource overhead in order to achieve an overall benefit due to partial runtime reconfiguration. While the communication bar approach for dedicated circuit switching connections omits any logic overhead in modules that are not connected to a particular link, some logic for providing access to a ReCoBus is required in all resource slots. The ReCoBus architecture is highly adaptable to the entire system requirements in terms of the bit widths for the data and address buses, the protocols (master or slave bus, pipelined or combinatorial), the granularity, or the placement style (one-dimensional slot based or a two-dimensional grid style). The aim of this section is to benchmark the different ReCoBus parameters for identifying the respective implementation cost and to take the best design decisions for a particular reconfigurable system. This covers also to evaluate the overhead functions for the most promising techniques as reported in Section 3.5. Note that the logic required to implement a ReCoBus is not completely overhead as it includes most of the bus multiplexing logic and some further logic. This logic would also be required when implementing a static-only system that does not use runtime reconfiguration.

For disclosing just a fraction of the diversity of the ReCoBus architectural parameters, three variants of a 32-bit slave bus are shown in Figure 3.52. The ReCoBus implementations vary in the resource slot width $W$ and the interleaving factor $N$. It can be seen that the implementation cost per resource slot shrinks substantially for highly interleaved buses (larger $N$ values). Each resource slot requires one reconfigurable reset generator for providing a module select a read enable signal and a module reset. The reconfigurable area for the shown example buses include two RAM columns that do not provide any connection logic to a ReCoBus. Note that the internal ReCoBus structure is identical for all logic columns in the case of a resource slot width of $W = 1$. For a resource slot width of $W = 2$, the layout at the RAM columns differs, thus disallowing to relocate a module, synthesized for the left RAM column to the right one. The reason for this stems from the fact, that the routing distance from one RAM column to the other one is a prime number (12 CLB columns plus one RAM column). Consequently, only a resource slot width of $W = 1$ or a multiple of the RAM column distance would allow an identical resource slot layout, where the latter case is impractical to implement.

In addition to the ReCoBus communication macros, some extra resources are required for the alignment multiplexing and some glue logic that is located in the static part of the system. This cost has been determined for a 32-bit ReCoBus by implementing this logic on a Xilinx Virtex-II FPGA. The results are listed in Table 3.4 as
Figure 3.52: Three ReCoBus implementation alternatives of a 32-bit slave bus. The bus on the top has a resource slot width of $W = 1$ and is $N = 2$ times interleaved. The bus in the middle is $N = 6$ times interleaved while the bottom one features a width of $W = 2$ and is $N = 3$ times interleaved. Write data lines (colored blue) and read data lines (red) share the same slices (the blue boxes that provide two LUTs and two flip-flops). Each bus contains a chain of reconfigurable select generators on the top, for selectively enabling read access to one or up to $N$ consecutive resource slots. The connections to the static part of the system are located at the right hand side of the figure. The gray wires in the enlarged boxes on the bottom are routing resources, available for the module implementation. For the bottom bus, it is highlighted that the signal routing at the two RAM columns is not identical. Each shown bus provides altogether 32 individual read as well as 32 write data lines, a module select, a module read enable, and a module reset signal, but no interrupt or any further address signals.
3.7 Experimental Results

Table 3.4: Logic cost (in terms of number of used look-up tables) for different configurations of a horizontal 32-bit slave bus. The cases with \((W = 1, N = 1)\), \((W = 1, N = 6)\), and \((W = 2, N = 3)\) are depicted in Figure 3.52.

<table>
<thead>
<tr>
<th>bus config.</th>
<th>static connection</th>
<th>fixed cost</th>
<th>alignment multiplexer</th>
<th>glue logic</th>
<th>variable cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>read</td>
<td>write</td>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>N</td>
<td>data control</td>
<td>sum</td>
<td>align. mul.</td>
<td>sum</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>64 12 76</td>
<td>32 32 64</td>
<td>15</td>
<td>155</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>96 12 108</td>
<td>128 96 224</td>
<td>19</td>
<td>351</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>96 12 108</td>
<td>128 144 272</td>
<td>23</td>
<td>403</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>64 6 70</td>
<td>0 0 0</td>
<td>5</td>
<td>75</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>72 6 78</td>
<td>64 66 130</td>
<td>19</td>
<td>227</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>64 12 76</td>
<td>32 32 64</td>
<td>15</td>
<td>155</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>64 6 70</td>
<td>32 32 64</td>
<td>15</td>
<td>149</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>64 6 70</td>
<td>0 0 0</td>
<td>5</td>
<td>75</td>
</tr>
</tbody>
</table>

The fixed cost together with the variable cost within the resource slots. As can be seen, the resource overhead in the static part of the system increases remarkably with the interleaving value \(N\). On the other side, interleaving also leads to a remarkable lower resource cost inside the resource slots. Consequently, interleaving is more suitable for larger systems.

In Chapter 3.5.6, it was proposed, how the one-dimensional interleaving scheme can also be applied for supporting a two-dimensional grid-based placement style of the reconfigurable modules. For this enhancement, the absolute logic cost within each resource slot will not rise and also the cost for the alignment multiplexers will be the same as compared to the one-dimensional case. Only the connection cost in the static part of the system has to be spent for each row of resource slots plus some additional glue logic (the OR-gates in front of the read alignment multiplexers shown in Figure 3.39). This relationship is depicted in Figure 3.53 that lists the total cost, required to implement 32-bit slave buses for different ReCoBus configurations. The values in the figure are based on Table 3.4 which lists the synthesis results determined for a Xilinx Virtex-II FPGA. In Table 3.4, the variable cost denotes the sum of the resource estimates denoting the cost in the resource slots. These estimates have been introduced in Section 3.5. For shared write and shared read signals (see Equation 3.5 and Section 3.5.3), the logic cost within the slots is \(S_{SW}/N\) and \(S_{SR}/N\). Note that these two signal classes can share the same resources, because the ReCoBus implementation uses the look-up tables for connecting shared read signals and the additionally available flip-flops for shared write signals at the same time. For the examined 32-bit slave ReCoBus implementations, this results in a variable cost of \(32/N\) look-up tables for implementing these two signal classes. Table 3.4 denotes a variable cost that is
always two look-up tables higher. This is the overhead required to implement the reconfigurable select generator as depicted in Figure 3.22.

As the logic cost is not related to the resource slot width $W$, only the cost for the smallest resource slot width is plotted for a particular interleaving value $N$. The first example ($R_V = 1$) depicts the logic cost for a one-dimensional slot based placement style. Here, it can be seen that interleaving starts to pay-off for more than 6 resource slots if a ReCoBus is two times interleaved ($N = 2$) and for more than 13 slots in the case of $N = 6$. By dividing one large vertical slot into multiple smaller ones ($R_V = 2, \ldots, 4$), the fixed logic cost in the static part of the system increases (mainly because of the additional connection cost) and the gradient of the linear slopes steps by the amount of vertical slots $R_V$. The total amount of resource slots within the system is $R = R_V \cdot R_H$. For a Xilinx Virtex-II XC2V6000 FPGA, this results for a resource slot width of $W = 1$ to the following amount of look-up tables per resource slot:

<table>
<thead>
<tr>
<th>$R_V$</th>
<th># LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>786</td>
</tr>
<tr>
<td>2</td>
<td>386</td>
</tr>
<tr>
<td>3</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>192</td>
</tr>
</tbody>
</table>

Figure 3.2 on page 87 shows the impact of the resource slot size and the communication cost on the overhead, caused by internal fragmentation. As can be seen in that figure, communication architectures, requiring 10 LUTs per resource slot, minimize the internal fragmentation if a single resource slots provides around 200 to 250 LUTs (for the assumed module distribution). Consequently, with the higher interleaved ReCoBuses ($N = 4$ and $N = 6$), this slot size can be implemented by tiling the full height of the XC2V6000 FPGA into three to four subcolumns without an explosion in the communication logic cost. Note that the ReCoBus that is $N = 6$ times interleaved requires 403 LUTs for $R_V = 1$ and 763 LUTs for $R_V = 4$ in the static part of the system. This difference of 360 LUTs is less than the average overhead stemming from internal fragmentation of just two modules (assuming a communication cost of 10 LUTs per resource slots for the example in Figure 3.2). To summarize this observation: **minimizing internal fragmentation by using a fine module resource slot layout is more important than optimizing the logic overhead in the static part of the system.**

In addition, a fine horizontal grid ($W = 1$) allows easier module relocation in the case of dedicated resource columns (e.g., a RAM column) as shown in Figure 3.52. Finally, tiling the resource slots in vertical direction allows reducing the fragmentation of the dedicated columns, as multiple modules may use resources from the same RAM or multiplier column. As exemplarily shown in Figure 3.65 on page 188, it is not necessarily required that all used resource slots of a module are conjoined into one consecutive block.

So far, only a slave bus has been examined without dedicated read signals (e.g., interrupts) and addresses. As presented in Section 3.5.4, dedicated read signals can be implemented without additional logic resources inside a ReCoBus by manipulating tiny fractions of the configuration data. Alternatively, the time multiplexing
Figure 3.53: Total implementation cost in terms of number of required look-up tables (the ordinate) in dependence of the amount of horizontal $R_H$ (the abscissa) and vertical $R_V$ resource slots for different interleaving factors of a 32-bit slave bus implementation.
approach can be used (see Figure 3.31 on page 136) that requires two LUTs within each resource slot. For providing address signals inside the resource slots, shared write signals can be interleaved as illustrated in Figure 3.18 on page 118. For instance, if the address space for the reconfigurable modules shall be 1K words for a ReCoBus that is 6 times interleaved, it requires \( \lceil 10/6 \rceil = 2 \) additional LUTs per resource slot (see also Equation (3.5) on page 118).

A ReCoBus can also implement a bus master interface. In the easiest case, another instance of a macro, shown in Figure 3.52, can be used to link the master output addresses towards the static part of the system. The resources of the read multiplexer chains can be shared with the address signals for the slave port. In the case of a 32-bit address bus, the cost in the static part of the system and inside the resource slots will double by this approach. Alternatively, addresses and data may be distributed in a time multiplexed manner on the same resources. This allows implementing the ReCoBus logic for a master interface inside the reconfigurable area at the same cost as compared to a pure slave interface. However, this approach comes along with additional latency and more complex logic within the static part of the system.

3.7.3 The Impact of Modularization on the Physical Implementation

Partial runtime reconfiguration requires to completely encapsulate the modules into a set of predefined resource slots. This means that modules are constrained to use logic and routing resources only within a predefined bounding box that can be adjusted in terms of resource slots. Consequently, this design methodology will influence the physical implementation of the modules. Only the communication via the module interface (through its entity) will pass the bounding box of a module with the help of special communication primitives (like a ReCoBus or a connection bar).

The impact on routability (the question, if all nets can be routed), power consumption, and speed, when constraining modules into fixed resource slots, was examined in [KPR04a]. The authors found that smaller modules pay a considerable penalty with respect to the achievable clock frequency for extreme aspect ratios. For example, a divider that is just one CLB column wide and that spans over the full height of the device has had a 94% increase in the propagation delay. For larger modules with more than 1000 slices (equivalent to 2000 4-bit LUTs), the timing overhead was in average below 10% for the results presented in [KPR04a].

For this thesis, similar experiments have been performed, and in addition, the impact of blocking specific wires for implementing the internal routing of a module was examined. This characterizes the cost of the ReCoBus and connection bar architecture in terms of routability and latency. Four different modules have been examined that all occupy more than 90% of the available logic resources within a bounding box that was set to a height \( h \) of 32 CLBs and width \( w \) of 5 CLBs. With eight look-up
3.7 Experimental Results

tables per CLB, this results in 1280 LUTs in total that are available within the bound-
ing box. Besides an FIR filter, a DES56 core, an FFT module, and a synthetic module
have been selected as benchmark modules. The latter implements a huge congested
ring shifter that cannot be fully routed with the selected logic utilization. The follow-
ing code fragment lists the kernel (in VHDL) that is responsible for the congestion:

```vhdl
if clock'event AND clock='1' then
  for k in 0 to tmp_size-1 loop
    tmp(k) <= I(k MOD in_width) XOR
        tmp((k+distance_1) MOD tmp_size) XOR
        tmp((k+distance_2) MOD tmp_size) XOR
        tmp((k+distance_3) MOD tmp_size);
  end loop;
end if;
```

The shifter consists of a large internal register (tmp) where each single flip-flop is
driven by a look-up table implementing a 4-input XOR-gate, hence, not allowing any
functional optimizations. The flip-flop outputs are circular connected to the inputs
of the XOR-gates. By increasing the distance between the connected flip-flops
highly congested circuits can be generated. These circuits can be placed on an FPGA
but they are impossible to be fully routed.

The experiments have been carried out on a Xilinx Virtex-II device that has 10
double as well as 10 hex lines starting in each cell towards all horizontal and vertical
directions. In each CLB from one to five double lines, hex lines, and double and
hex lines together have been blocked. The blocking was only done for horizontal
wires, but each time for both directions, because the proposed communication ar-
chitectures occupy almost only horizontal wires. Figure 3.54 depicts an exemplary
test design with the FIR filter where five double lines, as well as five hex lines are
blocked. Consequently, these wires are not available for the module implementation,
but are reserved for implementing the communication architecture responsible for
integrating the partial modules into the system at runtime.

Table 3.5 lists the total number of wires that can be used to distribute data over
the reconfigurable area. This implies that wires are interleaved and that a wire is ex-
tended if it ends in a CLB switch matrix within the module bounding box, as shown
in Figure 3.54. For example, in the case of blocking 3 hex & double lines in both
horizontal directions, the hex lines will be interleaved in 6 independent chains, each
containing three wires that are starting and respectively stopping in six horizontal
consecutively aligned CLBs outside the module bounding box. Analogous, the dou-
ble lines are interleaved in 2 independent chains. Therefore, one CLB row is blocking
\((6+2) \cdot 3 = 24\) wires per horizontal direction in this example. These are 48 wires for
both directions and 1536 wires in total over the full module height of \(h = 32\) CLBs.

In addition to the amount of blocked wires, the table lists the relative achieved
latency after routing the module with respect to a routed module without area con-
Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.54: FIR filter module constrained into a (yellow) bounding box. In both horizontal directions, five double lines (red) and five hex lines (blue) have been blocked for the module implementation. Special blocker macros, located around the bounding box, force the router not to use routing resources outside the module boundary (see also Section 4.1.2).

The first row of the table shows the influence when fitting a module into a fixed bounding box. The determined values confirm the results presented by Kalte et al. in [KPR04a]. All other values reveal the impact of additional wire blocking. The
### Table 3.5: Impact of wire blocking on the latency and the number of unrouted nets

The table contains two results for each module and wire blocking combination as two different methods for constraining the placement have been used. As expected, there was no strong performance decrease when blocking a larger amount of hex lines. The reason for this is that the module is only 5 CLBs wide. Therefore, only 40% of all hex lines can be accessed within the module bounding box via the mid-ports that allow to tap each hex line in the middle, after a routing distance of three CLBs. In other words, 60% of all horizontal hex lines could not be used for implementing inter-module communication because of these wires would leave the module bounding box. Because of the narrow, but tall aspect ratio that is best suitable for partial runtime
reconfiguration on Xilinx FPGAs, and because of the carry chains for implementing arithmetic functions that have a predefined vertical direction, the horizontal routing is much less congested than the vertical one, as depicted in Figure 3.55.

Figure 3.55: Magnified view of the partial FIR filter module that is constrained into a bounding box. The gray colored wires indicate unused routing resources and cyan is used for the module routing. While the vertical routing channels are almost fully utilized, there are still plenty of horizontal resources available.

3.7.4 Glitch-free Reconfiguration

When fractions of a system are exchanged by partial runtime reconfiguration, it must be ensured that this can be accomplished without unpredictable side effects to other parts of the system. In the best case, exchanging partial modules can be performed completely transparent with respect to the rest of the system and without affecting the communication infrastructure. This enormously simplifies the interaction between the reconfiguration management and the execution of the rest of the system. For example, a communication architecture may be isolated during runtime reconfiguration, but this may disrupt the communication for an unacceptable long time. Additionally, a transparent configuration process does also not demand any synchronization between the reconfiguration and the bus schedule, for example, to wait on a burst transfer completion.

This section is devoted to testing the behavior of the proposed ReCoBus and communication bar macros during reconfiguration. Firstly, it will be introduced how glitches can be measured on FPGAs and how this can be used to prove that the communication architecture allows to exchange modules at runtime without influencing
other parts of the system. This will be shown on the ESM platform [MTAB07] that provides a Xilinx Virtex-II FPGA. After this, the proposed communication architecture will be extended, such that also Spartan-3 FPGAs can be used for implementing ReCoBuses glitch-free.

The disturbances due to a reconfiguration process can be classified into three categories:

1. **glitch-free reconfiguration**: No influence to other modules or bus transactions if modules are exchanged.

2. **Glitch reconfiguration**: A transfer on a ReCoBus or communication bar may be disturbed seldom and for one predictable configuration clock cycle.

3. **Interrupted reconfiguration**: The bus may be disturbed or interrupted for multiple consecutive configuration clock cycles.

In almost all cases when the Xilinx documentation reports about glitches during runtime reconfiguration, this stands for the last category and disturbances may occur for many hundreds of configuration clock cycles (depending on the configuration data and the device). This categories have to be considered, when glitch-free reconfiguration has to be implemented on Xilinx Spartan-3 FPGAs, as revealed later in this section.

**Measuring Glitches on FPGAs**

In order to prove that partial runtime reconfiguration can be performed glitchless, such events must be measurable. Measuring glitches on FPGAs can be performed by the use of an oscilloscope, if they occur at I/O pins. This has to be done with caution, because glitches are events lasting typically in the area of a few nanoseconds or even below, if they occur on-chip. Consequently, this requires an oscilloscope with a high input bandwidth in the gigahertz range. However, it is virtually impossible to connect an oscilloscope probe to an on-chip signal in order to detect glitches on this signal. Also, adding a connection from such a signal to an output pin by using some FPGA routing resources allows only a limited detection of glitches, because pin drivers typically have significant capacities and will consequently filter potential glitches. To overcome such limitations, dedicated test circuits have been developed [KSHT08a] (see Figure 3.56) that can be added to existing systems in order to detect glitches on any wire.

This allows to prove that a partial reconfiguration process will not interfere with running data transfers on a ReCoBus or a communication bar. For demonstrating glitch-free runtime reconfiguration, when using the proposed communication macros, a test system was implemented on the ESM [MTAB07], as shown in Figure 3.57. In the static part of this system, random test data has been applied on all macro inputs.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.56: Glitch detection circuits: a) glitch triggered latch, b) glitch triggered flip-flop. After releasing reset, the occurrence of a glitch event at the probe input will be indicated at the glitch output. With the pos/neg input signal in a), the logic level of the idle state can be selected.

when exchanging some partial test modules. At the same time, the absence of glitches have been checked with the circuits depicted in Figure 3.56. Testing for glitches has been accomplished at the macro outputs, as well as on the internal signals of the communication architecture. As expected, no glitches have been detected during a reconfiguration process on a ReCoBus when using glitch triggered latches or glitch triggered flip-flops. This holds only for a ReCoBus architecture implemented on a Xilinx Virtex-II device. For Spartan-3 FPGAs the reconfiguration process follows more restrictions as revealed in the next section.

Connection bars are also not interfered due to a reconfiguration process as long as no module is included to or removed from a particular bar that also drives the bar. Exchanging a module that drives a connection bar will lead to a glitch reconfiguration and may also remove or include one or more registers in the data path. This is a short term event and occurs only on rearranging the data path. However, the point of time of this event is known, thus allowing to mask configuration effects to the rest of the system.

Tolerating Glitches on Spartan-3 FPGAs

The ReCoBus architecture aims at building systems, where modules can be hot-plugged to a reconfigurable bus at runtime, by glitch-free partial reconfiguration. If a partial reconfiguration will influence a system depends mainly on the capabilities of the target FPGA architecture. For example, Xilinx Inc. guarantees glitchless reconfiguration modes on Virtex-II, Virtex-4, and Virtex-5 FPGAs for all logic, memory, and routing resources, while interruptions or disturbances are the normal case for Spartan-3 FPGAs. Note that it is only guaranteed that identical fractions of a configuration can be written multiple times to a Virtex FPGA without causing glitches. If a circuit is modified or even exchanged some side effects may occur. Note that the ReCoBus and communication bar macros will never change and that these macros are capable of masking effects from a reconfigurable module to the communication architecture. In the case, a reconfigurable system should be implemented on Spartan-3 FPGAs the ReCoBus or communication architecture needs to be enhanced for supporting glitch-free reconfiguration.
3.7 Experimental Results

Figure 3.57: Test setup on an ESM [MTAB07] for proving glitchless reconfiguration. The logic at the bottom left corner contains the interface between a host PC and the internal logic. In addition to the configuration interface (an ICAP port shown in the red box), the system contains a ReCoBus (colored brown) and two connection bars in opposite direction (colored green). During reconfiguration of resource slots within the yellow box, static test data was applied to all macro inputs while observing the outputs with glitch detection circuits (highlighted in the green box).
Supporting Spartan-3 FPGAs for building dynamically reconfigurable systems is important as these FPGAs provide significantly more logic per monetary cost as compared to devices providing glitch-less partial reconfiguration modes (e.g., all Xilinx Virtex FPGAs). As saving monetary cost is one major goal for using partial runtime reconfiguration, it is obvious to also apply advanced partial reconfiguration techniques to these devices. Unfortunately, Spartan-3 FPGAs cannot be easily used for building systems that support glitch-free reconfiguration. The reason for this behavior is that Spartan-3 devices are blockwise reconfigurable FPGAs that require a clear operation to a complete logic column prior to the particular configuration data write operation. This clear operation also deletes the switch matrix setting within the logic column, hence potentially interrupting a ReCoBus or connection I/O bar. After the clear operation, a new configuration can be written in a frame by frame manner. A configuration frame is the smallest atomic piece of configuration data and a Spartan-3 FPGA has 19 frames that describe a complete logic column and the according switch matrices. The entire frame size depends on the FPGA and is in the range of a few hundreds to more than a thousand bytes. The delete operation is performed by sending a so-called snowplow command to the Spartan-3 FPGA. With this command, the configuration data of one complete column will be reset to a logical ‘0’ value. Consequently, communication macros will be interrupted if they use logic or routing resources of a reconfigured logic column.

In order to perform completely glitch-free reconfigurations on block erasable FPGAs, such as Xilinx Spartan-3-FPGAs, it must be possible 1) to exchange logic and routing of a column without influencing the surrounding system. In addition, 2) it must be possible to bypass the communication after any block erase. For achieving the first point, the behavior of wires during a column delete and configuration processes was examined. It was found that all horizontal double and hex lines carry a logical ‘1’ value if the column is deleted. This is independent from the state of all possible sources to such a line. This behavior probably stems from a level restorer inside the switch multiplexers as revealed in [LAB05].

For ensuring that the bus logic is fully available when connecting it to the neighbor resource slots, the configuration write process was split into two phases. In the first phase, the complete CLB column is configured, except the routing resources to adjacent slots that will be configured in a second step, as illustrated in Figure 3.58. Note that in Spartan-3 FPGAs, the configuration bits can be incrementally set over multiple write operations, while it is only possible to globally reset a full column. This is opposed to Virtex-II FPGAs, where all configuration bits can be randomly set or reset.

So far, it was assumed that configuration bits of the multiplexer Muxd (in Figure 3.58) are stored inside the same frame. In general, this multiplexer has much more inputs and consequently more configurable control bits that may be spread over multiple configuration frames. This situation is classified with the frame distance $FD$ that will be defined as the number of frames that differ between two configu-
3.7 Experimental Results

Figure 3.58: a) Signal path of a read multiplexer chain and b) a detailed view on an FPGA implementation. When this chain is implemented on a Xilinx Spartan-3 FPGA and slot i is deleted by a so-called snowplow command, the bus is interrupted and slot i-1 will sense a logical '1' at its input. If firstly the complete bus logic is configured (e.g., Muxa, ..., Muxc) and after this the output to the following resource slot (Muxd), slot i-1 will not sense an unintended '0' value during reconfiguration.

...
Figure 3.59: Test circuit for detecting glitches, resulting from reconfiguring a Re-CoBus resource slot, implemented on a Xilinx Spartan-3 FPGA. The CLB in the center drives an output signal via a double line (highlighted red) to another CLB containing a glitch detection latch. The shown blue routing resources pass the center CLB switch matrix and connect all further possible sources for the red output double line. By setting the state of the blue alternative sources different to the state of the red target signal wire, unintentional settings for the outgoing double line can be detected that may occur during removing a CLB column or during the following reconfiguration process.
inputs are possible by the use of additional routing resources.

With the help of such configuration-aware resources, no glitches have been detected in the test system (shown in Figure 3.59) on swapping some reconfigurable modules. A reconfiguration of a module that is one logic column wide \((w = 1)\), can be performed glitch-less on Spartan-3 FPGAs, when performing the following steps:

1) Disable any read or write operations to the modules that are involved in the reconfiguration process. The logic in the look-up tables of the read multiplexer chains must result in a logical ‘1’ value when the resource slot is not selected.

2) Clear the complete column by sending the column address information followed by the snowplow command to the Spartan-3-FPGA.

3) Write the module configuration with the bus logic to the FPGA. All configuration bits that connect the output wires are cleared in this process.

4) Configure the output wire connections to the bus.

5) Enable full access to the currently reconfigured modules.

Within steps 2) to 4), the inputs of the adjacent resource slots will detect a ‘1’ value without any glitch. In general, the assumption that a resource slot and a module is just one logic column wide is too restrictive. If multiple columns have to be reconfigured, steps 2) and 3) have to be sequentially repeated for each involved column until a complete resource slot is reconfigured.

As reconfiguring a slot leads to a stable ‘1’ value at its output, the communication architecture should be designed such that a ‘1’ value is neutral to all communication macro outputs. Consequently, the read multiplexer chains, proposed in Section 3.5.3, should be implemented with a distributed AND-gate instead of a distributed OR-gate. This can be achieved by applying De Morgan’s theorem on the original read multiplexer chain:

\[
\bigvee_{k=0}^{R-1} (en_k \land data_k) = \bigvee_{k=0}^{R-1} (en_k \land data_k) = \bigwedge_{k=0}^{R-1} (en_k \land data_k)
\]

Furthermore, redundant wires are used for bypassing the resource slots during reconfiguration, as depicted in Figure 3.60. Note that the output wires of a resource slot are connected to two different succeeding slots such that there is always one path available to a succeeding slot in the case a single resource slot is reconfigured. This does not demand two routing wires on Xilinx Spartan-3 FPGAs, as there are routing resources available that connect multiple cells with the same wire.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

**Figure 3.60:** Read multiplexer chain for glitch-less reconfiguration on Xilinx Spartan-3-FPGAs. During reconfiguration of slot i, the fat-drawn wire will be in a stable ‘1’ state and the dashed wire will bypass slot i.

**Write Signal Distribution for Spartan-3 FPGAs**

The bypass technique, presented in the last section can, also be applied to write signals, as shown in Figure 3.61. In the example, again, wires that are connected to the next and to the second next resource slot are used. The output wire configuration can only be ensured to be glitch-less if the corresponding multiplexer can be set with a one-hot configuration codeword, as mentioned in the last paragraph. In case of Xilinx Spartan-3-FPGAs, it was found that the output signal has to be routed over a stopover point in order to set the output multiplexer glitch-free with a one-hot configuration codeword. The configuration data for this stopover point must be written prior to the configuration word that sets the output multiplexer.

As an alternative implementation, a triple wire redundancy technique was examined that is also suitable for distributing write signals completely glitch-less on Xilinx Spartan-3-FPGAs. As shown in Figure 3.62, this technique uses three different signal paths in parallel for distributing the write signal. The signal paths are displaced in such way that a reconfiguration process of a resource slot (here a single CLB column)
3.7 Experimental Results

Figure 3.62: Triple wire redundancy write signal distribution.

may disturb only one path, while the other paths are not influenced. A majority voter in each destination resource slot will mask these disturbances. Both techniques have been successfully tested on a Spartan XC3S-200 FPGA. In both cases, no glitches have been detected on a snowplow clear operation or the following reconfiguration step on other parts of a simple test system.

As it requires additional logic for masking the configuration effects from the bus, write and read signals cannot share the same look-up table flip-flop resource combination as mentioned for shared write signals in Section 3.5.1. But it is possible to apply the interleaving technique. This reduces the overhead for all $S_W$ write signals in a system with $N$ interleaved signal paths to $\lceil \frac{S_W}{N} \rceil$ LUTs per resource slot. Thus, a module requiring all $S_W$ write signals must be at least $N$ resource slots wide. Figure 3.63 gives an example for the bypass wire approach, tailored to Xilinx Spartan-3-FPGAs that makes use of hex lines that route over a span of six CLBs and that can be tapped in the middle. The example shows the case where each resource slot is $W = 1$ CLB column wide. In general, each of the $N$ signal paths consists of two tracks that each span over a routing distance of $2 \cdot N \cdot W$ CLBs. The two tracks are displaced by $N \cdot W$ CLBs and all paths are tapped in the middle, such that the AND-gate for each interleaved write signal in a resource slot is connected to both associated tracks (see also Figure 3.61).

Overview on the Results

This section is devoted to implement partial runtime reconfiguration without disturbing communication that is carried out via a ReCoBus or a connection bar. In particular the following milestones have been achieved [KBT09a, KSHT08a, KSHT08b]:

- Design of on-FPGA glitch detection circuits.

- Prove that glitch-free partial runtime reconfiguration is provided in systems integrating partial modules with a) a ReCoBus and b) connection bar macros. This has been demonstrated on the Xilinx Virtex-II main FPGAs of the ESM platform [MTAB07].
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.63: Write signal distribution with \( N = 3 \) interleaved chains for glitch-less operation on Spartan-3 FPGAs. a) Logic and routing structure including the required alignment multiplexer. The fat drawn signal path is for the least significant subword of a module placed into slots 4 to 6. b) Physical implementation on a Xilinx Spartan-3 FPGA.

- Transferring the results to Xilinx Spartan-3 FPGAs, that are not especially designed for glitch-less reconfiguration.

3.7.5 Communication for Two-dimensional Module Placement

In the introduction of this chapter is was proven that the resource slots should be kept only a few hundred LUTs in size in order to optimize internal fragmentation. On a large FPGA, such as the Virtex-II-6000 of an ESM [MTAB07], this requires partitioning of the reconfigurable area into a two-dimensional grid for placing the reconfigurable modules. This also enhances the utilization of the resource columns providing dedicated resources including memory blocks and multipliers, because of the same dedicated resource column may be shared among multiple modules.

For demonstrating the two-dimensional communication capabilities of the proposed ReCoBus and connection bar communication architecture, the test system depicted in Figure 3.69 on page 193 was exemplarily implemented for the ESM platform [MTAB07]. This system provides a reconfigurable area with \( R_h = 62 \) (logic resource slots) + 4 (additional memory resource slots) in horizontal direction. In vertical direction, the full height of the device was partitioned into \( R_v = 4 \) rows of re-
3.7 Experimental Results

Figure 3.64: Example system featuring a two-dimensional grid style placement on an ESM [MTAB07]. The reconfigurable area is tiled into $R = R_h \cdot R_v = [(62 \text{ (logic)} + 4 \text{ (RAM)}) \cdot 4] = 264$ individual resource slots. A resource slot is one CLB (one RAM/multiplier column) wide and 24 CLBs (6 block RAMs/multipliers) in height. This results in $\sigma = 192$ look-up tables. In each resource slot, containing logic, modules can connect to a ReCoBus and to connection bars for video data (24+3 bit bidirectional) and push buttons. In addition, modules can connect to a circuit switching network for directly linking input pins (the ESM debug pins from the top) to the freely placeable modules. With a connection cost of $c = 14$ LUTs in each slot, the bus provides access to a slave bus interface with up to 8-bit read data, 8-bit write data, two address bits, a module select, a read enable signal, a module reset, and a shared interrupt wire per slot. The static system at the left hand side of the FPGA generates a VGA background data stream that can be linked to any of the four horizontal video bars. The output of a particular bar is passed to a pipeline register within the connection macro and directly send backwards to the left from where it can be send another time via any other video bar towards the right part of the static system. A register controlled multiplexer inside the right static part selects the particular output video bar. This system is controlled by a host PC for performing bus transactions and for sending configurations to the internal configuration access port (ICAP) on an XC2V6000 FPGA.
3. *Intra-FPGA Communication Architectures for Reconfigurable Systems*

**Figure 3.65:** Four implementation alternatives of the same FIR filter module. All variants connect the module within four resource slots to a 32-bit two-dimensional ReCoBus. The first example uses only logic resources and is $w = 15$ resource slots wide and $h = 2$ slots high. The second uses dedicated RAM/multiplier resource slots and requires only $(3_{\text{logic}} + 1_{\text{RAM}}) \times 2$ slots. The two implementations at the right hand side are based on the same netlist as the second variant but have been implemented with a different form factor of $(6_{\text{logic}} + 2_{\text{RAM}}) \times 1$. The last alternative provides a specialty, as it relaxes the requirement that the resource slots have to be continuously arranged. The module skips 4 logic resource slots and connects a single outlaying RAM/multiplier resource slot without using any routing resources within the switch matrices of the skipped slots. The gap can accommodate a module that may also access the full 32-bit of the ReCoBus without interfering with the surrounding FIR filter module.

source slots that are each $H = 24$ CLBs high. This is an integer multiple of the block RAM height, and the vertical routing span of the hex lines. Therefore, the modules may access equivalent resources (a particular hex line, a block RAM or some logic resources) at exactly the same relative position inside a resource slot, regard-
3.7 Experimental Results

less of which row the module is currently placed in. Altogether the system provides \( R = R_h \cdot R_v = 264 \) resource slots, whereof all 248 logic resource slots can directly connect a reconfigurable module to the communication architecture. With a resource slot width of \( W = 1 \) and eight LUTs per CLB, a slot provides \( \sigma = 24 \cdot 1 \cdot 8 = 192 \) LUTs in total per logic resource slot.

This fine slot grid allows implementing the reconfigurable modules by using different resources as well as by fitting modules into different shaped bounding boxes, as depicted in Figure 3.65. In the example, an FIR-filter module is once implemented without dedicated RAM/multiplier resources and three times with these resources. The logic-only module uses \( w \cdot h = 15 \cdot 2 = 30 \) slots, whereof only four slots are used for connecting the module. With a connection cost of \( c = 14 \) LUTs, the module wastes \((30 - 4) \cdot 14 = 364\) LUTs because of the unused ReCoBus connections. This is 6.3% of the logic available within the module bounding box. The internal fragmentation, in terms of the number of unused LUTs within the module bounding box, is 45 LUTs or 0.8%. Therefore, the total overhead of logic to be paid for fitting this particular module into a bounding box and for providing communication is about 7.1% more as compared to a static only implementation. For the last two implementation alternatives shown in Figure 3.65, the waste of logic resources is \((6 - 4) \cdot 14 = 28\) LUTs for unused ReCoBus connections and another 75 LUTs because of the internal fragmentation, which results in altogether 8.9% overhead with respect to the total amount of LUTs available in the six logic resource slots of the module. It must be mentioned that only focusing on the logic utilization does not express that in some cases the logic density has to be reduced for carrying out the routing. The reason for this is that the switch matrix resources consume considerably more area on the FPGA die as the logic cells itself. Therefore, shrinking the switch matrices by leaving some logic resources unused results in an overall benefit [Deh99].

Reconfigurable modules placed in the tile grid of the system, depicted in Figure 3.64, may communicate arbitrary via 1.) a ReCoBus, 2.) two connection bars, and 3.) a two dimensional circuit switching network, as listed in the following.

ReCoBus Communication

Figure 3.64 depicts the mapped ReCoBus logic. In the right hand side of the illustration, it can be determined that the communication cost per slot for implementing a bidirectional 8 bit slave ReCoBus interface is \( c = 14 \) LUTs (inside a slot, the seven depicted slices that are providing two LUTs each). The four ReCoBus macros are horizontally \( N_h = 6 \) times and vertically \( N_v = 4 \) times interleaved. By using six interleaved chains for the ReCoBus macros, the bus can be implemented with the help of hex lines that are materially faster for larger buses as compared to the alternatively available double lines. Highly interleaved buses can be used for implementing ReCoBuses with a lower amount of possible consecutive bus connections than specified by its interleaving factor as shown in Figure 3.66. This scheme is used for this exam-
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.66: Data bus interleaving of the example shown in Figure 3.69. Only up to 4 out of 6 independent interleaved chains are used for implementing ReCoBus data lines. The numbers inside the resource slots on the right hand side of the figure denote the multiplexer select value that is determined by the most left logic resource slot as shown for one module.

ple setup and it logically matches the system illustrated in Figure 3.39 on page 146 with a two-dimensional ReCoBus that is 4 times interleaved in both, horizontal and vertical direction for providing 32-bit data access. The address signals use the full horizontal interleaving scheme. This is used for extending the two bit address range of a single resource slot to up to $2 \cdot 6 = 12$ bits or 4K in this example.

Communication Bars

In addition to a ReCoBus macro, each row of resource slots provides two horizontal connection bars (see also Figure 3.64). One bar connects the same four push buttons in parallel to all four rows of resource slots, hence, these signals can be tapped in all 246 slots. The second bar distributes a VGA video signal with 24-bit color depth to all resource slots. All video connection bars allow modules accessing the data that is streamed from the left to the right of the system. In addition to this, each connection bar provides a backward path for returning the stream again to the left hand side of the system. As revealed in Figure 3.67, a multiplexer located at the beginning of each video connection bar macro (located at the left hand side of the figure), can selectively chose between an initial signal from a background module or a loop-back signal from another macro. This allows streaming the video data over all rows of resource slots in all $2^8$ possible permutations. Note that the video background signal or the data from a backward path can be sent to multiple video connection bars in parallel. A multiplexer in the right hand side of the system selects one video connection bar to be linked to the output pins.

The complete set of video signal is divided in two subsets that are two times interleaved. This requires an alignment multiplexer/demultiplexer pair for adjusting the input stream according to the module position (odd or even resource slot index) and also for adjusting the output stream according to the initial format. This is the only logic required within the reconfigurable area for the video bar and this logic is only necessary if the module connects to a video bar. The alignment can be controlled in multiple ways:
Figure 3.67: Architecture of the video bar routing via connection bars. For each row of resource slots, a multiplexer at the left hand side of the system can freely select between an initial video stream from a background module or from a loop-back signal of another bar. The video signal is $N_{\text{video}} = 2$ times interleaved, as depicted by different colored chains. Consequently, a module must be at least two slots wide and including alignment (de)multiplexers for accessing the stream adjusted with respect to the current start position of a module in a read-modify write fashion.

1. Using identity functions instead of multiplexers in the according LUTs that are exchanged by modifying the configuration bitstream.

2. For modules, only requiring a video input interface, it is possible to implement the alignment directly within a switch matrix, because on Virtex FPGAs the signals of both chains can be tapped within the same switch matrix. This requires also some bitstream manipulation.

3. Including extra signals into the connection bar may be used to automatically control the alignment (de)multiplexers. For example, by setting one additional wire within the first interleaved chain to a constant ’0’ value and a corresponding wire within the second chain to a constant ’1’ value. Within a reconfigurable module, sensing one wire indicates if a module is placed at an even or an odd horizontal resource slot index.

4. Instead of using extra wires within the connection video bar, existing signals may be used to determine if a module is placed at an even or odd slot position index. This approach is chosen for the example presented in this thesis.
3. Intra-FPGA Communication Architectures for Reconfigurable Systems

Figure 3.68: Automatic alignment control signal generation for video connection bars. The VGA video signal contains a horizontal synchronization signal (H\_sync) and another signal indicating if the current pixel of the video stream will be visible on the monitor or not. These signals will be distributed in separate horizontally interleaved connection bars (see also Figure 3.67) in such way that the signals use the same relative wiring resources within the interleaved bars. These wires are connected within a module to a flip-flop as shown in a). If the H\_sync signal is connected to the clock input, this indicates an even resource slot index and if the visible input is connected to the clock input, this will be recognized as an odd slot index as depicted in b) that shows the fixed phase relationship between the two signals. The odd/\ even signal can be directly connected to the alignment (de)multiplexers.

connecting the horizontal synchronization signal (H\_sync) and the visible signal that is provided together with the VGA color information in corresponding wires, located in separate interleaved connection bar chains, a simple flip-flop is sufficient to automatically detect if a module is placed at an even or odd slot position, as illustrated in Figure 3.68.

Two-dimensional Circuit switching Network

As a third alternative in the demo system depicted in Figure 3.64, modules can directly connect to some I/O pins via a two-dimensional circuit switching network. Figure 3.69 reveals details of the network implementation that is carried out directly within the switch matrices of the FPGA fabric. All network inputs (here the debug pins available at the top of the ESM) are routed vertically with hexlines down to the bottom of the device. At two vertical positions within each resource slot, these hexlines can be directly connected to horizontal channels that are implemented with double lines towards both directions, eastwards and westwards. Alternatively, the horizontal double lines may route through the incoming signal. As the network must be regularly arranged (for allowing free module relocation) such that all resource slots contain an identical internal logic and routing layout, the horizontal double lines have to be interleaved with $N = 2$. Note that a connection to a vertical pin wire automatically segments the horizontal routing track such that multiple pins may be routed to a set of modules within the same track.

A module connection is provided via a flip-flop connection macro. This flip-flop
3.7 Experimental Results

**Figure 3.69:** Details of the circuit switching network. Some input pins are passed through an input register and directly connected to a vertical pin wire that spans the full height of the reconfigurable area. In columns that do not provide a pin, the equivalent wire is instantiated as a dummy antenna. Two bidirectional routing channels per horizontal row of resource slots may route through the horizontal wires, connect it to a vertical pin wire, or connect it to a flip-flop for interfacing a module.

is either wired to the endpoint of a horizontal double wire routing track, to a middle tapping point of a double line, or to a vertical pin wire\(^9\). Consequently, modules can access a particular horizontal track independent from the horizontal placement position. Both, the pin wire connection to the horizontal routing track and the module connection is set via the configuration interface and controlled by the external host PC.

The whole network can be seen as a large crossbar that is scattered regularly over the resource slots. For minimizing internal fragmentation, the full horizontal logic columns should be actually tiled in three commensurately rows, which results in 256 LUTs per resource slot, when considering the results in Figure 3.3 on page 88 for a communication cost of \(c = 14\) LUTs. However, with this tiling, a slot would be 32 CLBs high, which is not divisible without remainder by six, which is the routing span of a hex line. In other words, this tiling is improper for the proposed two-dimensional circuit switching network implementation. An alternative implementation, using double lines for the vertical pin wires, would straightly allow a tiling in three rows but would substantially rise the propagation delay of the circuit switching network.

**Evaluation of the System**

The example system runs with a clock frequency of 100 MHz on the ESM platform [MTAB07]. This high speed is only possible by passing all inputs and outputs of the communication architecture through pipeline registers within the ReCoBus, the connection bar macros or the input and output terminals of the circuit switching network. For the latter one, the signals at the input pins to the circuit switching network must be synchronized with the system clock for preventing metastability. The

\(^9\) The pin wire connection requires an additional routing segment within the module connection switch matrix on a Xilinx Virtex-FPGA. This is considered in the connection macro design.
vertical routing to the bottom horizontal routing track consumes 6.1 ns out of the 10 ns timing budget when the system is clocked at 100 MHz. This allows a horizontal routing distance of up to ±26 resource slots. For the top row of resource slots, the horizontal routing track is fast enough to distribute a signal from any input pin above the reconfigurable area to any of the 66 horizontal resource slots. Note that longer routing distances or data rates can be achieved by removing the flip-flops from the routing path and by synchronizing the input data stream within the reconfigurable modules.

The following list denotes the logic cost of the implemented communication architecture depicted in Figure 3.64, in detail:

1. Logic within the resource slots: \( R \cdot c = 248 \cdot 14 = 3472 \) LUTs
2. Connection of the ReCoBus macros within the static part costs 528 LUTs in total and is composed of as follows:
   1. for connecting 8 read data and 8 write data signals in each resource slot, 16 flip-flops are instantiated (which is counted as 16 LUTs), multiplied by \( N_h \) and \( R_v = 384 \) LUTs.
   2. For the address signal connection \( 4 \cdot N_h \cdot R_v = 96 \) LUTs are used as the two required connections per ReCoBus macro and horizontally interleaved chains are not shared within the same slice (a slice contains two LUTs and two flip-flops on a Xilinx Virtex-II FPGA). As a half-occupied slices cannot be used to further implement additional logic by the Xilinx tools, this logic overhead was considered twice.
   3. The control signal connection requires \( 12 \cdot R_v = 48 \) LUTs
3. Alignment (de)multiplexers: 272 LUTs that are broken down as follows:
   1. data to static system (see Figure 3.66): \( 8 \cdot 4 \cdot 6 \)-input multiplexers = 128 LUTs (see Table 3.2 for the logic costs of a certain multiplexer),
   2. data from static system: \( 8 \cdot 6 \cdot 4 \)-input multiplexer = 96 LUTs, and
   3. address alignment: \( 6 \cdot 2 \cdot 6 \)-input multiplexer = 48 LUTs.
4. Static connection of the communication bar macros: \( R_v = 4 \cdot [64 \text{ (video)} + 12 \text{ (push buttons)}] = 304 \) LUTs,
5. Multiplexers for the video data routing, as depicted in Figure 3.67: 280 LUTs.

The video routing architecture consists of:
1. video connection bar input multiplexers: \( R_v = 4 \cdot 28 \) (# used wires of the video bar) = 112 4-input multiplexers requiring 224 LUTs and
2. the video output multiplexers with 28 4-input multiplexers, which are implemented with 56 LUTs.
3.7 Experimental Results

- ReCoBus and communication bar glue logic: 154 LUTs.

The total amount of LUTs required in the static part of the system is 1538, thus resulting in 5010 LUTs for the complete communication architecture. This is 7.4% of all available LUTs of the ESM main FPGA. As compared to the results presented in Section 3.7.2, the system presented here requires 40% more logic within the resource slots as it provides an additional address port and an interrupt input. At first glance, the logic cost within the resource slots seem to be unlikely large, but it amortizes easily because of the reduced internal fragmentation. For example, a module with a size of 1000 LUTs can be implemented in 6 resource slots that consume $6 \cdot l = 1152$ LUTs on the ESM main FPGA. [MTAB07] reports an alternative implementation of a one-dimensional reconfigurable system with a resource slot width of 4 CLB logic columns on the same ESM. The resource slots of that system provide 3072 LUTs and the 1000 LUT example module will leave $2072 - 152 = 1920$ more LUTs unused due to internal fragmentation as compared to the example discussed in this section. In this comparison, the waste of logic due to internal fragmentation is reduced by 13.6 times with the help of the ReCoBus communication architecture. Moreover, the vertical tiling further allows a higher utilization of the dedicated memory and multiplier resources.

Competative Analysis of the ReCoBus Architecture

The best competing system, providing an interface to reconfigurable modules via a backplane bus [HKKP07b], is based on a Xilinx Virtex-II-4000-4 FPGA with 16 resource slots that each are 4 CLB columns wide and one-dimensionally aligned. The authors report a resource consumption of 4277 Slices (equal to 8554 LUTs) for the full communication architecture providing a 32-bit master backplane bus. Due to the large resource slot size of 2560 LUTs, that system will be similarly vulnerable to internal fragmentation as the ESM system in [MTAB07]. The communication architecture in [HKKP07b] includes only the backplane bus and is not interleaved. Consequently, the throughput on that bus is rapidly decreasing down to a peak of only 280 MB/s [HKKP07a] for a bus with 16 slots, despite that the bus is equally pipelined as the ReCoBus that achieves 400 MB/s peak. The better performance of the ReCoBus has two main reasons. First, due to interleaving, the routing is carried out via fast hex lines and the signal path of one interleaved chain is only 10 LUTs deep, as compared to 16 LUTs in [HKKP07b]. The second reason is that the ReCoBus is implemented much more densely, hence, reducing the signal propagation delay within the bus.

The here proposed ReCoBus system only provides a slave interface. However, by joining two vertical resource slots together, one ReCoBus macro can be used for connecting data and the other one for connecting the address signals. This would result in about the same logic cost and half the amount of resource slots, but would
allow integrating modules with a master interface into the system. Such a system would then provide a 1.4 times higher throughput, and 8 times more resource slots within the same reconfigurable area, while requiring about half the amount of look-up tables for the implementation as opposed to [HKKP07b].

**Compitative Evaluation of the Connection Bar Architecture**

In [BHB+07], a network for linking video data in a reconfigurable system is proposed that is related to the two-dimensional connection bar example depicted in Figure 3.67. The approach includes multiple one-dimensional resource areas that each include multiple modules in a daisy chained fashion, as shown in Figure 3.9 on page 97. Each one-dimensional resource area contains a feedback path towards the static design, where some multiplexers carry out the routing among the different areas. This routing is similar to the here presented connection bar approach.

However, within a one-dimensional resource area, all signals crossing a module bounding box in [BHB+07] demand connection macros. This results in an overhead of two LUTs per signal wire that pass a module and the overhead is also required for the case when a module is not accessing a particular signal. As [BHB+07] does not consider variable module sizes, a fine grained, two-dimensional resource slot layout (as the one presented in Figure 3.67) would result in an overhead of $2 \cdot (24 + 3) \cdot 248 = 13392$ look-up tables for the video signal example (24-bit RGB plus control signals). These are 20% of all available LUTs within the main FPGA of an ESM. Opposed to this, a communication bar that is not accessed by a particular module is completely transparent to the module interface and does neither include any logic overhead nor demand module modification of the interface for looping unused signals through a module. And even in the case that a module requires read or write access to a signal, one LUT is sufficient per signal wire to connect the upstream together with the downstream at the same time. This results in half the amount of logic cost, as compared to [BHB+07], when connecting a signal within a module. Finally, the huge amount of look-up tables required by [BHB+07] leads either to a very long combinatorial path or to a latency of dozens of clock cycles, if the connection macros are pipelined for increasing throughput.
3.8 Chapter Contributions and Discussion

The ReCoBus and connection bar methodology, presented in this chapter, provides mechanisms for dynamically integrating bus-based communicating modules into a system at runtime. The contributions of this chapter are as follows:

- Overview of state of the art communication techniques, allowing to integrate reconfigurable modules into FPGA-based systems at runtime.

- Investigation of the internal fragmentation due to partitioning the reconfigurable area into resource slots of fixed size [KBT09c].

- Design and analysis of the novel ReCoBus communication architecture that is capable of directly mapping partial master and slave modules into the address space of an embedded system [KHT08, KSHT07, KSHT08a]. This architecture is capable of reducing the resource slot size - and consequently able to increase the amount of slots - by at least an order of magnitude as compared to the best existing approaches. This progress in placement flexibility is achieved at very low logic cost while still allowing high throughput on the bus.

- Extension of the ReCoBus architecture for allowing to place modules in a two-dimensional grid style with even smaller resource slots [KBT09c, KSHT08a]. So far, there is no other bus-based communication architecture presented that is capable to directly connect modules in such manner – despite the necessity to minimize internal fragmentation. Besides enhancing logic utilization, the grid style further allows to use dedicated resource columns, including block RAM and multiplier columns by multiple different modules.

- Design and analysis of the novel connection bar technique for directly linking reconfigurable modules with other parts of the system. This includes I/O pins, modules in the static part, or even other partially reconfigurable modules [KBT09c, KSHT08b]. Connection bars are highly adaptable and can be arbitrarily segmented into multiple sub bars at runtime with the capability of providing point-to-point connections as well as one-to-many multicast streams. The connection bar approach was further extended for grid style systems and allows high throughput links, even if modules are placed in two dimensions on the FPGA.

- Design of a two dimensional circuit switching network by reserving a set of wires for carrying out changeable module connections directly within the routing fabric of an FPGA at runtime. This technique builds a logical routing layer on top of the FPGA and it is the only available two-dimensional routing architecture allowing a runtime system to freely route signals throughout the recon-
figurable modules. Consequently, the reconfigurable modules are not obstacles for the proposed circuit switching network.

- Experimental evaluation of the proposed communication architecture for determining its latency and its logic cost [KHT08]. Furthermore, the impact of bounding modules into fixed regions and restricting the routing on the physical implementation was examined. For the ESM platform, it was proven that hundreds of wires can be reserved for implementing a communication architecture for including partially reconfigurable modules into a system at runtime, before noticeably reducing the achievable clock frequency or the routability of the partial modules [KBT09a].

- Implementation of the ReCoBus architecture on low cost, blockwise reconfigurable FPGAs, such as the members of the Xilinx Spartan-3 family [KBT09a]. This paves the way for exploiting runtime reconfiguration in very cost sensitive applications. With the extensions of the ReCoBus architecture for supporting glitch-free reconfiguration on Spartan-3 FPGAs, it was possible to provide multiple consecutive resource slots for the first time on these devices with the capability of swapping modules without interrupting communication with modules not involved in the reconfiguration process.

So far, partial runtime reconfiguration as an integral part of an FPGA-based system has not remarkably found its way into commercial applications. Moreover, this has not been widely exploited despite its potential benefit, including cost and static power reduction (that is related to the FPGA size). This has two major reasons. Firstly, the overhead due to internal and external fragmentation and for connecting reconfigurable modules with the static part of the system is mostly that high when using other approaches that typically no overall benefit may be achieved (see also Figure 3.4).

Secondly, present tools and design methodologies have not reached a sufficient stable state and designing partially reconfigurable systems is an error-prone process. The first issue is tackled with the ReCoBus and connection bar approaches that limits the worst case resource overhead per reconfigurable module to a few hundred look-up tables. This is in most practical cases more than an order of magnitude better than existing approaches presented for Xilinx FPGAs. The fine-grained placement grid is further helping for reducing external fragmentation. The tool and design issues will be considered in the next chapter.
4. Building Partially Reconfigurable Systems – Methods and Tools

This chapter covers all components required to specify, verify, and implement runtime reconfigurable systems based on the ReCoBus and connection bar communication architecture. This includes offline design aspects, as well as functionality required to load partial modules onto an FPGA at runtime. The goal of this chapter is to enhance a typical system design flow in such a way that fractions of the hardware (implemented on an FPGA) may be exchanged by partial reconfiguration. This comprises technical aspects that are required to provide the communication to reconfigurable regions on an FPGA, as well as controlling and managing the reconfiguration of the modules at runtime. This thesis puts focus on highly optimizing these components and integrating them into a framework, such that sophisticated reconfigurable systems can be built easier and faster, with a higher level of flexibility, and with the lowest possible overhead in terms of logic resources and reconfiguration time, as compared to many pioneering and related approaches in this field.

Throughout this chapter, the following key components for building partially runtime reconfigurable systems will be examined in detail:

**Designflow and Tools:** Designing runtime reconfigurable systems introduces new aspects into traditional FPGA design that demand a suitable designflow to deal with the increased complexity. The flow should provide a high level of abstraction with the help of tools for automating the FPGA device specific low level implementation aspects. This includes the specification and synthesis of the communication architecture that is responsible for integrating the reconfigurable modules into the system, as well as guiding through all physical design aspects. This includes floorplanning, constraining the place and route process, and assembling the bitstream.

**Reconfiguration Control and Management:** The module and resource management is responsible for deciding where and when a reconfigurable module is loaded to the FPGA device. Therefore, this component includes a placer and a scheduler that is typically provided as an operating system service. The reconfiguration control comprises all technical aspects, required for loading
4. Building Partially Reconfigurable Systems – Methods and Tools

Partial configuration bitstreams to a specific area on an FPGA. This implies online modifications to the configuration that is stored on the device or sent to the FPGA (e.g., for modifying fractions of the routing at runtime).

**High-speed Reconfiguration:** The reconfiguration speed has a major impact on the feasibility of applying runtime reconfiguration within a system. For example, it is only possible to allow redundant cold spare hardware modules in a fault tolerant system, if the entire reconfiguration time is lower than the allowed time to repair. Or reconfiguring compute kernels in a multimedia signal processing system will also demand to accomplish strict real-time constraints. For addressing configuration speed issues, configuration prefetching, bitstream decompression, and defragmentation techniques are presented.

With these components, it is possible to enhance FPGA based systems with the capability of partial runtime reconfiguration. The reconfiguration capabilities are closely related to the target FPGA features. This means that if a system must provide partial reconfiguration, also the FPGA must allow exchanging only portions of the currently loaded configuration. And if the system should continue execution during reconfiguration, which is mandatory in the case of partial *self reconfiguration*, this must be supported by the target FPGA architecture as well. However, as revealed in [KT04], a system may compose a new complete bitstream by linking together multiple module configurations that are then loaded as a *full configuration* to the device. As this method is different to the one used for active partial reconfiguration, this technique is called *pseudo partial reconfiguration*.

The remainder of this chapter reveals new contributions on the offline and online aspects of implementing partial reconfigurable systems on FPGAs, capable to exchange fractions of the configuration, while keeping the rest of the system active. In particular, focus is put on tools and online techniques supporting the ESM platform [MTAB07] that has been especially designed as a testbed for the research on partial runtime reconfiguration. However, other systems that may be designed around completely different FPGA families have been considered as well. Consequently, this chapter provides considerations on building reconfigurable computing system in general.

### 4.1 A Framework for Component-Based Reconfigurable System Design

The goal of this section is to expand typical FPGA design flows for implementing complex runtime reconfigurable systems. For enhancing design productivity while reducing error-prone manual design steps, this expansion is highly automated and supported with novel tools as illustrated in Figure 4.1. These tools have been seamlessly integrated into the Xilinx ISE design suite, resulting in a powerful framework
for easily designing runtime reconfigurable systems for Xilinx FPGAs. This framework is targeted on the design, test, synthesis, and bitstream assembly of complex reconfigurable systems on a high level of abstraction. In particular, the following additional steps, as compared to completely static only systems, have to be accomplished in this framework (see also Figure 4.1):

1. Static/dynamic partitioning for splitting the system into a static part and a set of dynamically reconfigurable modules.

2. Specifying the communication architecture that has to provide all communication demands of the reconfigurable modules. The partial modules will be completely encapsulated and any connections to other parts of the system have to be accomplished via this architecture.

3. Extending the simulation for simulating runtime reconfiguration.

4. After passing the functional simulation, the physical implementation can be carried out. This requires:
   
   a) Budgeting the resource requirements of the static system as well as for the individual partial modules.

   b) This information is then used for floorplanning the FPGA. Here, the device is partitioned into a fixed static region and an area, tiled into resource slots, for accommodating partial modules.

   c) In addition, this step requires to freeze the placement of the communication architecture that is implemented with the help of dedicated communication macros, located at fixed positions.

   d) With this information, module bounding boxes and corresponding module origins can be defined in terms of resource slots.

   e) Next, place and route will be performed separately for the static system and all partial modules according to the placement constraints specified during floorplanning.

5. Finally, the configuration bitstreams for the static system as well as for all reconfigurable modules are generated. For the latter ones, the partial module bitfiles have to be extracted from the full bitstreams.

6. If required, an initial bitstream may be composed out of the static bitstream and the partial modules.

The static/dynamic partitioning has to be done manually by the designer, as it is in general impossible to automatically derive information about mutually exclusive function blocks from a system. However, as these function blocks are typically encapsulated within separate branches of the design hierarchy, this partitioning step is
Figure 4.1: Design flow for building reconfigurable systems with the framework ReCoBus-Builder. The system designer has to partition the system into a static part and the dynamically reconfigurable modules. According to the port requirements, the communication architecture will be specified in the following. After passing simulation, the physical implementation will be carried out respecting area and location constraints, determined in the floorplanning step. Finally, the static and partial modules bitstreams are assembled.
4.1 A Framework for Component-Based Reconfigurable System Design

Figure 4.2: The communication architecture has to provide the port requirements $P_\cup$ for all partially reconfigurable modules $\mathcal{M}$. In this example, three modules ($m_{crypto}$, $m_{video}$, and $m_{audio}$) require a bus and a video connection as well as an audio I/O interface, which has to be implemented with ReCoBus and connection bar macros.

very easy to perform. The result of the static/dynamic partitioning is a static design, typically containing all modules that are required at any time, for example, a CPU, the memory controller, the configuration interface logic, and dedicated I/O interface modules. Beyond the static design, the partitioning will define the total set of partially reconfigurable modules $\mathcal{M}$. All reconfigurable modules and the components of the static system possess a port and the full set of ports is $P$.

Each module $m_j \in \mathcal{M}$ may possess different port requirements and to each module $m_j$ a set $P_j \in P$ is associated. The communication architecture has to fulfill all these port requirements and must provide at least the union of all requirements: $P_\cup = \bigcup P_j, \forall j = \{1, \ldots, |\mathcal{M}|\}$, as exemplarily depicted in Figure 4.2. This port set has to be provided by ReCoBus and connection bar macros. With these two types of macros, virtually all practical communication demands can be fulfilled. The macros are instantiated at the systems level as an interface module, like for example, a bus. The designer has to identify the port requirements and to specify the protocol on the physical layer for the macros. For connection bars, this involves a specification of the number and the direction of the signal wires and if the wire connections include flip-flops at the macro inputs, the outputs, or at the module side. For a ReCoBus, this is similar, but here, not only the direction has to be specified but also if a particular wire is dedicated or shared among multiple modules (see also Section 3.5). Furthermore, the maximum number of simultaneously loaded master and slave modules can be specified, as well as the signal leveling of the reset and control signals. Note that only the physical aspects (mainly in terms of wires and registers) are specified, hence, being virtually compatible with any protocol implemented on top of the communication architecture, for instance, a Wishbone or CoreConnect bus protocol. A physical specification of a ReCoBus or some connection bars can be reused by multiple designs. Such specifications are independent to the target FPGA, and the same specification may be used to build systems for different FPGA families. Currently, the framework supports all Xilinx Virtex-II/II Pro FPGAs as well as all Spartan-3 de-
4. Building Partially Reconfigurable Systems – Methods and Tools

The ReCoBus specification process itself is supported by a GUI inside ReCoBus-Builder tool and the process can be carried out within a few minutes.

The here considered systems are strictly modularized and allow dynamically exchanging hardware in a way that exceeds the capabilities of standard ASICs or static only FPGA designs. Consequently, this demands additional tools and techniques for implementing such systems. The following three sections reveal novel contributions for simulating (Section 4.1.1) and for physically implementing (Section 4.1.2) runtime reconfigurable systems, as well as a innovative methodology for composing partial configuration bitstreams by hardware linking (Section 4.1.3).

4.1.1 Simulating Run-time Reconfiguration

An essential requirement to correctly build any type of complex digital system is the capability to simulate it before implementing the actual design. This is required for debugging purposes, performance analysis and all kinds of diagnosis. For example, for verifying arbitration or handshaking mechanisms during swapping a reconfigurable module, it is necessary to simulate the reconfiguration process on a cycle-accurate base with the rest of the system.

In order to find errors, resulting from the integration of a partial module into a system, it is necessary to simulate the system at the register transfer level on a true cycle-accurate model that behaves exactly like the real model running in the FPGA. For instance, this allows verifying if a module follows a bus protocol during and after a configuration process.

ReCoBus Simulation

The ReCoBus communication macros allow plugging-in or disconnecting modules by partial reconfiguration such that the rest of the system can continue its operation. And even more important, modules may be swapped without influencing running transfers on the ReCoBus at any time. Consequently, the reconfiguration process itself is completely transparent to the reconfigurable system. This property is exploited for carrying out cycle-accurate simulation for ReCoBus-based reconfigurable systems. As the reconfiguration is a completely deterministic process\(^1\) that is transparent to the runtime system when using the ReCoBus communication architecture, it is possible to activate/deactivate modules, without considering the entire configuration data itself. This helps to enormously simplify the reconfiguration process.

\(^1\)For Xilinx Virtex-II FPGAs, this holds true for a configuration speed of up to 50 MHz [CMZS07]. Beyond this speed, the internal FPGA configuration state machine may request some unpredictable wait states.
4.1 A Framework for Component-Based Reconfigurable System Design

and it is sufficient to control the activation of the partial module instances without considering all internals of the FPGA fabric, as shown in Figure 4.3.

Here, a module, called *partial module activator*, senses the data sent to the configuration interface (typically the internal configuration access port ICAP). This data stimulates a state machine that stores for each slot if it is idle or accommodating a particular module type. In the latter case, the system will further remember the instance that is executing the partial module for each slot. This information is evaluated for setting the activation of the instances. Within the simulation model, a ReCoBus can connect any number of module instances in parallel and the activator will reflect the resource limitations of the runtime system. All ReCoBus sockets are basically identical. However, if the ReCoBus is interleaved, the interleaved signals will pass an alignment multiplexer/demultiplexer pair for each ReCoBus socket. The alignment is controlled by the activator that stores the start slot position. The designer has to specify the maximum amount of simultaneously integrable instances for each module. During simulation, the first available instance of the requested module will be activated. If a module is deactivated, its inputs and outputs are forced to an undefined state. This allows determining if partial modules consume random data or if the static system tries to read data from a terminated module instance during reconfiguration.
4. Building Partially Reconfigurable Systems – Methods and Tools

Connection Bar Simulation

Connection bars are simulated by a demultiplexer/multiplexer pair for each logic resource slot. This allows connecting the connection bar input data either to the next consecutive slot or to any of the partial module simulation instances. Note that in the case the connection bar is interleaved, not the next but a slot that is located \( N \) positions further will be connected.

Controlling the Module Activation

Within the activator, the particular state of the resource slots and consequently the activation of the particular modules is set according to the amount of bitstream data written to the configuration interface. For Xilinx FPGAs, the configurations are written in a frame by frame manner where a frame corresponds to the smallest atomic piece of configuration data that can be accessed in the FPGA fabric. For configuring a complete logic column (including the routing), an FPGA family-specific amount of frames has to be written to the device that is 21 frames in the case of Virtex-II FPGAs. The size of a frame depends on the height of the device in terms of basic elements (CLBs). A Virtex-II XC2V6000 FPGA (the main FPGA of an ESM) holds a frame size of 984 bytes (2 \( \cdot \) 12 bytes for the top and bottom I/O tiles, plus 96 \( \cdot \) 10 bytes for the CLBs). A configuration bitstream may include some header information that will be ignored by the FPGA configuration state machine. This is used to simplify the module identification in the activator by placing a module identifier in the bitstream header. The following lines denote a header example of a partial ESM bitstream:

0000: AD D0 00 02 // add instance of module 02
0004: AA 99 55 66 // start FPGA configuration
0008: 30 00 80 01 // reset CRC register
000C: 00 00 00 07 // reset CRC register
0010: 30 00 20 01 // write column address register
0014: 00 0C 00 00 // start column address is 6
0018: 30 00 40 F6 // write configuration command
001C: 00 00 00 00 // first frame data

When configuring the FPGA, the data of a full frame is at first shifted into an internal shadow register. From this register, a copy is stored in an additional hold register after finishing one frame. This data is then transferred to the logic cells on the arrival of a new frame in the hold register. Consequently, an extra dummy frame has to be written for flushing the last frame from the hold register to the logic cells. This is the procedure following the documentation [Inc07]. A slightly faster alternative was proposed by Claus et al. in [CMS06]. In that work, flushing the last frame is triggered by applying a so-called multiple frame write command instead of sending an extra dummy frame to the FPGA. While this technique is very advantageous on writing single frames in a random fashion, the benefit is marginal when reconfiguring larger
4.1 A Framework for Component-Based Reconfigurable System Design

For a Xilinx Virtex-II XC2V6000 device, the standard procedure results in an idle offset of 984 frame data write cycles for filling the configuration pipeline. After this, a new column configuration is accomplished after \(21 \cdot 984 = 20664\) write cycles. In other words, out of the complete bitstream information, the configuration activator will only interpret the beginning module identifier and the start column address. Thereby, the start column address is set by the placer and is determined by an operating system service in software within the run-time system. All further reconfiguration effects are now controlled by counters that are triggered by writing data to the configuration port. The simulation model requires some module specific parameters, including the module size, the ReCoBus interface width, and relative location information about the connection bars. This information has to be annotated to the simulation model for each module individually. With this approach, a cycle accurate simulation of the reconfiguration process is accomplished and a reconfigurable module, activated in the simulation, behaves exactly like the real module after a partial reconfiguration process. The activation time of a module is set to the point of time when the real system will finish the configuration. The simulation model will generate a module reset itself after activation, like the real module running on an FPGA (as described in Section 3.5.2). This reset is released by the system level reconfiguration step that sets the base address of the last reconfigured module (see also Section 3.5.2).

As the configuration data content is completely ignored, simulating runtime reconfiguration can be carried out with the help of placeholder modules, long before physically implementing a module. In this case, the module size has to be estimated and a bitstream containing dummy data will be sent to the module activator.

4.1.2 ReCoBus-Builder: a Builder for Reconfigurable Systems

The central tool of the proposed framework for building complex partially reconfigurable systems is the ReCoBus-Builder. As depicted in Figure 4.1, the tool is assisting and automating all further steps required for physically implementing reconfigurable systems. ReCoBus-Builder provides a comfortable GUI (see Figure 4.4) that helps in specifying and synthesizing a communication architecture fulfilling the requirements for all reconfigurable modules. The tool is seamlessly integrated into the Xilinx ISE design flow and helps floorplanning the FPGA. The coupling to the Xilinx vendor tools is achieved by synthesizing ReCoBus and connection bar communication macros that can be directly instantiated inside a VHDL specification of the system. The tool supports this process with an included template code generator. The floorplanning leads to a set of location constraints as well as bounding box and prohibit region definitions. ReCoBus-Builder is capable of compiling these specifications into constraints that guide all place and route processes. Furthermore, the tool
can generate so-called *blocker macros* that allocate all logic and an adjustable set of routing resources within user definable regions of the FPGA. These macros circumvent a lack in the current versions of Xilinx’s router that is not strictly following bounding box definitions.

**Floorplanning**

While static-only FPGA-based systems may be implemented without additional area constraints in a completely automatic fashion with the help of a placer tool, floorplanning is essential when designing runtime reconfigurable systems. Here, all partial modules have to be constrained into bounding boxes and during the implementation of the static part of the system, the assigned reconfigurable area must be prohibited for implementing any static logic or routing. This is illustrated in Figure 4.4.

Floorplanning typically starts with a preliminary synthesis run for budgeting the resource requirements of the static system and of all reconfigurable modules individually. Based on this initial resource budgeting, the target FPGA can be selected. On this target, one or more reconfigurable areas have to be defined that will be tiled into a grid of resource slots. In this step, the system is split into a static part and a dynamically reconfigurable part containing the FPGA resources that are shared by multiple modules over time. These slots will accommodate the partial modules and the grid must provide sufficient resources to fulfill all logic (look-up tables) and memory (block RAMs) requirements. In case that a ReCoNode is designed to be able to take over some hardware tasks from another node, for example, upon a node failure, these additional resource requirements have to be considered at this point. Another design decision is the tile grid as discussed in Chapter 3. The grid must be balanced between the communication cost and the overhead for internal fragmentation. For a given set of modules, this can be computed using Equation (3.1).

After tiling the reconfigurable area into a linear array or a grid of resource slots, the communication architecture must be set to fixed positions such that the infrastructure uses the same logic and routing resources at always the same relative position within the slots. This enables relocating modules to different placement positions at runtime or when assembling the initial system. The communication architecture will be implemented by fully placed and routed macros that are only instantiated and that maintain their internal structure during place and route of the static system or the partial modules. This allows that the individual reconfigurable modules can be also implemented in individual synthesis processes. These macros will then force the place and route steps to use exactly the same resources for the communication across the module interface. Note that the design process of different modules can be easily delegated to multiple design teams that can work in parallel.

With the tool PlanAhead [Inc08c], Xilinx offers a vendor tool that also guides designers during floorplanning for partially reconfigurable systems. However, PlanAhead (in combination with the present partial design flow [LBM⁺06]) can only as-
4.1 A Framework for Component-Based Reconfigurable System Design

System Specification
(Communication Architecture & Floorplan)

Figure 4.4: ReCoBus-Builder GUI. The tool assists in specifying the complete ReCoBus and connection bar communication architecture. In addition, the tool provides a geometrical view on the target FPGA for floorplanning the design. This includes the definition and tiling of the reconfigurable region, placement of the communication infrastructure, and adjusting module bounding boxes. The synthesis of the communication architecture is a push-button process and ReCoBus-Builder generates all instantiation templates and constraints required for easily implementing the system with the Xilinx ISE design suite. This results in a static system and a module repository. Finally, the tool assists in the integration of partial modules by bitstream linking (see also Section 4.1.3).
Figure 4.5: Enhancing relocatability for modules including static routing. a) Module $m_1$ cannot be swapped with module $m_2$ as this will cut the static routes through the reconfiguration island. b) by merging the static routes of both islands that are then c) included into all reconfigurable modules, relocation becomes possible. d) However, merging may fail on wire conflicts, as a wire track can only implement one static routing path through the reconfigurable area.

Routing Constraints

While the Xilinx vendor placer properly follows area constraints, this is not strictly performed for the routing. For the standard partial design flow proposed by Xilinx [LBM+06], this has no side effects, because partial modules are always placed to exactly the same reconfigurable island as illustrated in Figure 4.5a). However, this causes problems if modules have to be relocated to different positions on the FPGA. This could be solved by merging routing violations from multiple reconfigurable islands into one island that is then used for implementing the reconfigurable modules as depicted in Figure 4.5b) and c). This is not applicable for systems with plenty of resource slots and the merging may fail as shown in Figure 4.5d).

The name *bus macro* is misleading, as these macros provide only point-to-point links. In this thesis the phrase *ReCoBus* is used for a monolithic macro containing a real bus with connections to multiple (reconfigurable) modules at the same time. In addition, the flexible technique providing multiple point-to-point connections will be named with the phrase *connection bar*.
Figure 4.6: Blocker macro generation for constraining the routing. a) On top of the communication architecture, b) blocker macros can be defined and c) automatically generated that will wall the module routing into bounding boxes. Note that the blocker uses resources that are disjoint to the communication architecture as depicted in the enlarged boxes. d) Blocking is also used for defining prohibited routing regions which is required when implementing the static part of the system (see also Figure 4.4).
As a workaround, ReCoBus-Builder can generate so-called blocker macros that occupy all logic and routing resources within a user defineable region of the FPGA. Blocking can be performed for all CLB tiles as well as for block RAM or I/O tiles, as depicted in Figure 4.6. This process incorporates that the logic and routing resources of the communication architecture have to be skipped by the blocker. The blocker includes antennas into the switch matrices such that all outgoing wires will be occupied with such antennas. Consequently, the Xilinx vendor router will be forced to use alternative wire segments in unblocked switch matrices. The blocking antennas demand a logic tile (e.g., a look-up table) driving such an antenna to be compatible with the router. As the ReCoBus uses logic tiles that are then not available for blocking, fully blocking all outputs may be impossible by only using resources of a particular switch matrix. ReCoBus-Builder solves this issue with a blocking algorithm that can flood switch matrix outputs via neighboring CLBs. The designer can select within ReCoBus-Builder if any combination of 1.) local interconnect lines, 2. horizontal long lines, or 3.) vertical long lines should be blocked or left for implementing the static or module routing. For example, this allows using horizontal long lines for linking some static signals across the reconfigurable area. Independently, vertical long lines may carry out the routing inside modules that are multiple resource slots high.

As illustrated in Figure 4.4, the router must be prevented from using routing resources within a reconfigurable region during the implementation of the static part of the system. This is implemented by instantiating look-up tables that drive blocker antennas. In addition, these look-up tables will be connected to one or more global clock trees. This activates all necessary clock drivers within the reconfigurable region. The configuration of the clock drivers is stored in a dedicated part within the initial bitstream. The configuration of the clock network is not touched at runtime and the blockers will consequently ensure that the partial modules will be clocked, regardless of the placement position within the partial region.

The blockers are included after synthesis and placement but before the final routing step. This is achieved by modifying the netlist after the placement with the help of the XDL language. The process is fully automated and carried out in seconds. Note that blocking is not limited to rectangular regions and bounding boxes may be defined possessing any shape. Furthermore, modules are not necessarily required to be bound to a single continuous area, as shown in Figure 3.65 on page 188.

ReCoBus Generation

After specifying the system specific and communication related parameters, ReCoBus-Builder generates a so-called hard macro, containing the complete logic and routing of the ReCoBus that provides the communication between the static part of the system and the reconfigurable modules. Once generated, hard macros will not require any logic synthesis and they will be placed on the FPGA prior to the final
place and route step. In this step, the connections between the monolithic ReCoBus hard macro and the rest of the design will be accomplished. The complete hard macro is generated using the Xilinx Design Language (XDL) that gives designers, among other lower level details, full access to all of the FPGA logic resources (e.g., the look-up tables and routing wires). In addition, FPGA device descriptions, containing all device specific information, can be generated in the XDL language format. This information permits building device specific ReCoBuses as dedicated monolithic hard macros. For instance, this allows automatically deriving the positions of the block RAM columns in order to generate regular structured hard macros even if the reconfigurable area contains such dedicated resources. Note that partially reconfigurable modules may use such RAM columns. Another example for device specific information that can be derived from the XDL device description, is the set of available routing wires provided by a particular FPGA. It was found that available routing wires vary even within the same FPGA family. For instance, the largest Xilinx Spartan-3-FPGA (XC3S5000) provides so-called 'hex lines' within all CLBs that are not available in the CLBs of the smallest device (XC3S50) within the same device family. ReCoBus-Builder considers all this information in order to build an optimized monolithic ReCoBus hard macro.

A ReCoBus macro is considered to be optimized if it uses the minimum amount of slices within the resource slots that are connected with minimum routing cost. Thus ReCoBus-Builder packs as much logic inside the slices as possible to provide the bus logic and the connection to the reconfigurable modules. For instance, the two look-up tables inside one slice will be used for implementing the logic for two read multiplexer chains (see also Section 3.5.3). Independently, the two additional available flip-flops will provide a connection for two write data signals from the static part of the system to the reconfigurable modules connected in the same slot. With this packing, double the amount of signals per slice, as compared to the slice macros proposed in the Xilinx partial design flow [LBM+06], can be provided.

A ReCoBus is built by first placing the slices according to the desired resource slot width and specified density within the area assigned for the resource slots. The slices are placed in a regular fashion along the resource slots such that their relative positions are the same for all slots. Then, a breadth-first search is carried out for finding the routing paths between two consecutive resource slots and the internal routing within a resource slot. The search will continue in order to find all possible paths that require the minimum number of wires (hops) between the allocated slices. Out of all found paths, the one leading to the lowest propagation delay will be selected. In case that multiple paths have the same delay, the one containing wires providing the lowest connection grade will be chosen. This leaves the wires with higher connection grade for implementing the partially reconfigurable modules. ReCoBus-Builder is capable to solve routing conflicts that may occur if the ReCoBus is densely built. The paths found between the first two resource slots and inside the first resource slot are then copied to all other resource slots in order to provide a homogenous routing
over all resource slots and the borders to the static part of the system. After this, the slice logic is set according to the determined routing. This includes a shuffling of the look-up table inputs according to the found routing tracks. Finally, a macro interface for connecting the static part of the system is generated. The complete ReCoBus hard macro generation runs automatically as a push button process.

Most commonly, the ReCoBus will map slaves directly into the system address space and in the case of masters, the ReCoBus will allow these modules to directly access the address space. In the case of multiple masters, an arbiter within the static part of the system is required that directs the access to the bus. ReCoBus-Builder does not generate an arbiter together with the ReCoBus, but it can provide all dedicated signals between multiple masters and an arbiter. Consequently, any policy for granting masters access to the bus can be implemented.

**Connection Bar Generation**

Besides the ReCoBus hard macro, ReCoBus-Builder assists in specifying the routing and placement of the connection bars. Multiple bars can be included into the system at different horizontally aligned positions that are located parallel to the ReCoBus as depicted in Figure 3.41. It can be selected if and at which position a connection bar has to provide a module connection point. Connection points are only required for modules that demand access to a particular bar. If no bar connection is required, no logic resources will be wasted. Each bar is exported as an individual hard macro, as described in the last paragraph for the ReCoBus. In each resource slot, the ReCoBus-Builder tool ensures that the routing is carried out by using the same equivalent wire across the slot boundary, regardless of whether a connection bar contains a connection or not. Consequently, modules can be placed freely within the reconfigurable resource area.

**Timing Specification**

Apart from constraining the physical wires, the communication architecture is further defining the timing behavior of the ReCoBus and the connection bars. This allows generating the different partial modules completely independently from the static part of the system. Accordingly, the static part can be implemented independent from all reconfigurable modules. This allows design iterations without interfering with other parts of the system. The static and the module timing can be decoupled by placing flip-flops between all signals that cross partial module boundaries or connect to the static system. Alternatively, the Xilinx vendor place and route tool allows defining slack offsets for a routing net. With this constraint, the total available slack (typically the reciprocal of the clock frequency) can be split into a fraction for the static system and another fraction for the partial modules. During the place and route phase of the partial modules, they will be implemented at the most critical position that is furthest
4.1 A Framework for Component-Based Reconfigurable System Design

away from the static part of the system. This ensures a valid timing when placing the module to any position within the reconfigurable area.

4.1.3 Bitstream Assembly and Hardware Linking

ReCoBus-Builder assists in forcing modules into bounding boxes, as well as defining and implementing swappable interfaces including their timing. Thus, partial modules can be cropped out of the complete bitstream for linking them together to an initial system or for loading them to an FPGA at runtime with the help of partial runtime reconfiguration. For this purpose, a bitstream linker was developed that can compose together partial modules in a two-dimensional fashion. This linker provides all functionality required to produce a module repository and an initial system at design time. The capabilities of the bitstream linker are as follows:

• Extraction of modules from a full bitstream with the capability to extract any logic tile, BRAM/multiplier tile, the BRAM content, and the I/O tiles. The latter one is in particular required for implementing partial systems on an ESM platform [MTAB07] where reconfigurable modules include external pins that are directly connected via an additional external crossbar to some external peripherals.

• Merging of extracted module bitstreams into a full bitstream where the target placement position can be selected in a two-dimensional fashion. The linker will check if the module resource requirements match the provided resources within the target region. Besides generating full bitstreams for an initial system, it is alternatively possible to generate partial bitstreams that contain only the configuration data for a particular module.

• Modifying compare values inside the reconfigurable select generators that decode dedicated signals for the reconfigurable modules inside the ReCoBus hard macro. This includes select and grant signals. With this feature, address ranges can be assigned for each module instance separately.

• Analyzing or remapping of clock domains within a definable region. The linker can scan a region for all used clock domains (global and local clock nets). Additionally, all clock input multiplexers of the FPGA fabric can be adjusted to any one of the global clock nets. This is helpful when porting a partial module from one design to another or for adapting the speed of a module, and hence, the power consumption.

• Adjusting connection bars for selecting within each resource slot to either bypass or connect a data stream of a particular connection bar. In detail, the output multiplexer, selecting between the preceding resource slot or the module output, can be adjusted.
Compared to related approaches [BJRK$^+$03, HLK02, KJdlTR05, BBH$^+$06, MMP$^+$03], the bitstream linker is not only capable of extracting and merging configuration bitstreams but also able of performing small bitstream manipulations for exchanging look-up table functions as well as modifying the routing. The extraction and merging allows two-dimensional module relocation (as opposed to [HLK02, MMP$^+$03]) and supports all logic, dedicated memory, and I/O cells. JBits [GLS99], providing a library of Java classes for manipulating configuration bitstreams, includes all functions required for bitstream linking. However, JBits supports only Xilinx Virtex-II devices and no Virtex-II Pro or Spartan-3 FPGAs. The developed bitstream linker supports all these devices and the binary code of the runtime bitstream linker requires less than 10 KB when implemented on a ReCoNode. When alternatively using JBits, a java runtime environment will be required.

Hardware linking may be carried out alternatively on the netlist level. This may be helpful for debugging the system, but is inapplicable for runtime reconfigurable systems, as the configuration bitstream generation from a netlist is unlikely to be performed at runtime. Therefore, this thesis puts focus on hardware linking based on configuration bitstreams.

4.1.4 Core-Based System Design

As the communication infrastructure requires only low resource overhead by providing high flexibility and low latency, the ReCoBus-Builder tool flow is also suitable to implement static-only systems. As the interface of the partial modules is completely encapsulated, a re-synthesis step of a particular module will not influence the timing of other parts of the system, and consequently, does not require a timing re-verification of the untouched parts. In addition, this tool flow allows to parallelize the synthesis as well as the place and route process. Furthermore, the partitioned design flow may help to massively reduce the memory consumption of the design tools, because focus has to be put only on fractions of the design. If a module is once encapsulated and fitted into a bounding box, it can be integrated into a system by instantiating the module as a fully routed hard-IP core.

Gupta et al. postulated in [GZ97] to compose on-chip systems with the help of hard-IP cores. In that work, focus is put on ASIC design and the motivation for the core-based system design is identical to designing static systems based on the ReCoBus-Builder tool flow. In both cases (ASIC and FPGA), the system is composed of fully routed modules that provide a standardized interface for carrying out the top level communication. Modules are verified against this standardized interface, hence simplifying the integration of such IP cores into a system. This allows to skip costly verification processes and enhances design productivity. While in the ASIC case, dedicated metal layers can be used for wiring the top level modules, routing resources have to be preserved in the FPGA case. These resources form a logical layer on top of the FPGA routing fabric that allows to link modules without
4.2 Run-time Reconfiguration

The last section has covered aspects on building reconfigurable systems, including specification, simulation, floorplanning, and partial bitstream generation that have to be considered at design time. This section puts focus on run-time aspects. When fractions of an FPGA-based system have to be exchanged, the configuration stored inside the FPGA has to be exchanged as well. This is carried out by sending a configuration bitstream to the FPGA that updates only the changed parts inside the FPGA fabric. This process is related to the used FPGA architecture that defines how configuration data can be accessed and also how the FPGA primitives are encoded inside the configuration bitstream. For instance, while the configuration of Atmel AT40KAL FPGAs can be performed via an SRAM memory like interface [Inc03] in a very fine grained manner in portions of 8 bit, Xilinx Virtex-II FPGAs can only be configured in frames. Such a frame stores a fraction of the configuration data for a full logic or memory column of the FPGA including the attached top and bottom I/O cells. Consequently, the frame size depends on the device height and is, for example, 7872
4. Building Partially Reconfigurable Systems – Methods and Tools

bit in the case of a Virtex-II-6000 FPGA. This amount of data is the smallest atomic piece of configuration data that can be individually accessed on these devices. As most of the practical work of this thesis is based on Xilinx Virtex-II FPGAs, focus will be put on this device family. The following paragraphs reveal more details on the configuration bitstream format and the required manipulations.

4.2.1 Horizontal Bitstream Relocation

In reconfigurable systems, providing a very flexible module placement, it is inapplicable to store alternatives for all possible positions inside a module repository of the run-time system. Consequently, a partial module bitstream should be able to run at multiple possible placement positions. For relocating modules in horizontal direction on a Virtex-II FPGA, the start position of a logic or RAM column has to be adjusted inside the configuration bitstream that is depicted in Figure 4.7. As it is possible to write the module configuration data as one single block\(^3\), it is sufficient to set the start position once at the beginning of the configuration process. This can be easily carried out in software. Alternatively, it is possible to manipulate the module start position within the bitstream in hardware. This approach was firstly presented by Kalte et al. for Xilinx Virtex-E FPGAs in [KLPR05] and later for Virtex-II / II Pro FPGAs in [KP06]. Manipulating bitstreams in hardware will not show an advantage as compared to a manipulation in software, if modules densely fit into resource slots, which is possible when using a fine grained resource slot tile grid. The reason for this is that for a densely packed module, typically all configuration frames within the area of a swapped module have to be written to the FPGA. Consequently, it is sufficient to set the bitstream position once at the beginning of the configuration process. The partial bitstream generation, presented in Section 4.1.3 will ensure that all columns of a module are included inside the resulting partial module bitfile.

For one-dimensional slot based reconfigurable systems, based on Xilinx Virtex-II FPGAs, such as the ESM [MTAB07], a horizontal bitstream relocation scheme is sufficient and a partial module bitfile can be directly written to the target position on the FPGA. This requires that modules span the full height of the device including the I/O pins located above or below a particular module. This typically prevents using these I/O pins and all communication has to be carried out with I/O pins located at the left or right border of the device, or I/O pins that are not located within the reconfigurable region of the system. Consequently, designing slot based reconfigurable systems does also include considerations for the PCB (printed circuit board). An outstanding platform dealing with these issues is the ESM [MTAB07] that actually provides an external crossbar switch for allowing partial modules to connect to ex-

\(^3\)This holds true for consecutive modules that contain no block RAM or dedicated multiplier resources. For non-consecutive modules (see Figure 3.65 for an example) and for dedicated resources, multiple blocks, each containing another start position, have to be written to the FPGA.
Figure 4.7: Configuration bitstream format for Xilinx FPGAs. The configuration state machine scans the input data for a synchronisation (SYNC) sequence. Any preceding optional user data before this sequence will be ignored. After this, some initialisation commands follow. Next, the frame start position (POS), the size of the configuration data, and the configuration data itself is send to the FPGA. The last three steps may be repeated arbitrary times. At the end further commands are required for activating the configuration or for finishing the configuration process. The frame start position is automatically incremented if multiple consecutive frames are written to the FPGA.

ternal peripherals. This simplifies the configuration process at run-time but is bought with an increased latency through an additional FPGA for the crossbar.

4.2.2 Vertical Bitstream Relocation

For saving an external crossbar and enhancing I/O utilization, partial module bitstreams have to be mergeable with the static part of the system or other reconfigurable modules. This means that a partial module may not span the full height of the device and that parts of the static system or even other reconfigurable modules share the same CLB column. For Xilinx Virtex-II FPGAs, this is obligatory if modules are placed in a two-dimensional grid style. As mentioned before for these devices, the atomic reconfigurable fraction of configuration data that can be directly accessed is a full frame. However, Xilinx guarantees that no glitches or other side effects occur when overwriting any configuration with exactly the same content. This holds true as long as no look-up tables are overwritten that contain a state. Look-up tables may contain a state if they are used as 1) shift registers or 2) distributed memory. The Xilinx synthesis tool can be constrained not to use these two options. Note that it is not necessarily prohibited to use distributed memory within reconfigurable modules. If, for example, the configuration of a reconfigurable module will never be overwritten as long as the module is running, because it spans the full height of the reconfigurable area, the state inside the look-up table is not required to be touched by any reconfiguration process. This is the case for one-dimensional slot based systems and for such systems, a state stored inside some look-up tables must not be further considered.

Merging configuration bitstreams for Xilinx FPGAs was first proposed in a patent of the Xilinx Inc. [CKL+]. Sedcole et al. used the same approach for relocating modules on Xilinx Virtex-4 FPGAs in a two-dimensional manner [SBB+06]. In both approaches, the bitstream merging was carried out in a configuration readback-
4. Building Partially Reconfigurable Systems – Methods and Tools

Figure 4.8: a) Readback-modify-write bitstream manipulation. For changing fractions of a particular full column on a Xilinx FPGA, the columns can first be stored in a shadow memory using the configuration readback capability. This shadow can then be accessed freely for performing bitstream manipulations. Finally, the modified configuration data is written to its original location. b) Vertical inner-frame addressing for Xilinx Virtex-II FPGAs. The configuration data for I/O tiles and CLBs is strictly separated and the data of all tiles is concatenated from top to bottom and over all frames within the configuration bitstream.

modify-write operation, as illustrated in Figure 4.8a). In the presented prototypes in this thesis, the bitstream manipulations are not performed directly within the FPGA but in an extra shadow memory, hence, hiding the configuration readback process. Figure 4.8b) depicts the inner-frame addressing scheme of Xilinx Virtex-II FPGAs. As can be seen in the figure, the configuration bitstream is similarly regularly structured as the FPGA architecture. A single block RAM is sufficient to store a full configuration frame even for the largest Virtex-II device. However, a single frame buffer will dramatically reduce the configuration speed, because of Virtex-II FPGAs require one extra frame to be written to the device for each configuration read or write operation. This is independent of the amount of configuration data that is transferred during read or write operation. Let \( c_r \) and \( c_w \) be the data transfer rate for configuration read and write operations and \( o_r \) and \( o_w \) be the overhead required for reading and writing a portion of configuration data via the configuration interface, then the time for readback-modify-write bitstream manipulation method is:

\[
  t_{rmw} \approx f_m \cdot \left( \frac{f_h}{c_r} + \frac{o_r}{f_h \cdot c_r} + \frac{f_h}{c_w} + \frac{o_w}{f_h \cdot c_w} + \frac{m_h}{p_m} \right) \tag{4.1}
\]

Where \( f_m \) denotes the number of configuration frames, involved in the modification process, and \( f_h \) and \( m_h \) specify the total amount or respectively the amount of configuration data for a module inside one frame. \( f_h \) and \( m_h \) are related to the height of the device and the module. The buffer size \( f_h \) is given in terms of frames and
the actual bitstream manipulation is processed with the rate $p_m$. Equation (4.1) extends a description by Sedcole et al. [SBB+06] by a parametrizable buffer size and the initial overhead required for each access to the configuration port. The equation gives an estimation for the upper bound, as it is not assumed that the processes readback, modify, and write are performed in parallel. For instance, by using a double buffer for the readback data, the bitstream modifications can be performed in parallel with the readback process. The maximum configuration read and write transfer rate is in most cases the same and the bitstream manipulation in the buffer can be typically carried out considerably faster ($c_r = c_w \leq p_m$). The overheads $o_r$ and $o_w$ are slightly more than the amount of data stored inside one frame ($f_h$) in the case of Xilinx Virtex-II FPGAs. The configuration time is therefore at least four times longer than the time required to write all $f_m$ frames to the configuration port, if the buffer can store one frame. For larger buffers, the configuration time converges towards double the amount of time to write all $f_m$ frames. However, by using a buffer that can store all frames of the reconfigurable area, the readback process can be completely omitted. In this case, the data rate of the configuration interface can be fully utilized. A two-dimensional reconfiguration process is then carried out by first updating the configuration image in the buffer followed by writing all modified frames to the FPGA in typically one block. This will enormously speed-up the configuration process for the full buffer-based bitstream manipulation to roughly:

$$t_b \approx f_m \cdot \left( \frac{f_h}{c_w} + \frac{m_h}{p_m} \right)$$

Note that the configuration image manipulations can be performed per DMA and concurrently with the write process to the configuration port. Claus et al. presented in [CMZS07] how DMA can be used for transferring data to the internal configuration access port (ICAP) of a Xilinx Virtex-II Pro FPGA at very high speed. The reported configuration speed is about 95KB/ms which allows to exchange a 32 bit softcore processor in less than 2 ms. However, this comes along with a considerable memory burst during the configuration process that may interfere with the rest of the system. Section 4.3.2 presents compression techniques for reducing such bursts.

This reconfiguration speed enhancement of the full buffer-based bitstream manipulation is bought at the cost of an additional amount of RAM that is, in the worst case, identical to the size of a full FPGA configuration bitstream. However, the monetary cost for this additional memory will typically be negligible as compared to the cost of the FPGA itself. Alternatively, the shadow memory can be saved by concatenating the configuration data on the fly by directly merging bitfiles from the initial configuration storage and the partial module repository.
4.2.3 Small Bitstream Manipulations

Besides modifying larger parts of the configuration when loading new modules to the FPGA, some additional small manipulations may be required at runtime. This allows, for example, to change the address of a reconfigurable select generator (see also Section 3.5.2) or to connect the output of a partial module to a connection bar after configuration and initialization of a partial module. An individual configuration bit of a CLB can be addressed inside the configuration bitstream by its CLB position \((CLB_x, CLB_y)\) and the bit offset \((b_{CLB})\) within the CLB:

\[
b_{CLB,ADR}(CLB_x, CLB_y, b_{CLB}) = b_{init} + f_h \cdot f_w \cdot CLB_x + f_h \left\lfloor \frac{b_{CLB}}{CLB_h} \right\rfloor + IO_h + CLB_h \cdot CLB_y + b_{CLB} \mod CLB_h \tag{4.3}
\]

With the device specific constants \(b_{init}\) for denoting the initial offset for the first CLB column, \(f_h\) and \(f_w\) for stating the frame length and the number of frames per CLB column, and \(IO_h\) and \(CLB_h\) for defining the configuration frame length related to a single I/O cell or a single CLB. All parameters are illustrated in Figure 4.9. The BRAM configuration data is separated from the CLB configuration data in Xilinx Virtex-II FPGAs and located at the end of a configuration bitstream [Inc07]. The bit addressing inside the BRAM columns is analogous to the CLB bit addressing scheme but with a different parameter set.

While the configuration bitstream data for a particular LUT is stored well-ordered in one single frame, all switch matrix configuration bits are spread irregularly over a CLB configuration bit array of size \(f_w \cdot CLB_h\). However, the internal bitstream structure of one CLB was found identical throughout the complete FPGA fabric. Within the software driver, all bits belonging to a particular elementary reconfigurable resource (e.g., a clock input multiplexer), are stored in a list. This list first comprises the number of configuration bits \(n_c\) followed by a set of CLB configuration bit offsets: \(\{n_c, b_{CLB,0}, \ldots, b_{CLB,n_c-1}\}\). With this data structure, the specific configuration encodings for the different possible settings of the elementary resources can be easily

![Figure 4.9: Addressing an individual bit inside a CLB.](image-url)
4.3 High Speed Reconfiguration

In a ReCoNet, reconfiguration will be triggered seldom as the environment of the system is typically changing very slowly and faults may possibly never occur. However, reconfiguration speed is still an important factor within a ReCoNet, as it defines the majority of the time required to repair a hardware task. This dictates if a shadow task can be implemented resource efficient as a cold spare or if the shadow task must be already running as a hot spare. In other systems, reconfiguration may be frequently triggered and the configuration time can, in some cases, dominate within the system. For example, in the DISC-system [WH95], 71% of the overall time is spent on updating instructions in a reconfigurable processor.

Consequently, techniques for speeding-up the reconfiguration process are highly relevant for many runtime reconfigurable systems. These techniques include configuration prefetching, caching, bitstream decompression, and defragmentation. It is possible to apply these techniques individually or in combination with others. For example, bitstream decompression can be used for accelerating the defragmentation of the reconfigurable area, which, in turn, may be accomplished in the background before a new task reconfiguration is requested.

In the following sections, contributions for accelerating reconfiguration and, consequently, enhancing repair times in a ReCoNet will be revealed and discussed with related work.

4.3.1 Configuration Prefetching

Whenever possible, configurations should be prefetched such that the configuration and the execution of different modules overlap in time. The goal of this technique is to fully hide the reconfiguration time by swapping modules in the background. However, while this may be possible for periodically activated tasks, this is difficult if activations are unpredictable. In this case, speculative prefetching can be applied. Hauck proposed in [Hau98] to trigger the configuration prefetch with the help of dedicated preload instructions that are automatically included in a software binary. In the proposed system, some kernels of a software program have been replaced with reconfigurable accelerator modules and the configuration is triggered with respect to the current program state (defined by the program counter) and prior to the kernel invocation. Other related work includes reconfiguration tasks belonging to a particular hardware task that will be scheduled to a reconfiguration interface. In [DF07], Dittmann and Frank presented deadline monotonic scheduling of such reconfigura-
Figure 4.10: Configuration prefetching. Assuming a set of replicas (t'_2, t'_3, and t'_4) that each possess a repair slack sufficient to configure one resource slot before activating a replica in case of a fault, it is possible to bind all replicas as a cold spare to the same node n_1. In this case, any single node failure of n_2, n_3, or n_4 can be compensated.

Fast repair times may allow implementing replicas as cold spares even if the full reconfiguration time of a replica will exceed the time budget available for the repair, as shown in Figure 4.10. In the example, prefetching is used to host three replica tasks on one node that could otherwise only provide a subset of these replicas as the repair time is assumed not to allow reconfiguring more than one slot in case of a fault. This allows, for example, not only to place t'_3 as a hot spare on n_1 but in addition the replicas t'_2 and t'_4 all as cold spares. Hence, allowing higher reliability at the same cost.

Implementing replica tasks as cold spares on an FPGA for compensating a single node failure is feasible if the following requirements are met:

**Load**: The load of all running modules, plus the load of all partly configured cold spares, plus the maximum of the additional load required to activate all cold spares that will become active in case of a particular node failure must fit into the available resources.
Activation time: All cold spares that will be activated on a particular node failure must be activated before their entire repair time budget. Hot spare replica tasks possess an activation time of zero.

The activation time is mainly related to the reconfiguration time. If the activation time violates the repair time, bitstream decompression can reduce configuration time by shrinking the configuration data that is required to be transferred via low speed network links or read from slow memories.

4.3.2 Hardware Accelerated Bitstream Decompression

With progress in FPGA evolution, two trends, related to reconfiguration speed, can be observed: 1) higher possible configuration speeds at the FPGA configuration interfaces and 2) an increase in functionality (FPGA size), and consequently, an increase of the configuration data to be stored and transferred within the system. Not only the functionality of the device is rapidly increasing but also the complexity of typical modules that are integrated in a system. For instance, instead of integrating a simple 8 bit CPU, present systems may provide a complex 32 bit core.

The first trend results in an increasing gap between the possible FPGA reconfiguration speed and the throughput of non-volatile configuration memories. While many present FPGAs can be configured with data rates of up to 400 MB/s (e.g., Xilinx Virtex 4/5 or Altera Stratix IV), a typical NAND-flash memory device provides only a few tens of megabytes per second. Similarly, the gap between the network throughput in distributed embedded systems and the configuration speed has rapidly increased, hence, inhibiting remote configuration, where one node may configure another one via the network at full FPGA configuration speed. Newer network standards used in embedded systems, including FlexRay, or fast Ethernet as well as academic approaches, such as CAN+ [ZWT09] or the ReCoNets communication protocol [KSD06] are more than an order of magnitude slower than current FPGA configuration interfaces. The trend towards an increased functionality leads to larger configuration memory requirements as well as to an increased amount of configuration data that has to be transferred for loading some modules to an FPGA.

These effects can be diminished by storing and transferring compressed configuration data. This data can then be decompressed just in front of the relatively fast configuration interface. Because of the required throughput, this decompression has to be carried out in hardware. If we assume, for example, that the configuration data can be compressed to 50% of its original size, only half of the amount of data is required to be stored and transferred. Hence, the configuration speed may be potentially doubled. Storing bitstreams compressed is especially relevant for a ReCoNet where redundant copies of the same bitstream are required to provide fault tolerance within the system, as exemplarily depicted in Figure 4.11.

Bitstreams may contain fractions of data that are selectable to any value (don’t
Figure 4.11: Optimizing reconfiguration speed and configuration storage with hardware accelerated decompression in a ReCoNet. Each ReCoNode of the ReCoNet is equipped with an FPGA for implementing a CPU, a CompactFlash controller (CF), some reconfigurable modules, a network interface, a memory controller, and an interface for the FPGA internal access configuration port (ICAP). A decompressor (DECO) in front of the ICAP port can optionally decompress configuration data stored on a CompactFlash card, the local memory, or data located on another node. For tolerating node failures, configuration data has to be stored redundantly on different ReCoNodes.

care values) that can be exploited for achieving higher compression ratios [LH99]. However, as all bitstream formats of today’s FPGAs are confidential, exploiting don’t care values is not used and only lossless compression algorithms will be examined. Throughout this section, different hardware accelerated decompression techniques will be evaluated according to the following objectives:

- **Compression ratio**: for allowing considerable memory savings and throughput enhancements,

- **Throughput**: the decompressor has to emit at high continuous data rate. This includes the achievable clock frequency of the hardware decompressor as well as the compression algorithm.

- **Resource overhead**: The decompression hardware should be as small as possible.

**Impact of the Compression Ratio on the Configuration Speed**

Bistream compression is the process of encoding FPGA configurations using fewer bits than the original unencoded bitstream. Data savings are expressed with the compression ratio \( \eta = \frac{\text{size(compressed\_bitstream)}}{\text{size(original\_bitstream)}} \). With respect to the ReCoNets project, the main goal of bitstream compression is to widen the throughput bottleneck of the slow,
non-volatile memory and the low data rate on the network links. For the following, it is assumed that a bitstream decompressor is capable to emit one output token of word size $|w_{deco}|$ per clock cycle. With $|w_{FPGA}|$ being the size of the configuration interface, the decompression accelerator has to operate with at least $\frac{|w_{FPGA}|}{|w_{deco}|}$ times the clock frequency of the configuration interface in order to allow full configuration speed.

For accelerating the configuration process, it is not only important to put focus on the compression ratio, but also to consider the limited bandwidth at the decompressor input and output. Considering an FPGA with the peak configuration data rate $d_{FPGA}$ from a configuration memory providing the peak read data rate $d_{MEM}$, then the configuration data rate is:

$$w_c = \begin{cases} d_{FPGA} & \text{if } d_{MEM} \cdot \frac{1}{\eta} \geq d_{FPGA} \\ d_{MEM} \cdot \frac{1}{\eta} & \text{else} \end{cases}$$

Note that for any lossless compression algorithm, it is possible to construct cases where the compression ratio is larger than one [Heu03], thus, degrading the configuration time. For example, assuming a configuration bitstream where 10% of the bitstream contains worst case compression data that may be compressed by a ratio of $\eta_1 = 4$, while the 90% remaining bitstream data may be compressed at $\eta_2 = 0.1$. Then, the average compression ratio is $10\% \cdot \eta_1 + 90\% \cdot \eta_2 = 49\%$. When further assuming that the configuration memory is of half the throughput of the FPGA configuration interface. Then, at first view, it seems possible to achieve the optimal configuration time $t_{opt}$, that is only related to the size of the configuration bitstream divided by maximum configuration interface speed $d_{FPGA}$.

However, this holds true only if a decompression accelerator reads at a continuous data rate from the configuration memory. With respect to the assumed example, the 10% worst case configuration data produce a configuration memory burst that idles the configuration interface, while for the remaining 90% of the bitstream, the maximum configuration speed is limited by the configuration interface data rate. In the latter case, it is not possible to take full benefit of the high compression ratio and throughput is wasted at the configuration memory interface. As shown in Section 4.3.4, these bursts cannot be efficiently hidden with additional buffers between the configuration memory and the decompression accelerator. When taking the bandwidth of the configuration memory into account, the configuration time for the assumed example is $\frac{10\% \cdot \eta_1}{d_{FPGA}} + \frac{90\%}{d_{FPGA}} = 1.7 \cdot t_{opt}$. This means that the configuration time is reduced by only 15% as compared to an uncompressed configuration process that would take two times $t_{opt}$ for the assumed memory data rate. Despite

\footnote{The vertical bars denote the word size of the surrounded word that is the number of bits used to code a word. This notation is taken from Salomon [Sal04] and will be consistently used throughout this section.}
the good average compression ratio of below 50%, this result is far away from the optimal configuration time \( t_{opt} \).

If the bitstream compression algorithm will be designed, such that the worst case compression ratio is limited at the cost of a reduced best case compression ratio, the configuration time can be further enhanced. For instance, by limiting the compression ratio of the 10% of the bitstream containing worst case configuration data to, for example, \( \eta_1' = 2 \), while having a compression ratio of \( \eta_2' = 0.4 \) for the remaining 90%, the average compression ratio will result in \( 10\% \cdot \eta_1' + 90\% \cdot \eta_2' = 56\% \). The configuration time for this adjusted algorithm is then only \( \frac{10\% \cdot \eta_1'}{d_{FPGA}} + \frac{90\% \cdot \eta_2'}{d_{FPGA}} = 1.3 \cdot t_{opt} \).

These numerical examples point out that looking only at the overall compression ratio will not necessarily enhance the configuration speed. This observation reflects Amdahl’s law. As compared to the execution time of an algorithm that cannot be reduced below its sequential fraction, the configuration process cannot be accelerated beyond the configuration interface limit \( (t_{opt}) \) regardless of the achieved compression ratio.

### 4.3.3 Algorithms for Bitstream Compression

In this section, existing data compression algorithms will be analyzed with respect to their suitability for hardware accelerated bitstream decompression. Various modifications to standard algorithms will be discussed in order to fulfill the objectives of compression ratio, throughput, and resource overhead at the same time and with high quality. Here, high quality means implementations that achieve compression ratios that can compete with state of the art software programs, such as gzip, implementations that allow throughputs of several hundreds of megabytes, and implementations with small resource requirements. In the following, we will examine configuration ratio issues together with implementation aspects for variations of well known compression algorithms, such as run length encoding, Lempel Ziv, and Huffman encoding.

#### Run Length Encoding

Run length encoding [Sal04] is a simple technique to compress a sequence of identical tokens belonging to a data stream. Thus, run length encoding is a good candidate to compress configuration bitstreams that are sparsely populated with binary 1 values and that offer lots of consecutive identical tokens. The recurrences will be encoded by tuples \((\tau, l)\) with \(\tau\) being the next token emitted by the decompressor and \(l\) being the run length denoting the number of times this token is repeated in the original data. For example, the sequence \(\text{A B B C C C} \) is encoded as \((\text{A}, 1) (\text{B}, 2) (\text{C}, 3)\). The data width of a token is \(|\tau|\) bit and the amount of bits used to encode the run length is \(|l|\) bit. Run lengths larger than \(2^{|l|}\) will be encoded by multiple tuples that allow to recompose the original input data. When \(\tau\) and \(l\) are encoded with a fixed amount
of bits, the compression ratio $\eta_{RLE}$ is in the following range ($\hat{\eta}_{RLE}$ denotes the worst case compression ratio):

$$\frac{|\tau| + |l|}{|\tau| \cdot 2^{|l|}} \leq \eta_{RLE} \leq \frac{|\tau| + |l|}{|\tau|} = \hat{\eta}_{RLE}. \quad (4.4)$$

The cost to encode a run length tuple is always $|\tau| + |l|$ bits. These bits encode in the best case $|\tau| \cdot 2^{|l|}$ bits and in the worst case only $|\tau|$ bits. Note that for the latter case, the compressed data will be larger than the original input data.

The token bit width $|\tau|$ (and consequently, the amount of different tokens) and the amount of bits to encode the run length $|l|$ have to be determined with care, because when random data is compressed using run length encoding, the compressed data will be larger than the original input data. This is illustrated in Figure 4.12 on the basis of a probabilistic state machine. The probability that the next token in a random data stream is identical to the present one is $\hat{p}(1) = \frac{1}{2^{|\tau|}}$. The probability for longer recurrences $i$ decreases exponentially: \(\hat{p}(i) = \frac{1}{2^{|\tau|}} \cdot i\). The compression ratio for an infinite random data stream is therefore:

$$\bar{\eta}_{RLE} = \sum_{i=1}^{2^{|l|}} \frac{|\tau| + |l|}{|\tau| \cdot i} \cdot \frac{1}{2^{|\tau|} \cdot i}. \quad (4.5)$$

Even for relatively small token bit widths, the probability that just two consecutive words are identical is about zero (e.g., $< 1\%$ for bytes). Therefore, $\bar{\eta}_{RLE}$ converges against the worst case compression ratio $\hat{\eta}_{RLE} = \frac{|\tau| + |l|}{|\tau|}$ for increasing $|\tau|$.

The values of $\bar{\eta}_{RLE}$ and $\hat{\eta}_{RLE}$ indicate the behavior when the bitstream contains fractions of high entropy data. This can happen especially for initialization data of internal RAM blocks. As the relative amount of RAM initialization data is not negligible\(^5\), the compression algorithms should have an $\eta$ and an $\hat{\eta}$ as small as possible, with $1 < \eta < \hat{\eta}$.

For parameterizing the run length algorithm, a histogram of the probability distribution of the run lengths for different token bit widths $|\tau|$ of the complete benchmark corpus (see also Section 4.3.4) has been analyzed.

\(^5\)According to the data sheets for the chosen benchmark FPGAs [Alt09, Xil07] in Section 4.3.4, the relative amount of RAM preload data is in average 14% of the overall bitstream size.
Figure 4.13: Histogram of the first 16 most occurring run lengths found inside all benchmark bitstreams (see Section 4.3.4). The abscissa enumerates classes of run lengths by their occurrence that is not necessary the run length itself. However, class 1 also denotes the occurrence for a run length of 1. The columns quote the portion of a particular run length occurrence class for different token bit widths (1, 2, 4, 8, 16) with respect to the total amount of codewords required to encode the benchmark using RLE.

The histogram in Figure 4.13 reveals that higher rates of consecutive sequences exist only for small token bit widths. For instance, for a token bit width of $|\tau| = 16$, the probability is 55% that the run length is 1, which means that a sequence of identical tokens contains only one token. Furthermore, the histogram points out the potential for data compression through run length encoding. By adding, for example, the probabilities of the first ten run lengths ($\sum_{i=1}^{10} p(i)$), it can be observed that for all token bit widths except for the width $|\tau| = 16$, the sum of the probabilities is below 50%. Thus, for these token bit widths, the majority of run lengths is larger than ten, thus, pointing out feasibility for run length compression. However, when using the described encoding, the run length 1 will lead to a relatively large portion of data $\sigma$ in the compressed data stream that is larger than the original one: $\sigma = \frac{|\tau|}{|\tau|} \cdot p(1)$.

For instance, if the token bit width is set to $|\tau| = 4$ bit, 27% of two consecutive words differ (see class 1 in Figure 4.13).

In order to overcome these issues, two RLE variations have been examined for hardware accelerated bitstream decompression. The first one works on a bit width of a single bit, while the second one uses a flag to distinguish between the case of a recurrent sequence or the case that a sequence contains only one word.

Toggle RLE

If the token bit width is set to $|\tau| = 1$ bit, the run length value specifies the distance between two bit flips. By further defining a fixed start bit value, the complete bitstream data is compressible by encoding only run lengths $l$ while omitting the next token $\tau$. By reserving some codes for specifying run lengths without toggling at the
Figure 4.14: Toggle run length decompressor. At the beginning of a decompression process, the output flip-flop is initialized. Each time when the sequence of equal bits ends, the flip-flop is toggled and the equal bit counter will be loaded with the next run length until the complete bitstream is decompressed.

end, it is possible to compress recurrent sequences of any length. For encoding the run length, different prefix free encodings have been tested for balancing between the compression ratio and the hardware required to decode the prefix free code words. In addition, the encodings ensure a bounded compression ratio for random data as well as worst case examples. For an example, the following encoding is assumed:

<table>
<thead>
<tr>
<th>run length</th>
<th>1  2  3  4  4*  8*  16*  64*</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>00  01 100 101 1100 1101 1110 1111</td>
</tr>
</tbody>
</table>

*no toggling after this run length

If the start bit value is chosen to 0 then the sequence 1 00 111 000000 is encoded as 00 01 100 1100 01. In this example, a run length of 1 will double the amount of data required to be transferred from the configuration memory. However, a run length of 1 exists in only 7% of the used benchmark bitstreams as depicted in Figure 4.13 for class 1 and \(|\tau| = 1\).

As illustrated in Figure 4.14, the toggle RLE hardware decompressor requires only an input buffer (containing the decoder for the prefix free code), two counters, a control state machine, and a decoder for the run length values. The toggle RLE technique is suitable for FPGA as well as CPLD implementations. The major drawback of this decompression technique is its sequential operation.

Flag RLE

For enhancing the throughput, a variation of the traditional run length encoding scheme was examined that uses a flag to distinguish between the case that consecutive tokens differ and the case of a recurrent sequence of identical tokens (run length equal or larger than one). The corresponding encodings are \((0', \tau)\) and \((1', \tau, l)\). Thus, the worst case compression ratio is \(\hat{n}_{RLE} = \frac{|\tau|+1}{|\tau|} \). Note that the flag RLE approach is a special case of a prefix free encoding of the run length \(l\). When running at the same clock frequency, the flag RLE approach possess a \(|\tau|\) times higher throughput as compared to a toggle RLE decompressor. The flag RLE algorithm is suitable for FPGA as well as CPLD implementations.
Figure 4.15: LZSS compression. A search buffer holds already compressed data and a look-ahead buffer contains the next portion of uncompressed data. The example shows a five bit prefix of the look-ahead buffer that is also found in the search buffer at position six. This match will be coded as $(1', 6, 5)$ and denotes a reference to the search buffer.

LZSS Encoding

The LZ algorithms (named after their inventors A. Lempel and J. Ziv) are string substitution schemes. A well known LZ variant is LZ77 [ZL77]. String substitution schemes replace uncompressed data by references to already compressed data. During compression, as shown in Figure 4.15, a search buffer keeps track of the last already compressed tokens and a look-ahead buffer allows accessing the next tokens of the uncompressed data. When compressing data, the search buffer is scanned for matches of all prefixes within the look-ahead buffer. The longest prefix of the look-ahead buffer, found in the search buffer, is then replaced by its starting offset and its length. If no prefix is found, the first token (nextToken) of the look-ahead buffer will be emitted.

The LZSS approach [SS82], as an improvement of LZ77, that uses a flag to distinguish between these two cases (that a match was found in the search buffer or not). This has similarities with the flag RLE methodology presented in the last section. Let $o$, $l$, and $\tau$ denote the offset, the prefix length, and the nextToken and $|o|$, $|l|$, and $|\tau|$ the number of bits to code these values. Then, during LZSS compression, either the codeword $(1', o, l)$ or $(0', \tau)$ is emitted (see also Fig. 4.16). The best compression ratio is achieved if the longest possible prefix match of size $l_{\text{max}} = 2|l|$ is found. In this case, $l_{\text{max}} \cdot |\tau|$ bits are encoded with $1 + |o| + |l|$ bits. If no match was found, a token of $|\tau|$ bits is encoded by $1 + |\tau|$ bits. This leads to the following lower and upper bound for the compression rate $\eta$ of LZSS:

\[
\frac{1 + |o| + |l|}{l \cdot |\tau|} \leq \eta_{\text{LZSS}} \leq \frac{1 + |\tau|}{|\tau|} = \tilde{\eta}_{\text{LZSS}} \tag{4.6}
\]

The upper bound is equal to the upper bound of the flag RLE algorithm. Thus, LZSS does not lead to a drastic downgrading of the compression ratio if fractions of the configuration bitstream contain data with a high entropy.
When assuming the following LZSS encoding with $|\tau| = 8$, $|o| = 5$, $|l| = 3$ for encoding some ASCII characters:

$$
(0, 01000001) \quad (0, 01000010) \quad (1, 00001, 010) \quad (0, 01000011)
$$

The decompressed sequence of this encoding is: A B A B C

*In the final implementations, the shortest length is 2 (encoded as 000) because a length of 0 is futile and a length 1 is encoded with a new token.

**Figure 4.16:** LZ77 HW decompressor proposed in [HW99], and an accord example for LZSS decompression.

### LZSS Hardware Decompression

During decompression, offset and length refer to already emitted data. Thus, the decompressor must keep track of the last $n$ emitted tokens, with $n$ being the search buffer size of the compression program. For the LZSS decompression, an existing LZ77 hardware decompressor [HW99] was enhanced with a state machine that decodes LZSS encodings as illustrated in Figure 4.16. During decompression, the emitted data is fed into a shift register that represents the mentioned search buffer. If there is no match in the shift register, the next token $nextToken$ will be emitted. In case of a match, the offset field $o$ specifies the position of the shift register containing the first token of an $l$ token long match.

As CPLDs only provide a relatively small amount of memory bits, LZ decompressors are inapplicable for CPLD implementations, because of the required shift register. For implementing the shift register on FPGAs, the following four options exist:

1. Look-up table flip-flops (the flip-flops available behind each look-up table),

2. Frame Data Input Register (FDRI) provided in Xilinx FPGAs [Inc07] (this register stores the configuration data of a complete frame and is only accessible via shift operations),

3. Dedicated RAM-Blocks, and

4. Dedicated shift register primitives (SRL16) in Xilinx FPGAs [Inc05a].

The first solution consumes an unfavorable amount of hardware resources. The second one is advantageous when implementing hardware decompression on the FPGA.
4. Building Partially Reconfigurable Systems – Methods and Tools

itself as a fixed part of the chip. Accessing data stored inside the FDRI is very slow when using the configuration interface. However, [PMW04] and [LH01] present compression techniques that are based on the FDRI, but both papers omit a discussion of a possible hardware decompressor implementation.

The third variant of implementing the LZ shift register uses dedicated on-chip RAM resources and allows implementing very large shift registers. This was proposed by Hübner in [HUWB04, HUWB05] for hardware accelerated LZSS configuration data decompression. However, as huge shift registers require more bits |o| for specifying the offset o, no benefits have been found by using larger shift registers. By analyzing how circuits are typically mapped onto an FPGA, many regularities can be found within the resulting configuration primitives and consequently within the corresponding configuration data. For example, many arithmetic functions use carry chains, hence, resulting in identical configurations among a set of consecutive basic elements. In the case of Xilinx FPGAs, the basic elements of a carry chain are vertically aligned; and the configuration for the basic elements is stored directly adjacent within the bitstream. Consequently, short shift registers perform better on LZSS for decompressing configuration bitstreams for Xilinx FPGAs than larger ones.

Instead of using dedicated RAM blocks, the last option for implementing the LZ shift register with the help of dedicated shift register primitives is most advantageous for bitstream decompression. It was discovered that even small search buffers, as little as 32 tokens, result in good compression ratios. Such shift registers are appropriate to be implemented with the help of SRL16 primitives on Xilinx FPGAs. This primitive allows to use a 4-bit look-up table alternatively as a 16 bit shift register with random access to the individual register contents. An l deep shift register of token bit width |τ| can be created by cascading |τ| ⌈l / 16⌉ SRL16 primitives.

The best parameter set found for LZSS encoding was one bit for the flag, |l| = 3 bit, |o| = 5 bit, and |τ| = 8 bit. This results in a nine bit long codeword which does not match a typical 8-bit interface. To simplify the connection to 8-bit wide flash memory interfaces, an extra register is used for the flags. This register stores the flags for a block of 8 consecutive codewords, thus, leading to 8-bit wide remaining codewords.

Recent FPGAs offer high speed configuration interfaces that are up to 32-bit wide and that operate with up to 100 MHz [Alt09, Xil07]. In order to match these interfaces, the token bit width can be extended to 32 bits. This would influence the compression as follows: With an increased token bit width, long prefix lengths become more rarely. Therefore, the subset of encoded prefix lengths has to be adapted. In addition, the worst case compression rate improves with higher word widths. However, it is also possible to enhance the throughput by increasing the clock frequency of the decompressor in combination with a serial-to-parallel converter at the output. The LZSS decompressor consists of a few simple basis elements and the whole design can be easily pipelined. Consequently, we can easily build decompression modules that run at least at 200 MHz (e.g., when implemented on a Xilinx Virtex-II device).
This speed in combination with a datapath width of 16 bit allows decompressing at a rate of 400 MB/s. Thus, the full configuration interface bandwidth of today’s FPGAs can be utilized under optimal conditions.

**Huffman Encoding**

As an entropy encoding scheme, Huffman encoding [Huf52] assigns short codewords to frequently occurring tokens and longer code words to infrequently occurring tokens. Huffman encoding is based on a Huffman tree, which in turn is created upon a probability distribution of tokens. Creating an individual Huffman tree for each bitstream file is called *dynamic Huffman encoding*. Such a Huffman tree is part of the compressed data and has to be reconstructed before decompression. Such decompressors turn out to be costly in terms of hardware usage [WL05].

Applying the same Huffman tree for all bitstreams is called *static Huffman encoding*. The static Huffman tree is based on the probability distribution of a benchmark corpus and will never change. The higher the variation in the probability distribution, the better the compression ratio achievable with Huffman encoding.

Let $|h_{\text{min}}|$ and $|h_{\text{max}}|$ denoting the smallest and the largest Huffman codeword sizes found inside a Huffman tree and $|\tau|$ be the token bit word size; then, the compression ratio for Huffman encoding is:

$$\frac{|h_{\text{min}}|}{|\tau|} \leq \eta_{\text{Huff}} \leq \frac{|h_{\text{max}}|}{|\tau|} = \hat{\eta}_{\text{Huff}}$$

If the decompression ratio has to be bounded in such a way that fractions of high entropy data will not completely cut off the throughput, the Huffman code has to be restricted. For example, if the worst case compression ratio $\hat{\eta}_{\text{Huff}}$ has to be restricted to, for example, less than 2, the largest code word size has to be limited to $|h_{\text{max}}| \leq 2 \cdot |\tau|$. In the case $\hat{\eta}_{\text{Huff}}$ is larger than 2, the depth of the Huffman tree has to be reduced, as illustrated in Figure 4.17.

Reducing a Huffman tree does not necessarily reduce the maximum configuration speed. For example, in the histogram in Figure 4.18 that depicts the probability of occurrence for the most frequently occurring tokens within all benchmark bitstreams, it can be seen that for a word size of $\tau = 8$ bit there is one token occurring with a probability of 58%. Consequently, this token will be encoded in a classical Huffman encoding with one bit. However, when assuming that the configuration memory interface can deliver 50% of the maximum configuration data rate, up to 4 bits can be spent for the most occurring token (that possesses a word size of 8 bits), before the configuration speed drops down. In other words, before this point, the maximum throughput of the configuration interface will limit the configuration speed. By reducing the height of the Huffman tree, the average compression ratio will be decreased. However, this will still enhance the configuration speed, because the longest code word size has been reduced.
4. Building Partially Reconfigurable Systems – Methods and Tools

Figure 4.17: a) Huffman tree and b) a reduced Huffman tree. Both examples are for the same distribution of 3-bit wide tokens (the eight leaves of the tree). The reduced Huffman tree has a better worst case behavior, which means that in the longest encoding of a token requires less bits (compare the encodings for the values $111$ and $110$ in a) and b)). This reduces the peak input data rate of the decompressor by the cost of an inferior average compression ratio. This can be applied to speed up the configuration process.

Figure 4.18: Normalized histogram of the probability of occurrences for the first 16 most frequently appearing tokens found inside all benchmark bit-streams. The abscissa enumerates tokens by their probability of occurrence. Note that in the case of a token bit width of $|\tau| = 4$, the sum over all listed occurrences is 100%; because in this case, only $2^{|\tau|} = 2^4 = 16$ different tokens exist.
4.3 High Speed Reconfiguration

Huffman Hardware Decompression

For static Huffman trees, a code generator was implemented that can automatically create the hardware for Huffman decompressors. An example of such a decompressor is illustrated in Figure 4.19. When in this example a new word is decoded upon enabling the `start` input, the values in the so-called `alignment register` (flip-flops $r_0,\ldots,r_2$) will activate exactly one output of the decoder tree. As the words of the original bitstream data are encoded into codewords of different sizes $h(i)$, the alignment register is used to arrange the incoming data $d$, such that the Huffman tree decoder gets its incoming codeword always at exactly the same position. The outputs of the Huffman tree decoder indicate the decoded token of the alphabet $\{00,01,10,11\}$ that is emitted in the following. Concurrently, the alignment register is shifted right by the length $|h(i)|$ of the encoded token. For instance, if the output for token 01 becomes active, the alignment register is shifted by two, which is selected by the multiplexer input $C2$.

A state machine (not shown in Figure 4.19) tracks the fill level of the alignment register. When there is sufficient space in the alignment register, the next value from the input $d_0,\ldots,d_3$ will be loaded via the bottom multiplexers to the most right free position. For instance, in a scenario where $r_3,\ldots,r_5$ are empty (not containing a codeword) and where $r_0=0$ and $r_1=1$, the output 01 of the Huffman tree decoder will become active. Consequently, $r_0$ and $r_1$ will be consumed and the last valid fragment of the compressed bitstream stored in $r_2$ will be shifted into $r_0$. This will result in five free entries in the alignment register and a new input will be stored in the registers $r_1,\ldots,r_4$ which is selected by the multiplexer input $f5$.

The size of the alignment register $|r|$ depends on the input data width $|d|$ and the Huffman tree depth which is equal to $|h_{\text{max}}|$. If in the case that the alignment register is missing one bit to decode the next token, it must be possible to load the alignment register with a new input word for preventing a deadlock. Thus, the required alignment register size is $|r| \geq |d| + |h_{\text{max}}| - 1$ flip-flops.

The decompressor example in Figure 4.19 points out that it requires more logic for aligning the input data than for the Huffman decoder itself. The logic overhead for the multiplexers is enormous for larger trees. For each occurrence of a Huffman codeword width, one input for the top multiplexers as well as one input for the bottom ones adjusting the input data is required. In addition, the structure contains long combinatorial paths and is difficult to pipeline, because only when a decode phase has determined the next output token, it is known from which position in the alignment register the next decode phase will start. However, as shown in the following section, static Huffman encoding reaches better compression ratios than the RLE or LZSS variants.
4. Building Partially Reconfigurable Systems – Methods and Tools

Figure 4.19: Huffman decompressor. A Huffman tree decoder decodes a new output word of the alphabet \{00, 01, 10, 11\} (|t| = 2) upon start=1. The values in brackets denote the compressed encodings. The multiplexers in front of the alignment register r shift the compressed bitstream data according to the size of the current code word h(i) such that the next code word is located again from the right inside r. The bottom multiplexers load new compressed input data to r according to the current fill level.

4.3.4 Benchmarking Bitstream Decompression Accelerators

All compression algorithms exploit statistical characteristics within the uncompressed data like non-uniform probability distributions of tokens or some regularities found in the input data. In the case of FPGA configuration bitstreams, such characteristics include sparsely populated configuration data (more ‘0’-bits than ‘1’-bits) and recurrences due to regularities in the mapped circuits. Such regularities can be visualized, as shown in Figure 4.20.

Experimental Setup

For studying bitstream compression algorithms, suitable benchmark bitstreams are required that represent the statistical characteristics for a wide range of different FPGA designs. Most FPGA designs will hold a high logic utilization for allowing smaller FPGAs\(^6\). This significantly minimizes overheads like monetary cost, static power consumption, or configuration time. Consequently, suitable benchmark bitstreams are requested to hold a logic utilization of at least 90% of the available look-

\(^6\)As the routing fabric of an FPGA requires more area on a die than the logic (mainly the look-up tables), it is advantageous to delimit the routing capabilities. This will result in a lower logic utilization but still allows mapping circuits to FPGAs possessing an on average smaller die [Deh99]. However, this observation does not consider that the place and the route steps are separately performed in the tools of the main FPGA vendors. In addition, customers demand high logic utilization, hence driving FPGA companies to include sufficient routing capabilities in their architectures.
4.3 High Speed Reconfiguration

Figure 4.20: Structure of a configuration bitstream, a software binary, and an ASCII file. Inside the images, a black pixel represents a ‘1’-bit. (from [Bec07]).

up table resources. In order to detect influences on the compression ratios that are related to the implemented algorithm of a module and not only to the logic utilization, examples have been selected from different application domains. This includes cryptographic accelerators, signal processing cores, communication modules, and an SoC-design. More detailed information on the benchmark can be found in [KBT07].

Evaluating Compression Ratios

Figure 4.21 lists measured compression ratios for multiple decompressor variations. As a reference, the figure lists the compression ratios achieved with the program gzip (V 1.3.5 - -best) and for two FPGA vendor-specific dedicated configuration memory devices [xcf07, epc07] that both perform bitstream decompression. Beside the different bitstreams of the benchmark modules, the average value over all bitstreams (AV) is quoted. For the toggle run length encoding, a decompressor with fixed length encoding (T-RLE) as well as with a prefix free encoding (T-RLE*) was examined. The two LZSS alternatives differ by their data path size (8/16 bit). The word size of the flag RLE and the Huffman decompressor was selected to 8 bit.

For the LZSS accelerators, it was found that most data inside the compressed stream is spent on nextTokens (see also Figure 4.16). Consequently, hybrid accelerators have been tested that use beside the LZSS decompressor an additional Huffman decompressor for the nextTokens. In order to keep the logic overhead in acceptable margins, two variants of Huffman decompressors with 4 bit wide words ($|\tau| = 4$) have been used. The first one (named LZSS+Huff2 in Figure 4.21) is based on a binary Huffman tree, while the second one (LZSS+Huff4) is based on a quaternary Huffman tree [KBT09b]. The latter one achieves an on average 1.2% inferior compression ratio, but with 436 look-up tables, requires only 63% of the 693 LUTs necessary to implement the accelerator based on binary Huffman decoders.

The best compression ratios have been measured for the Huffman decompressor
that in some cases outperforms gzip. Huffman performs good because bitstreams are sparsely populated with ‘1’-bits, hence, leading to a strong non-uniform distribution of the code words. For the Altera Cyclone-II benchmark bitstreams, the flag RLE (F-RLE) ranges always and on average 6% better than the dedicated configuration device EPC16 (available from Altera). For Xilinx Virtex-II and Virtex-5 bitstreams, the toggle RLE using a simple prefix free encoding (T-RLE*) ranges with one exception (the FIR filter on Virtex-II FPGAs) always better than the XCF16P configuration device from Xilinx. The LZSS cannot compete with the RLE algorithms for the Cyclone-II benchmark, but it is the most advantageous one for decompressing configuration data for Xilinx FPGAs, because of its high speed and its low implementation cost as revealed in the following. It is assumed that the configuration data within Cyclone-II FPGAs and the logic tiles are not directly correlated. For example, this happens if the configuration is written row by row to the FPGA while circuits are preferentially mapped in a column-wise manner (e.g., when using carry chains).

**Synthesis Results for the Bitstream Decompressors**

The different decompression accelerators have been implemented and the synthesis results are listed in Table 4.1. The table reveals that less than a 100 look-up tables are
4.3 High Speed Reconfiguration

sufficient to achieve average compression ratios of 50% on an ESM [MTAB07] (see also Figure 4.21). For instance, the LZSS8 accelerator costs only 83 LUTs (0.14% of the XC2V6000 main FPGA) but may decompress data with double the peak data rate of that of a Virtex-II configuration interface. By doubling the size of the data path, the LZSS16 is capable to decompress configuration data at 400 MB/s, which is in turn the peak configuration speed of Xilinx Virtex-4/5 FPGAs.

The best hardware compression ratios have been reached with Huffman encoding that indeed turns out to become much more costly than the other accelerators. According to the data sheets [xcf07, epc07], both dedicated configuration devices, the EPC16 and the XCF16P, provide a maximum output data rate of up to 40 MB/s, which is an order of magnitude below the LZSS16. Assuming that the input stream arrives fast enough at the decompression modules, all accelerators are capable to emit one token of $|\tau|$ bit per clock cycle.

**Table 4.1:** Synthesis results of the RLE and LZSS decompression accelerators.

<table>
<thead>
<tr>
<th>T-RLE${}^{\dagger}$</th>
<th>T-RLE${}^{\dagger}$</th>
<th>F-RLE${}^{\ddagger}$</th>
<th>LZSS8${}^{\dagger}$</th>
<th>LZSS16${}^{\dagger}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs / MCs</td>
<td>111</td>
<td>57</td>
<td>85</td>
<td>83</td>
</tr>
<tr>
<td>$F_{\text{max}}$[MHz]</td>
<td>165</td>
<td>67</td>
<td>193</td>
<td>198</td>
</tr>
</tbody>
</table>

Implemented on: ${}^{\dagger}$ Xilinx XC2V40-5 FPGA, ${}^{\ddagger}$ Altera EPM7064-10 CPLD

Synthesis results for the static Huffman decompressors.

<table>
<thead>
<tr>
<th>Huffman</th>
<th>Cyclone</th>
<th>Spartan-III</th>
<th>Virtex-II</th>
<th>Virtex-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>4451</td>
<td>4223</td>
<td>5286</td>
<td>4114</td>
</tr>
<tr>
<td>$F_{\text{max}}$[MHz]</td>
<td>50</td>
<td>60</td>
<td>45</td>
<td>66</td>
</tr>
</tbody>
</table>

All timing results are based on the slowest available speed-grade.

**Configuration Time Improvements**

As discussed in Section 4.3.2, the maximum clock frequency and the achieved compression ratio will not automatically deliver the minimum reconfiguration time because the decompressor may be stalled on missing input data. This impact of the memory bandwidth on the configuration speed has been measured for the LZSS decompression accelerator that is best suitable for Xilinx FPGAs. The exact decompression time was determined with the help of RTL simulation. Furthermore, the impact of a FIFO between the memory and the decompressor has been investigated for different FIFO sizes. During simulation, accesses to the memory was restricted by allowing a read operation only every $\gamma$ cycles. For instance, $\gamma = 3$ denotes that the peak throughput rate of the decompressor is three times higher than the memory data rate. The optimal time is bounded by the achieved compression ratio or limited by the configuration interface throughput in the case when $\eta \leq \frac{1}{\gamma}$.

As can be seen in Table 4.2, the optimal reconfiguration times cannot be reached, but still the configuration speed was significantly enhanced by balancing compression
Table 4.2: Summarized decompression times for various FIFO sizes and diverse memory bandwidths $\gamma = \frac{d_{FPGA}}{d_{MEM}}$. The table lists the relative time compared to the not compressed case. Italic values denote that the optimal speed-up is limited by the maximum decompressor output data rate and not by the compression rate.

<table>
<thead>
<tr>
<th>$\gamma$</th>
<th>opt.</th>
<th>FIFO size in byte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Altera</td>
<td>2</td>
<td>63.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>63.1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>63.1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>63.1</td>
</tr>
<tr>
<td>Cyclone-II</td>
<td>2</td>
<td>50.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>43.5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>43.5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>43.5</td>
</tr>
<tr>
<td>Xilinx</td>
<td>2</td>
<td>59.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>59.1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>59.1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>59.1</td>
</tr>
<tr>
<td>Virtex-II</td>
<td>2</td>
<td>50.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>49.3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>49.3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>49.3</td>
</tr>
</tbody>
</table>

Furthermore, the table points out that the FIFO size does not have much impact on the final configuration time. Experiments with larger FIFOs have not demonstrated a significant further improvement of the configuration speed.

**Firmware Decompression**

Beside accelerating FPGA reconfiguration, hardware supported decompression is further suitable for speeding up system boot or migrating software tasks within a ReCoNet. This has been experimentally evaluated for the firmware of the ESM platform [MTAB07]. In an ESM, a Linux kernel and a file system is copied from a NAND flash device to the SDRAM at system start. With four flash memory devices, a read data rate of 25 MB/s (32 bit @ 6.25 MHz) can be achieved from non-volatile memory, while the SDRAM memory transfers at a data rate of 200 MB/s. Therefore, the firmware boot process represents a similar situation as the FPGA configuration.
Table 4.3: Firmware loading times. The native column denotes the values for the case that the complete firmware binaries will be stored uncompressed in the non-volatile flash memory.

<table>
<thead>
<tr>
<th></th>
<th>native</th>
<th>SW decompression</th>
<th>LZSS8 HW decompr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>flash size [MB]</td>
<td>20.185 (100%)</td>
<td>4.757 (24%)</td>
<td>8.535 (43%)</td>
</tr>
<tr>
<td>loading time [s]</td>
<td>0.91 (100%)</td>
<td>4.4 (484%)</td>
<td>0.63 (69%)</td>
</tr>
<tr>
<td>throughput [MB/s]</td>
<td>22.22 (100%)</td>
<td>4.59 (25%)</td>
<td>32.1 (144%)</td>
</tr>
</tbody>
</table>

process. In both cases, data has to be transferred from a slow memory to a much faster interface.

Three alternatives for loading the firmware have been considered. In the 1) native variant, the complete firmware will be directly copied from the uncompressed flash memory to the SDRAM. Alternatively, the firmware will be 2) decompressed in software, or 3) decompressed in hardware before writing it to the SDRAM. The results are listed in Table 4.3 that states the memory consumption in the flash memory, the time required to load the firmware, and the throughput achieved to transfer the firmware from the slow flash memory to the SDRAM.

The listed throughput for the native variant is computed on the assumption that the full transfer speed of 25 MB/s for the flash and 200 MB/s for the RAM can be achieved. Then, the combined throughput is 

\[
\frac{1}{\frac{1}{25} + \frac{1}{200}} MB/s = 22.22 MB/s.
\]

Additional latencies, like for example, addressing overheads for the SDRAM, are ignored. The software decompression alternative utilizes the Power-PC CPU (MPC875) of the ESM that is running at 50 MHz. The decompression is part of a boot loader that decompresses the kernel and the RAM-file system using GNU zip. The time was measured by monitoring the chip select signal of the flash memories. The last alternative lists the results determined for LZSS hardware decompression. The time was acquired by an RTL simulation, similar to the simulations performed in the last section for decompressing configuration data. In the simulation, the flash memory access was limited to every eighth clock cycle (\(\gamma = 8\)) while the RAM was accessed with the full 50 MHz system clock. The compression ratio of 43.4% was achieved with the help of a 16 byte input FIFO. Note that the FIFO and the LZSS decompressor can be implemented together with less than a hundred look-up tables on most Xilinx FPGAs by using some LUTs as a shift register (SRL16 primitive [Inc05a]).

The LZSS decompression does not perform as good as GNU zip. As a consequence, it requires almost double the amount of flash memory to store the firmware. On the other side, the LZSS decompressor is about 7 times faster. As depicted in Figure 4.20, software binaries have a completely different structure than configuration bitstreams. It was found that Huffman encoding enhances the compression ratio to 39.5% and a combined LZSS-Huffman encoding to 33.7%. However, this compression ratio is bought by a large resource overhead for implementing the hardware
decompressor. For allowing good compression ratios without taking the hardware overhead, partial runtime reconfiguration may be used. For instance, the combined LZSS-Huffman accelerator would require about 5000 look-up tables that can be reconfigured within 5 ms on a Xilinx Virtex-II FPGA. This holds true for the assumption to configure the device with the same throughput as achieved for decompressing the software (32.1 MB/s). Note that this 5 ms are only 1% of the time required to perform the firmware decompression.

4.3.5 ASIC Implementation of an LZSS Decompressor

Instead of using the FPGA logic fabric for implementing a bitstream decompressor, it was examined to include a decompression accelerator directly within the FPGA. In order to determine the area overhead for this approach, the 8-bit LZSS decompressor including a 16 word input FIFO has been synthesized by the use of the Synopsis Design Compiler (version Y-2006.06) and a 90nm, 1.0V supply voltage standard cell library. For the standard cell implementation, the original RTL model of the LZSS FPGA implementation was reused. Only the SRL16 shift register primitives available on Xilinx FPGAs have been reimplemented. The size of the accelerator was reported to be 0.01 mm² when selecting the target clock frequency to 500 MHz. The reported area is estimated to be about the same as one FPGA Spartan-3 CLB that is fabricated in the same process technology. The largest Spartan-3 FPGA, for instance, provides 6912 CLBs. With an assumed area of 0.01 mm² per CLB, this results into 69.12 mm² for all CLBs, thus proving that the area estimates are realistic. The highly optimized decompressor implementation on the FPGA fabric requires 13 CLBs. Note that all LZSS decompression accelerators can emit one byte at each clock cycle if no stall is requested by the configuration source.

4.3.6 High-speed Defragmentation

As shown throughout the last sections, runtime reconfiguration can be accomplished fast at high configuration data rates. However, if modules are loaded and replaced at runtime, the reconfigurable area may get fragmented, which means that not all resource slots form a continuous area. This is known as external fragmentation. If assuming that a module requires resource slots as one complete consecutive area, the number of free slots alone is not sufficient to decide whether a module can be placed. By rearranging the module layout among the resource slots, the area with free consecutive resource slots can be increased, as illustrated in Figure 4.22. The example shows that module \( m_2 \) has to be moved before placing \( m_4 \). As the defragmentation arranges the module layout into larger blocks of continuous free slots and continuous occupied slots, this process is also called compaction [FKS08]. Dependent on the entire task of a moved module, there are different scenarios possible of how the defragmentation influences the execution of a particular module:
4.3 High Speed Reconfiguration

Figure 4.22: a) Scenario where module $m_4$ cannot be placed despite that there are sufficient free resource slots available because of the free slots are not continuous (external fragmentation); b) by moving module $m_2$ to slot 0, the reconfigurable area is defragmented and $m_4$ can be placed.

1. Restarting with a new iteration on an additional instance of $m_2$ allows to perform the defragmentation without interrupting the task execution. This is only applicable if there are no data dependencies between different iterations of $m_2$. The lane detection in the ReCoNets demonstrator, for example, as shown in Figure 2.39, can be restarted at another placement position after accomplishing the segmentation within a full frame of the input video stream.

2. Transferring the internal state to a new instance of $m_2$ before its restart is required when $m_2$ demands some results from preceding iterations. This means that $m_2$ contains an iteration dependent internal state. Note that the module reconfiguration can be carried out before the state transfer for continuing execution in parallel when the new instance of $m_2$ is reconfigured at the new placement position.

3. If the module to be moved cannot be fully placed to the new position, one or more modules i) have to be stopped, ii) their internal state has to be captured, then, iii) the modules can be placed to the new position, and finally, iv) can be restarted after restoring their internal state. For example, instead of moving $m_2$ in the example in Figure 4.22 a), it is alternatively possible to move $m_1$ either one slot to the right or two slots to the left for placing $m_4$. In both cases, module $m_1$ has to stop execution as it must be partially overwritten with its displaced instance.

In the first scenario, no internal state of a moved module is considered. This restricted model has been examined by Platzner [WP02] for two-dimensional module placement where modules of any arbitrary shape (not necessary rectangular) are replaced only after task completion. In the last two scenarios, a module is preempted at one position and resumed somewhere else for enlarging the continuous free resource area. The scenarios point out that defragmenting an FPGA module layout can comprise either an overhead in terms of free resource slots (space), in terms of stopping module execution (time), or a mixture of space and time. The following
sections depict contributions for speeding up defragmentation and for optimizing the defragmentation process.

**Hardware supported FPGA Defragmentation**

If a hardware module has to be stopped, state extracted, and resumed later, the hardware checkpointing techniques presented in Section 2.6.3 can be applied. However, when using these techniques for defragmenting an FPGA, a module is migrated within the same FPGA and not among some ReCoNodes. Therefore, configuration data is copied within the same FPGA only. This can be exploited by extending the FPGA configuration logic as it is possible to transfer data on-chip at rates that are orders of magnitudes beyond of-chip communication via a configuration port.

Brebner and Diessel proposed in [BD01] for a one-dimensional slot-based reconfigurable system to arrange all configuration SRAM cells as shift registers such that a full configuration can be shifted further by one logic column within a single clock cycle. The shifting can be individually masked for each logic column, such that in the case that a module is terminated all succeeding slots can be shifted towards the next used preceding slot. This allows compacting the reconfigurable area within $w^{BE}$ clock cycles after terminating a module being $w^{BE}$ logic columns wide (in terms of basic elements; see also Section 3.3). The main restriction of this approach is that placed modules are obstacles for the shift process, hence, not allowing inhomogeneous resources such as memory columns to be located within the reconfigurable area.

For allowing obstacles when compacting the module layout, Compton et al. suggested to move modules via a shadow register capable to store one full logic column [CLC+02]. In this approach, a module is moved by reading the first logic column of a module from its original position to a shadow register and writing it from there to the new position. This process takes two clock cycles per logic column and is repeated for all other logic columns until the complete module is moved. This technique requires only small changes to the current configuration logic of Xilinx FPGAs that already contains shadow registers. However, in Xilinx FPGAs there exist two independent registers, one for configuring the fabric and a separate one for reading back configuration data. Furthermore, the shadow registers can only hold a frame that contains a subset of the configuration bits ($\frac{1}{11}$ in the case of a Virtex-II device) required to configure a full logic column. Consequently, moving a module would take the amount of frames per logic column times longer on a Xilinx FPGA, if the two shadow registers will be merged.

Both the module shifting [BD01] and the shadow register approach [CLC+02] are limited to compact a one-dimensional module layout with slots spanning the entire height of an FPGA. For supporting two-dimensional grid style reconfigurable systems, an according enhancement was developed that also uses a shadow register. But, as compared to [CLC+02] access to the shadow register can be masked and the
### 4.3 High Speed Reconfiguration

A maskable subcolumn can be selectively copied into the row shift register, then ring-shifted, and selectively written to another maskable subcolumn. Note that one mask register pair can control access to all configuration SRAM cells within a row of logic basic elements.

For configuring the FPGA fabric with a new module, the write mask register chain is updated with the according row write mask and the configuration data is shifted through the row shift register to its target position. The configuration bitstream contains only data for the masked region and the configuration of the different subcolumns (or subframes) is concatenated to one module string. By activating all mask bits, the device configuration would follow the manner provided in Xilinx Virtex-II...
Figure 4.24: Shift register-based defragmentation. a) If some rows overlap ($\Delta_{AB} < h^{BE}$), it requires $(2 + \Delta_{AB}) \cdot w^{BE}$ clock cycles to move the complete module to the new position, because reading and writing the row shift register cannot be pipelined as the column decoder is only allowed to activate one column at a point of time (see also Figure 4.23). b) In the case that the source and the destination have no overlapping rows, a module can be moved within $2 + w^{BE} \cdot h^{BE} + (\Delta_{AB} - h^{BE})$ clock cycles. This holds only true if there is no simultaneous read and write operation to the row shift register required. In a), the read mask is set to $11111000$ and the write mask to $00011111$. In b), the according settings are $00001110$ for the read mask and $11100000$ for write mask register.
4.4 Chapter Summary and Contributions

FPGAs.

Figure 4.24 exemplarily reveals how a maskable row shift register chain is used for defragmenting an FPGA. By firstly loading a particular module column to the register and then shifting this data to the target row position, a module can be moved completely two-dimensional within the FPGA. Because of the high on-chip bandwidth, this process can be carried out quickly. With $\Delta_{AB}$ denoting the vertical displacement and $w^{BE}$ and $h^{BE}$ stating the width and height of a module in basic elements, the amount of clock cycles for moving this module is:

$$t_{\text{defrag}} = \begin{cases} (2 + \Delta_{AB}) \cdot w^{BE} & \text{if } \Delta_{AB} < h^{BE} \\ 2 \cdot w^{BE} \cdot h^{BE} + (\Delta_{AB} - h^{BE}) + w^{BE} & \text{else} \end{cases}$$

(4.7)

When applying this technique to the column wise reconfigurable Virtex-II-6000 FPGA of the ESM, the width $w^{BE}$ has to be multiplied with the number of frames per logic column which is 21 for this device. For a quantitative example related to this platform, a module of 3000 look-up tables is assumed, what corresponds to the logic of an average 32-bit soft-core CPU. When such a module is located in the top half of the device, it will occupy a block of $w^{BE} \times h^{BE} = 8 \times 48$ consecutive configurable logic blocks (CLB). According to Equation 4.7, moving this module to another position in the bottom half of the device ($\Delta = 48$) requires $2 + 8 \cdot 21 \cdot 48 + (48 - 48) + 8 \cdot 21 = 8234$ clock cycles. At 100 MHz, this corresponds to 82.34 $\mu$s for the complete module move. The total amount of configuration data transferred in this example is 80640 byte. For the assumed 100 MHz this results in a data rate of 979 MB/s. For this high transfer rate, it requires 80 flip-flops within the row shift register per CLB row and according 80 vertical wires downwards and upwards for implementing the ring-shifting.

4.4 Chapter Summary and Contributions

A gap in methodologies and tools is one of the major reasons why partial runtime reconfiguration is still very exotic and only rarely used in commercial applications. In this chapter, a novel framework called ReCoBus-Builder [KBT08] was presented that closes this gap. Based on new and efficient realizations of partially reconfigurable bus-systems using the proposed ReCoBus and connection bar communication architecture, the framework assists in specifying the system, simulating reconfiguration processes, partitioning of the FPGA resources and performing the physical implementation of the static system as well as all partial modules. The last step includes in particular the bitstream generation.

In addition to the offline design aspects, sophisticated reconfiguration techniques have been implemented. This allows to relocate modules arbitrarily in a two-dimensional manner within a reconfigurable area on an FPGA. Despite the full reconfiguration scheme of the main FPGA of an ESM [MTAB07], this was demonstrated especially for this platform. Furthermore, bitstream manipulation techniques have been
presented. With these techniques, the setting of the switches of the circuit-switching network, presented in Section 3.7.5, have been adjusted directly within the routing fabric of an FPGA. The here presented two-dimensional circuit-switching approach is the first one that is fully carried out within the switch matrices and that is embedded in the modules without constituting any obstacle that may restrict the module placement.

For speeding-up the task of hardware task migration and repair processes in a ReCoNet and for accelerating the reconfiguration process in general, configuration prefetching, bitstream decompression, and FPGA defragmentation techniques have been revealed. It was shown that it is possible to decompress configuration data at the full configuration speed of 400 MB/s, achievable with today’s high-end FPGAs, whereas the implementation cost is only 120 look-up tables [KBT07, KBT09b]. With this speed, a one million gate equivalent FPGA design can be exchanged within a millisecond.

When partially reconfigurable modules are loaded and terminated, the reconfigurable area may get fragmented, such that the free resources form no continuous area. This can prevent to place a module despite that the total amount of resources are sufficient. However, by compacting the module layout (the present placement position of the modules), a larger free consecutive space may be generated and a larger module may now being placed. But performing compaction takes an inappropriate amount of time when all configuration data has to be transferred via an configuration port. In this chapter, it was shown that very little additional logic within the FPGA configuration logic is sufficient for materially enhancing defragmentation speed by moving modules directly by the configuration logic within the FPGA [KBA07c].
5. Concluding Remarks

In this monograph, methods architectures and tools have been proposed for runtime reconfigurable systems based on FPGAs. The results provide material improvements in the efficiency of integrating reconfigurable modules into an FPGA, the tools to build such systems, and the capabilities to reconfigure a system at runtime. The key contributions are as follows:

**Operating system services:** For migrating hardware and software tasks and for changing the implementation style of a task by hardware/software morphing at runtime, corresponding OS services have been developed, analyzed, and implemented with the help of novel tools. All these services maintain internal states and can be provided transparently to the overlying application.

**Protocols and services for reconfigurable networks:** The migration and morphing of tasks demands in particular a flexible task-to-task communication that is transparent with respect to the current task binding and implementation style. Consequently, a complete protocol stack has been defined and implemented that is especially tailored to small fault-tolerant embedded control systems.

**Communication Architecture for module-based on-FPGA communication:** It has been proven that existing methodologies and communication architectures provide a tiling of the reconfigurable area that is too coarse-grained for most practical applications. With the novel ReCoBus and connection bar architecture, the size of the tiles have been reduced by more than an order of magnitude while reducing the cost to implement the complete infrastructure and to increase throughput at the same time. This is the key technology improvement required to efficiently implement runtime reconfigurable systems.

**Design flow and tools:** For implementing the ReCoBus and connection bar communication architecture, a powerful tool has been developed that is capable to map the communication infrastructure in a regular structured manner on an FPGA. Furthermore, this tool provides a floorplanner capable to generate all location and resource constraints required to build the static system or the configurable modules.

**Techniques for flexible high-speed runtime reconfiguration:** For partial runtime reconfiguration, techniques have been proposed that allow to place mod-
5. Concluding Remarks

**Figure 5.1:** Production cost of a hypothetical system over the units for an assumed ASIC or FPGA implementation (the values are taken from [Kot]). The FPGA has zero initial cost, but a much higher price per unit than compared with the ASIC solution. By utilizing partial runtime reconfiguration, the FPGA domain is shifted remarkable into higher volume regions, hence, squeezing the ASIC out of the mid range market.

With these methods and tools, a technical basics for achieving a considerable benefit over completely statically implemented systems on FPGAs has been provided and evaluated. This will enormously help other researchers in the field of reconfigurable computing to prove the advantages of adapting systems at runtime. Additionally, the results are relevant for commercial applications and electronic design automation in general. With partial runtime reconfiguration, systems can be implemented cheaper and with lower power consumption by using smaller devices. Furthermore, this is an option to add additional functionality to existing systems. Even if only a relatively small part of the worldwide 3.6 bn US$ FPGA market [EW0] can substantial benefit from runtime reconfiguration, the possible savings are many tens of millions US$ per year. However these savings will not decrease the total FPGA market, as illustrated in Figure 5.1. Hence, partial runtime reconfiguration is advantageous for both, FPGA users and vendors.

A further important aspect is the fully encapsulated component-based design methodology that is advantageous for FPGA design in general. With the here presented design flow, implemented and foremost fully placed and routed modules are composed together to complex systems based on interface specifications that also include the timing behavior. This enormously simplifies the system integration phase and
addresses the demands of an easy design reuse to close the design productivity gap, as sketched in Figure 5.2. With introducing design reuse, more functionality can be designed in less time and fully implemented modules will in particular ensure timing and reduce the module test expenses. Furthermore, the encapsulated design methodology prevents that changes in one part of the system will influence the rest of the system. Also the synthesis, and place and route itself will be accelerated as they can be carried out fully parallel for each module and because of the reduced problem size by partition the implementation steps in smaller units.

### 5.1 Future Work

The field of research on runtime reconfigurable systems is huge and in this thesis focus was put on technical aspects required to efficiently implement according systems based on FPGAs. Consequently, there remain many challenging open problems for future research.

One very useful enhancement for increasing the performance on the ReCoNodes in the system would be a separate accelerator for the message passing task-to-task communication. In the present system, all data is transferred via the CPU, which in particular is a bottleneck if a node hosts multiple concurrently running hardware tasks. Such a communication accelerator should further be able to invoke the network if messages have to be send among the nodes in a ReCoNet. The here defined ReCoNet protocol is a good starting point as it is designed especially for the demands of small distributed embedded control systems and as the protocol stack is suitable for a full hardware implementation. The long-term objective for the task-to-task communication is a uniformed message passing system for hardware and software tasks. This would constitute an interface being identical for any combination of hardware
5. Concluding Remarks

or software task, thus, matching ideal to the proposed hardware/software morphing.

Real-time processing is an important criterion for distributed embedded systems and this topic is only covered for the communication protocol definition. An interesting option for future distributed systems would be to couple the communication network with the local schedulers running on the different nodes in a ReCoNet. This allows to guarantee lower latencies by reducing the memory requirements at the same time. In a sensor-controller-actuator chain, for example, sensor data may be captured and sent directly to an active controller task where it will be immediately processed and passed further to an also active actuator task without interrupting other tasks on the nodes.

For further enhancing the design productivity beyond the here presented component-based design methodology, the ReCoBus-Builder tool should be coupled with modern design languages and behavioral compilers, such as Catapult-C from Mentor Graphics or Impulse C from Impulse Accelerated Technologies. This would combine the advantages of behavioral synthesis with the advanced system integration capabilities provided by the ReCoBus and connection bar communication architecture in combination with the tool ReCoBus-Builder. Behavioral compilers allow communication constructs that may be automatically mapped to a customized or standardized communication architecture. For instance, pointers may result in a bus master interface to a ReCoBus and streaming ports may be mapped to corresponding connection bars.

Finally, the here presented methodologies have to be permanently ported to the newest devices to keep track with the progress in FPGA evolution. This will rise to reconfigurable systems providing lower cost per logic function, higher performance, and lower power consumption in future self-adaptive systems.
6. German Part

Architekturen, Methoden und Werkzeuge für verteilte Systeme basierend auf zur Laufzeit rekonfigurierbarer FPGA-Technologie
Zusammenfassung


Zur Realisierung dieser Techniken sind mehrere Werkzeuge und entsprechende Entwurfsprozesse entwickelt worden, die den Zugriff auf den internen Zustand von Hardware-Prozessen und die Transformation zwischen den Zuständen eines Hardware- und eines Software-Prozesses erlauben. Um Module zur Laufzeit durch partielle Selbstrekonfiguration auf einen FPGA Baustein integrieren zu können, ist ferner eine spezielle Kommunikationsarchitektur entwickelt worden, die gegenüber existierenden Ansätzen, die Kosten für die Integration von Hardwaremodulen um mindestens eine Größenordnung verbessert. Dabei wurde gleichzeitig eine deutlich höhere Flexibilität als auch ein höherer Datendurchsatz erreicht. Hierdurch sind die technischen Grundlagen für die Implementierung von feingranularen rekonfigurierbaren Systemen geschaffen worden, die effizienter, durch die Verwendung kleinerer und somit kostengünstigeren und auch stromsparenderen FPGAs umgesetzt werden können, als dieses durch eine klassische rein statische Lösung möglich wäre.

Mit der Entwicklung eines komfortablen Werkzeugs zur automatischen Synthese der Kommunikationsarchitektur, sowie zur Integration des rekonfigurierbaren Systems auf einem FPGA, ist der Entwurfsprozess von zur Laufzeit rekonfigurierbaren FPGA-Systemen gegenüber allen vergleichbaren Lösungen erheblich vereinfacht worden. Dabei sind gleichzeitig die Möglichkeiten, zum Beispiel zur freien und zweidimensionalen Platzierung von Modulen auf einem FPGA, sinnvoll erweitert worden.

Beiträge

Methoden und Modelle


Als Prozessmodell ist dazu ein Automatenmodell herangezogen worden und so
Figure 6.1: Beispiel eines verteilten selbstorganisierenden und selbstreparierenden Systems zu Steuerung einer Automobilapplikation. a) Das ursprüngliche System besteht aus vier miteinander vernetzten Steuergeräten (ECU1, ..., ECU4). b) Im Fall eines Leitungsdefekts wird der Datenverkehr automatisch auf die verbliebenen Ressourcen umgeleitet. c) Im Fall des Ausfalls eines Steuergeräts, kann die Funktionalität auf andere Steuergeräte verlagert werden, und somit die Funktion aufrecht erhalten werden. d) Neue Prozesse (hier Prozess N), können zur Laufzeit in das System integriert und später beliebig terminiert werden. e) Das System erlaubt Topologieänderungen, wie beispielsweise das Hinzufügen eines Knotens. Die Fähigkeit des Systems zur Selbstoptimierung kann die Migration von Prozessen bewirken. Dabei ist die Ausführung des Prozesses nicht durch die Migration beeinträchtigt.
transformiert worden, dass zum einen die Prozessausführung kontrollierbar gemacht
wurde und zum anderen auf den internen Zustand des Prozesses zugegriffen wer-
den kann. Dieses ist vergleichbar mit Checkpointing-Techniken, die für verteilte
Software-Systeme bekannt sind. Zusätzlich ist gezeigt worden, dass es möglich ist,
den Zustand eines Hardware-Prozesses in den Zustand eines Software-Prozesses zu
übersetzen und umgekehrt. Mit dieser Methode ist ein Morphing der Implementie-
tierung eines Prozesses zwischen Hardware- oder Softwareausführung realisiert wor-
den. Dieses erlaubt einen zusätzlichen Freiheitgrad der Rekonfiguration und damit
beispielsweise der Kompensation von Fehlern, der bislang ungenutzt war.

Architekturen

Mit der Entwicklung einer speziell auf FPGAs angepassten Kommunikationsarchitek-
tur, welche partiell rekonfigurierbaren Modulen sowohl eine direkte Busankopplung,
as auch Zugang zu einem Circuit-Switching Netzwerk erlaubt, konnte die Gran-
ularität, also die Kachelung der FPGA-Ressourcen in Zellen zur Aufnahme par-
tieller Module, erheblich verfeinert werden. Dadurch ist insbesondere der Verschnitt
(interne Fragmentierung), der durch die Implementierung von Modulen in solche
Kacheln entsteht entsprechend reduziert worden. Wie analytisch gezeigt wurde, stellte
dieser Verschnitt bislang den größten Anteil des Logik-Overheads dar, der in Kauf
genommen werden muss, wenn partielle Laufzeitrekonfiguration eingesetzt wird.

Werkzeuge

Das hier verwendete Modell von transformierten Zustandsmaschinen für die Migra-
tion von Prozessen ist für Hardware-Module mit dem Werkzeug StateAccess automa-
tisiert worden und die notwendigen Transformationen sind auf Netzlistenebene re-
alisiert worden. Dabei sind verschiedene Verfahren zur Kontrolle der Ausführung
von Hardwaremodulen als auch Methoden für den Zugriff auf den internen Zustand
twickelt worden. Im Einzelnen sind Scan-Chain-Verfahren für sowohl flächenop-
timierte als auch latenztimizierte Verfahren analysiert und implementiert worden.
Als alternativer Ansatz ist das Einblenden des Hardware-Zustands (im Wesentlichen
sämtliche Flipflops und Speicherblöcke) in den Adressraum eines Prozessors betrach-
tet worden.

Die bislang verfügbaren Werkzeuge und Verfahren zur Implementierung von zur
Laufzeit rekonfigurierbaren Systemen auf FPGAs besitzen viele Einschränkungen,
insbesondere wenn Module frei platziert, also auch relokiert werden sollen. Mit dem
Werkzeug ReCoBus-Builder ist ein ganzes Framework geschaffen worden, dass zum
einen die oben beschriebene Kommunikationsarchitektur synthetisieren, platziern
und verdrahten kann als auch die Planung der FPGA-Ressourcen unterstützt. ReCo-
Bus-Builder ist nahtlos in die Werkzeugkette des FPGA Herstellers Xilinx integri-
ert worden und stellt sämtliche Funktionen, welche zusätzlich zum Entwurf eines rein statischen Systems benötigt werden, bereit. Dieses beinhaltet, zusätzlich zur Synthese der Kommunikationsarchitektur, die Generierung der Konfigurationsdatenströme der partiellen Module.

**Ergebnisse**


Bibliography


[BLC07] Tobias Becker, Wayne Luk, and Peter Y. K. Cheung. Enhancing Relo-
catability of Partial Bitstreams for Run-Time Reconfiguration. In Pro-
ceedings of the 15th Annual IEEE Symposium on Field-Programmable
Custom Computing Machines (FCCM), pages 35–44, Napa, California,

[BM02] Luca Benini and Giovanni De Micheli. Networks on Chips: A New

[BMG06] Carsten Bieser and Klaus-D. Mueller-Glaser. Rapid Prototyping De-
sign Acceleration Using a Novel Merging Methodology for Partial
Configuration Streams of Xilinx Virtex-II FPGAs. In Proceedings of the 17th IEEE Int. Workshop on Rapid System Prototyping (RSP),
pages 193–199, Los Alamitos, CA, USA, 2006. IEEE Computer So-
ciety.

[BMN+01] Volker Baumgarte, F. May, Armin Nückel, Martin Vorbach, and
Markus Weinhardt. PACT XPP - A Self-Reconfigurable Data Pro-
on Engineering of Reconfigurable Systems and Algorithms (ERSA),

Checkpointing (UT-CS-94-269). Technical report, Knoxville, TN,
USA, 1994.

[BQF99] Roberto Baldoni, Francesco Quaglia, and Paolo Fornara. An Index-
Based Checkpointing Algorithm for Autonomous Distributed Sys-
tems. IEEE Transactions on Parallel and Distributed Systems,

[BR97] Vaughn Betz and Jonathan Rose. VPR: A New Packing, Placement
and Routing Tool for FPGA Research. In Proceedings of the 7th In-
ternational Workshop on Field-Programmable Logic and Applications

[Bre97] Gordon Brebner. The Swappable Logic Unit: A Paradigm for Virtual
Hardware. In Kenneth L. Pocock and Jeffrey Arnold, editors, IEEE
Symposium on FPGAs for Custom Computing Machines (FPGA),

Limitations in FPGA-based Logic Emulators. In IEEE Workshop on


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>[CS1]</td>
<td>Department of Computer Science 12, ReCoNets-Project website, University of Erlangen Nuremberg, Germany, <a href="http://www.reconets.de">www.reconets.de</a>.</td>
</tr>
</tbody>
</table>


[HUWB05] Michael Hübner, Michael Ullman, Florian Weisssel, and Jürgen Becker. Real-time Configuration Code Decompression for Dy-


Bibliography


[SWT09] Joon Edward Sim, Weng-Fai Wong, and Jürgen Teich. Optimal Placement-aware Trace-based Scheduling of Hardware Recon-
figurations for FPGA Accelerators. In Proceedings of the 17th Annual IEEE Symposium on Field-Programmable Custom Computing Ma-
chines (FCCM), Napa, CA, USA, April 2009. IEEE Computer So-

[Syn08] Synopsys Inc. DesignWare GTECH Library, September 2008. available online: www.synopsys.com


[TCJW97] Steve Trimberger, Dean Carberry, Anders Johnson, and Jenifer Wong. A time-multiplexed FPGA. In Proceedings of the 5th IEEE Sympo-

[Tei07] Jürgen Teich. Lecture on Reconfigurable Computing: Chapter 1. In-
roduction, University of Erlangen-Nuremberg, 2007. available online:

1998.


[TS84] Yuval Tamir and Carlo H. Séquin. Error Recovery in Multicomput-


Bibliography


Bibliography


Author’s Publications


Author’s Publications


Author’s Publications


### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>BIST</td>
<td>Built-In Self-Test</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block RAM</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CIF</td>
<td>Common Intermediate Format</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CCITT</td>
<td>Comité Consultativ International Télégraphique et Téléphonique</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>ESM</td>
<td>Erlangen Slot Machine</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out memory</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU’s Not Unix</td>
</tr>
<tr>
<td>HDLC</td>
<td>High-Level Data Link Control</td>
</tr>
<tr>
<td>ICAP</td>
<td>Internal Configuration Access Port</td>
</tr>
<tr>
<td>IMEC</td>
<td>Interuniversity Microelectronics Centre</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ISP</td>
<td>Instruction Set Processor</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
</tbody>
</table>

*Stan Kelly-Bootle gave in *The Computer Contradictionary* the following definition for an ALU: “A random-number generator supplied as standard with all computer systems”.*
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MISRA</td>
<td>Motor Industry Software Reliability Association</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MTTF</td>
<td>Mean Time To Failure</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>NTP</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OS4RS</td>
<td>Operating System for Reconfigurable Systems</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PCIe</td>
<td>Peripheral Component Interconnect Express</td>
</tr>
<tr>
<td>ppm</td>
<td>pulses per million</td>
</tr>
<tr>
<td>PTP</td>
<td>Point To Point</td>
</tr>
<tr>
<td>RDMA</td>
<td>Remote Direct Memory Access</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Layer</td>
</tr>
<tr>
<td>SGC</td>
<td>Secure Group Communication</td>
</tr>
<tr>
<td>SPDIF</td>
<td>Sony Philips Digital interface</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time Division Multiple Access</td>
</tr>
<tr>
<td>TMH</td>
<td>Task Message Handler</td>
</tr>
</tbody>
</table>
Symbols

\( b \) .................................................. input buffer size of a ReCoNet router
\( b_{CLB} \) .................................................. bit offset within an CLB
\( b_{\text{init}} \) .............................................. initial configuration data offset for the first CLB
\( B \) ............................................................ buswidth in bit
\( BE \) ........................................................... basic element
\( c \) ............................................................ communication cost in look-up tables
\( c_r \) ........................................................... configuration readback data rate
\( c_w \) ........................................................... configuration write data rate
\( c_{\text{const}} \) ................................................. constant cost for providing a configuration interface in look-up tables
\( C^S \) ......................................................... channel of span \( S \)
\( CLB_{\text{h}} \) .................................................. configuration data per CLB in one frame
\( CLB_{\text{x}} \) .................................................... horizontal CLB position
\( CLB_{\text{y}} \) .................................................... vertical CLB position
\( \Delta \) .......................................................... vertical module displacement
\( \eta \) ........................................................... compression ratio
\( \eta \) ........................................................... worst case compression ratio
\( \hat{\eta} \) ....................................................... \( \hat{\eta} \) of a Huffman compressor
\( \eta_{\text{LZSS}} \) ................................................. compression ratio of a LZSS compressor
\( \hat{\eta}_{\text{LZSS}} \) ............................................. \( \hat{\eta} \) of a LZSS compressor
\( \eta_{\text{RLE}} \) .................................................. compression ratio of a run length compressor
\( \hat{\eta}_{\text{RLE}} \) .............................................. \( \hat{\eta} \) of a run length compressor
\( d_{\text{FPGA}} \) .................................................. FPGA peak configuration data rate
\( d_{\text{MEM}} \) ................................................... memory data rate
\( \Delta_{\text{AB}} \) .................................................. vertical displacement between two modules in terms of BEs
\( f \) ........................................................... clock frequency
\( f_b \) ........................................................ size of the configuration frame buffer in frames
\( f_h \) ........................................................ size of a single frame
\( f_m \) ......................................................... number of frames of a particular module
### Symbols

- $F_{\text{max}}$: maximal clock frequency of a module
- $\gamma$: memory bandwidth limitation factor
- $h$: module height in resource slots
- $h^{BE}$: module height in basic elements
- $|h_{\text{max}}|$: maximal Huffman code word size
- $|h_{\text{min}}|$: minimal Huffman code word size
- $H$: height of a resource slot in basic elements
- $i$: interface requirement
- $I$: finite set of module interface requirements (subset of $\mathcal{I}$)
- $I_{\cup}$: finite set of all reconfigurable module interface requirements (subset of $\mathcal{I}$)
- $\mathcal{I}$: finite set of interface requirements
- $IO_h$: configuration data per I/O cell in one frame
- $k$: number of look-up table inputs
- $l$: length in data items
- $\lambda$: number of attached resource slots
- $L$: logic overhead in look-up tables
- $L_{\text{DR}}$: dedicated read signal logic overhead (refinement of $L$)
- $L_{\text{DW}}$: dedicated write signal logic overhead (refinement of $L$)
- $L_{\text{SR}}$: shared read signal logic overhead (refinement of $L$)
- $L_{\text{SW}}$: shared write signal logic overhead (refinement of $L$)
- $m$: module $m \in \mathcal{M}$
- $m_h$: size of the module configuration data per frame
- $M$: maximal number of simultaneously integrated modules
- $\mathcal{M}$: finite set of modules
- $n$: node $n \in \mathcal{N}$
- $n_c$: number of configuration bits
- $N$: interleaving factor
- $\mathcal{N}$: finite set of nodes
- $o$: data offset
- $o_r$: overhead for configuration readback operations
- $o_w$: overhead for configuration write operations
- $O$: module configuration overhead in look-up tables
- $\bar{O}$: average of $O$ over all modules
- $\hat{O}$: peak value of $O$ over all modules
- $p$: number of ports receiving a burst
\( \tilde{p} \) .................................................... probability
\( p_m \) ........................................... bitstream manipulation processing speed
\( P \) ............................................. number of packets of a traffic shaping period
\( t_p \) ........................................... traffic shaping period in terms of packets
\( R \) ............................................... total number of resource slots
\( P^{BE} \) ....................................... horizontal amount of basic elements
\( R_H \) ......................................... number of horizontal resource slots
\( R_V \) ............................................ number of vertical resource slots
\( s \) ........................................... module reference (start) resource slot
\( \sigma \) ........................................ number of look-up tables per resource slot
\( S \) ............................................... wire span
\( \tau \) ........................................... token (data item emitted by a decompressor)
\(|\tau|\) ........................................ the number of bits to code a token
\( t_b \) ........................................... full buffer-based module configuration time
\( t_{conf} \) ........................................ delay of a configuration clock cycle
\( t_{defrag} \) ..................................... time for moving a module for a defragmentation step
\( t_h \) ............................................... router hop time
\( t_i \) ........................................... initial router delay
\( t_{opt} \) ......................................... optimal configuration speed
\( t_{prop} \) ........................................ propagation delay
\( t_p \) ........................................... traffic shaping period in terms of packets
\( t_{wc} \) ........................................ worst case end-to-end propagation delay
\( t_{pipe}^{wc} \) ................................ worst case propagation delay using pipelining
\( T \) ............................................... throughput
\( \mathcal{V} \) ........................................ finite set of tasks
\( \mathcal{V}^{HW} \) .................................. finite set of hardware tasks (refinement of \( \mathcal{V} \))
\( \mathcal{V}^{SW} \) .................................. finite set of software tasks (refinement of \( \mathcal{V} \))
\( w \) ............................................... module width in resource slots
\( w^{BE} \) ........................................ module width in basic elements
\(|w_{deco}|\) .................................. decompressor output word size
\(|w_{FPGA}|\) .................................. FPGA configuration port word size
\( W \) ............................................... resource slot width in basic elements