

ReCoNet: Modeling and Implementation of Fault Tolerant Distributed Reconfigurable Hardware*

Christian Haubelt, Dirk Koch, Jürgen Teich
Department of Computer Science 12
University of Erlangen-Nuremberg
{haubelt, dirk.koch, teich}@cs.fau.de

Abstract

Recent research was mainly focused on the OS support for a single reconfigurable chip. This paper presents a general approach to manage fault tolerant distributed reconfigurable hardware. In order to run such a system, three basic tasks must be implemented: (i) rerouting to compensate line errors, (ii) rebinding to compensate node failures, and (iii) hardware reconfiguration to allow the optimization of these systems during runtime. This paper proposes first ideas and solutions of these management functions. Furthermore, a prototype implementation consisting of four fully connected FPGAs will be presented.

1 Introduction

Distributed reconfigurable hardware platforms [7, 12] are becoming more and more important for applications in the area of automotive, body area networks, ambient intelligence, etc. The most outstanding property of these systems is the ability of hardware reconfiguration. In terms of system synthesis, this means that the binding of processes to resources is not static. In the context of FPGAs, recent research focuses on the OS support [14] by dynamically assigning hardware tasks to an FPGA.

In a network of connected FPGAs, it becomes possible to migrate hardware tasks from one node to another during the system operation. Thus, resource faults can be compensated by *rebinding* tasks to fully functional nodes of the network. The task of rebinding is also called *repartitioning* or *online partitioning*. A network of reconfigurable nodes that implements repartitioning will be termed *ReCoNet* in the following (see Figure 3(a) for an example of a ReCoNet).

This paper describes a graph-based model and first approaches in how to implement a ReCoNet. As a result, three basic tasks have to be implemented in order to run a Re-

CoNet. The first two tasks are (1) *rerouting* and (2) *repartitioning*. They deal with erroneous resources. Thus, we are able to compensate line errors by computing new routes for broken communications and we can migrate tasks from one node in the network to another. These two tasks are also necessary in any distributed system, not only in reconfigurable hardware systems. By using (3) *hardware reconfiguration*, we have the opportunity to efficiently optimize several objectives, like power consumption, latency, etc., simultaneously and during runtime. Note, this is not possible when using a software only approach. Furthermore, by just reconfiguring a single node an FPGA independent way of *partial reconfiguration* is established.

To the best of our knowledge, there is no implementation of a ReCoNet as described above. Reconfigurable Architectures like PACT [4] and Chameleon [6] are first approaches for coarse grained, networked processing units without providing support for online reconfiguration and optimization.

This paper is organized as follows: Section 2 introduces the concept and models of a so-called *ReCoNet*. Section 3 discusses the basic tasks needed to provide the necessary support for distributed reconfigurable hardware. Section 4 focuses on the issue how to configure a single node in such a network and how this node should react to changes of the network topology while section 5 presents our first simple prototype implementation of a ReCoNet.

2 Concepts and Models of a ReCoNet

In this paper, we consider embedded systems consisting of small networks of hardware reconfigurable nodes. The three main aspects are:

- **small:** Each node in the network can store the current state of the whole network. The state of the network is given by all available nodes, available connections, and the distribution of the processes in the network.
- **embedded:** requires the optimization of different objectives, like power consumption, latency, etc. simultaneously.

*Supported in part by the German Science Foundation (DFG), SPP 1148 (Rekonfigurierbare Rechensysteme) and DaimlerChrysler Research & Technology, REM / EB, Esslingen, Germany.

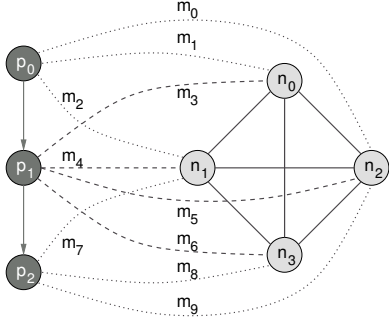


Figure 1. Model of a ReCoNet consisting of four nodes $n_0, n_1, n_2,$ and n_3 and three processes $p_0, p_1,$ and p_2 . The additional mapping edges m_i model possible bindings of processes onto nodes.

- hardware reconfiguration: Allows the implementation of arbitrary functions in hardware. Thus, it accelerates the computation of the corresponding functions required in the network.

A distributed reconfigurable system that possesses these properties is called *ReCoNet* in the following. Furthermore, a ReCoNet must support online repartitioning of processes in the network. Therefore, we implement three tasks. The first two tasks, namely the *rerouting* and *repartitioning* deal with erroneous resources. Thus, we can compensate line errors by computing a new routing for the broken communications and we can migrate processes from one node in the network to another. These two tasks are also necessary in all distributed systems, not only reconfigurable hardware systems. But based on these two tasks, we can define an FPGA independent way of *partial reconfiguration* and we can support online optimization of the ReCoNet. Here, the reconfiguration is partial for the whole network. Note, that all three tasks can be hardware supported. This is especially important in the case of line and node detection as described later. The three tasks will be discussed in Section 3.

In order to describe the three required tasks mathematically, we need an appropriate model: The behavior of the system is modeled by a *process graph* $g_p = (P, D)$ consisting of communicating processes $p \in P$. The data dependencies are modeled by edges $d \in D \subseteq P \times P$. The structure of the ReCoNet is given by a so-called *architecture graph* $g_a = (N, C)$, where N is the finite set of nodes and $C \subseteq N \times N$ is the finite set of connections in the network. In order to relate the behavior and the structure of our system, we use so-called *mapping edges* $m \in M \subseteq P \times N$ that relate processes $p \in P$ to nodes $n \in N$. For more detailed information about the model see [5, 10].

Figure 1 shows a specification of a ReCoNet. The set of processes P and data dependencies D are given by $P = \{p_0, p_1, p_2\}$ and $D = \{(p_0, p_1), (p_1, p_2)\}$, respec-

tively. This process graph models the coarse grain behavior of a distributed control system, where p_0 corresponds to a sample process sampling a sensor, p_1 corresponds to the control process implementing the actual control, and p_2 models the driver process driving an actuator.

The architecture graph in Figure 1 consists of the nodes $N = \{n_0, n_1, n_2, n_3\}$. Each reconfigurable node could directly communicate with each other, i.e., the architecture graph is a clique. The mapping edges $M = \{m_0, \dots, m_9\}$ indicate that the sample process p_0 may be executed on any reconfigurable node n_0 to n_2 and the driver process p_2 may be performed on any of the reconfigurable node n_1 to n_3 . The controller process p_1 could be bound to any of the reconfigurable nodes in the architecture graph.

Beside the specification of the ReCoNet, we need a representation for an implementation of a ReCoNet. Here, a ReCoNet implementation is given by a time variant allocation $\alpha(t)$ and time variant binding $\beta(t)$. The allocation $\alpha(t)$ is the set of all available nodes $n \in N$ and all available connections $c \in C$ at time t . Hence, $\alpha(t) \subseteq N \cup C$. The binding $\beta(t) \subseteq P \times N$ describes the execution of a process $p \in P$ on node $n \in N$ at time t .

3 Basic ReCoNet Tasks

Since ReCoNets are designed for a dynamically changing environment, e.g. some connections or nodes fail, or a new process must be created, etc., we must define a way to react to these changes. This section describes the basic tasks needed for running a ReCoNet.

3.1 Rerouting

The first tasks to be defined is the task of rerouting. Rerouting is required if a connection $c' \in C$ in the network fails at time t' , i.e., $\alpha(t)$ changes at time t' . All communications done over this connection has to be rerouted. There are several publications dealing with this issue. Recent work was mainly focused on probabilistic approaches [8]. Here, we consider a high-level fault tolerant approach. Rerouting itself can be decomposed in three subproblems:

1. Line detection: Is a link $c = (n_1, n_2)$ between two nodes n_1 and n_2 available or not?
2. Network state distribution: If a connection c' between two nodes n_1 and n_2 fails at time t , all nodes $n \in N \cap \alpha(t) \setminus \{n_1, n_2\}$ in the network not incident to c' must be informed.
3. Routing of failing communication links $d \in D$.

The first subproblem can be solved in several ways. The easiest implementation is to periodically send some predefined data over each connection $c \in C$. If we do not detect any signal changes at the other end of this connection, either

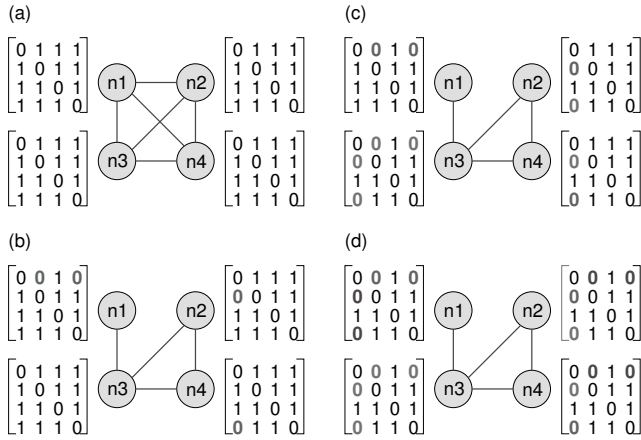


Figure 2. (a) Example of a ReCoNet consisting of four ReCoNodes and the corresponding incidence matrices. (b) Two connections failed. (c) All directed neighbors are notified. (d) All indirect neighbors are notified.

the line or the sending node may be defect. In both cases, we cannot use this connection any longer.

The second subproblem, the distribution of the network state, could also be solved in many ways. Again, we just mention the simplest one, the broadcast. The detecting node n_1 just sends this new information to all its neighbors $n \in N \mid (n_1, n) \in C$. These neighbors will relay this message until all nodes in the network are notified. The main problem in this solution is to keep track of already known messages which should not be relayed. Another problem is the time needed for the distribution. If an error is intermittent, there may be several different messages in the network all representing a different network states. Thus, also if we limit ourselves to applications without any time constraints, we need some time base in the network [11].

The last subproblem, the computation of a new routing can be done with any routing algorithm. Since we only consider small networks, we are able to store the state of all connections of the network in a so-called *incidence matrix* (see Figure 2). This matrix is of dimension $|N| \times |N|$. An element of the incidence matrix is non-zero if there is a direct connection between the two corresponding nodes. Based on this matrix, we can calculate the new routes using for example the shortest path algorithm (see [13]).

Figure 2 shows an example of the first two steps of rerouting. Figure 2(a) shows a ReCoNet consisting of four fully connected ReCoNodes n_1, n_2, n_3 , and n_4 . The corresponding incidence matrices are shown near each node. In Figure 2(b) the situation is shown where two connection $c_1 = (n_1, n_2)$ and $c_2 = (n_1, n_4)$ fail. Each incident node n_1, n_2 , and n_4 detects this error and updates its incidence matrix. In a next step (Figure 2(c)) all direct neighbors are informed about the topology change of the network. Note

that all nodes store different information of the state of the network. Finally, in Figure 2(d), all nodes have the same information about the state of the network. Based on this information, each node computes the new routing.

3.2 Repartitioning

Rebinding describes the migration of hardware processes from one node in the network to another, i.e. the binding $\beta(t)$ of processes onto nodes changes. Note that if we configure the reconfigurable nodes $n \in N$ with a processor, it may be possible to migrate hardware processes to software processes and vice versa, too. The task of rebinding is also called *repartitioning* or *online partitioning* in the following.

But when should we migrate a process? In this paper, we just focus on the case of resource faults, i.e., if a node in the network fails, all processes running on this node must be migrated to other nodes. In other words: the allocation $\alpha(t)$ changes at time t' , e.g., a node in the ReCoNet fails, leading to a new allocation. In general, the old binding $\beta(t)$ is no longer feasible for this new allocation. Hence, we have to perform a rebinding of the processes $p \in P$, leading to a new feasible binding. Thus, the task of repartitioning can be divided into two subproblems:

1. Detection of resource errors and
2. rebinding of tasks to nodes.

The first subproblem again can be solved by using the incidence matrix. If a node $n' \in N$ has no working connection to any of its neighbors at time t , i.e., $\forall c \in C \mid c = (n', n) \vee c = (n, n') : c \notin \alpha(t)$, it is called *isolated*. An isolated node n' cannot be used for process execution any more and all process $p \in P \mid (p, n') \in \beta(t)$ bound to this node at time t must be migrated.

An important question is how to perform a *save process migration*, i.e., how to keep track on the current state of a process. Here, we limit ourselves to stateless processes which can be started in an appropriate state. It is explained in [11] how to handle non-stateless processes. Furthermore, we must consider the question of on which node should a process be restarted. Again, there are several well-known approaches and the simplest one is to use a priority binding list for each process, i.e., each process is bound to the node with highest priority in its priority list. If this node fails, it is rebound to the next available node with the highest remaining priority, and so on. Note that when rebinding processes, we must recompute the routing as well.

The concepts of rerouting and repartitioning allow the compensation of resource faults in a ReCoNet. Obviously, there are limits regarding the capacity of the nodes (implemented as FPGAs) and the connections between the nodes. And, hence, we do not have infinite fault tolerance. A SAT-based approach to analyze the fault tolerance of a ReCoNet during design time is proposed in [10].

3.3 Reconfiguration

In order to be as general as possible, we do not require partial reconfiguration as provided by some Xilinx FPGAs [15]. By disconnecting a single node, the remaining network should not be affected. Thus, we can reconfigure a disconnected node completely. (A theoretical approach for online FPGA module placement can be found in [9].)

If we allow only full hardware reconfiguration, we must ensure that all tasks implemented on the given node are not needed during reconfiguration. If we are not sure about this, we first have to migrate these tasks to other nodes.

Figure 3 shows a scenario where an additional hardware process is loaded into the system. The initial state of the network is shown in Figure 3(a). Each of the four nodes executes software (circles) and hardware (rectangles) processes. Also, the configuration memory of each node is shown. Next, an additional hardware process (p9) should be loaded into the system. Here, we assume that p9 should be loaded on node ReCoNode4. Therefore, all processes (p7, p8) running on ReCoNode4 are suspended and restarted on ReCoNode2 and ReCoNode3, respectively (see Figure 3(b)). Note that process p8 is implemented in software. Furthermore, the new configuration for ReCoNode4 is stored into the configuration memory. In a last step (Figure 3(c)), the reset signal is issued on ReCoNode4. ReCoNode4 starts with the new configuration. The processes p7 and p8 on ReCoNode2 and ReCoNode3 are suspended.

4 Implementation of a ReCoNode

A ReCoNode must implement at least the three basic functions as described in Section 3. Figure 4(a) shows a simple example of a ReCoNode.

The reconfigurable hardware computes the new routing and partitioning based on its information about the state of the network. Note that the hardware could be configured with a processor (as will be presented in the next section) to do the required computation. The communication module is needed for sending and receiving packages over the network. Furthermore, it observes the state of each directly connected line. If the state of a line changes, the reconfigurable hardware will be notified. As described in Section 3, the reconfigurable node will then compute the new incidence matrix and will notify all other ReCoNodes in the network about the change.

To do the reconfiguration of the hardware, the ReCoNode itself can write the configuration memory and do the reset on the FPGA. That way, our solution does not require any partial reconfiguration of the FPGAs. In the simplest scenario, we just migrate all processes to other nodes of the network, write the desired configuration into the configuration memory, and do the reset on the FPGA (see also Figure 3). For non-stateless processes, more work is needed to migrate processes in a save way.

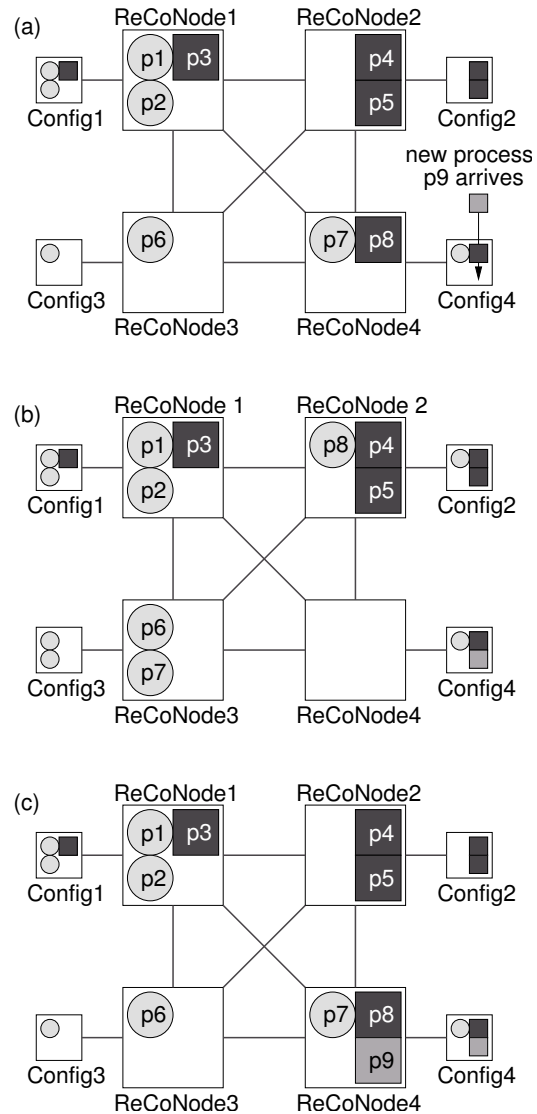


Figure 3. Example of Repartitioning. (a) shows the original ReCoNet configuration. There are four ReCoNodes. Each node is configured with software (circles) and hardware (rectangles) processes. The configuration memory for each node is also displayed. (b) To configure ReCoNode4 with an additional hardware process (p9), we must move all running processes of this node to other nodes in the network. At the same time the configuration memory (Config4) is updated with the additional process. (c) After resetting ReCoNode4 the new configuration is loaded.

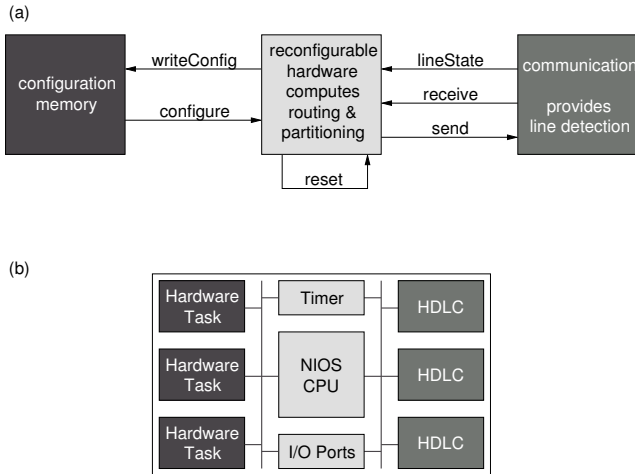


Figure 4. (a) Example of a ReCoNode. The reconfigurable hardware computes the routing and partitioning of the distributed application. The communication module provides information about local connections. (b) Configuration of a single ReCoNode. Beside the communication modules (HDLC) and the hardware tasks, there is a NIOS CPU with additional timers and I/O ports.

Figure 4(b) shows the configuration of a single ReCoNode as used in the sample implementation described in the next section. Beside the communication modules (HDLC), there is also a CPU and user-defined hardware. By using a CPU, we have the possibility to run processes in hardware as well as in software or any mixed implementation.

Figure 5 shows the reaction of a single ReCoNode on a network topology update. The `updateTopology` event may be caused by an line error or and broadcast over the network. In all cases the update event is relayed to all neighbors. Next, the ReCoNode checks if the binding $\beta(t)$ is still feasible, i.e., all processes can be bound. In this case, we compute the new routes and exit. If not, we must determine the new process binding $\beta(t)$ with a function `computeBinding` (see Figure 5).

The current configuration of the ReCoNode may not be appropriate for the new process binding. In that case, we have to perform a reconfiguration by: request the new configuration, transfer the new configuration, reset the ReCoNode, and transfer the current network state to this node. All four steps are initiated by the ReCoNode itself. But note, there is no exception handling in case there is no appropriate configuration stored somewhere in the ReCoNet. In our implementation (see next section), we assume that an appropriate configuration can be found in all cases. In future implementations, we will provide an external configuration server to handle these exceptions. Now, the configuration is

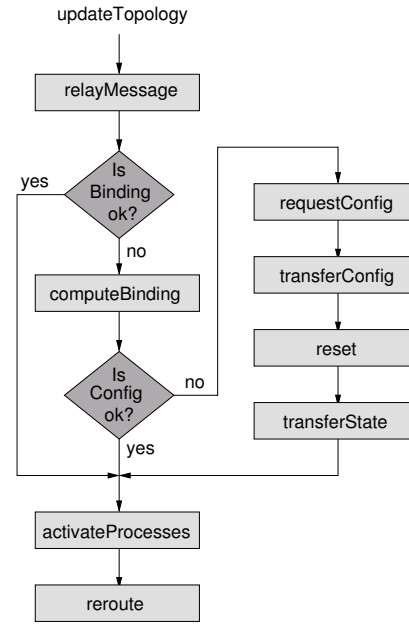


Figure 5. Reaction of a single ReCoNode on network topology update.

ready for process execution. In a last step, we just have to activate all processes and perform the rerouting.

5 A ReCoNet Implementation

In order to test our new approach, we have build a distributed reconfigurable prototype consisting of four Altera Excalibur development boards [2]. With a single Excalibur board, the user can develop a hardware-software-system by using Altera's SoPC-builder [1]. Such a systems is composed of Altera's NIOS processor [3], user-specified hardware, and user-specified software. The software is given in C/C++. The hardware and the NIOS processor are connected by using a hardware description language like VHDL or Verilog. After synthesis of the hardware design, it is downloaded to the FPGA on the Excalibur board.

A single ReCoNode of the prototype implementation is configured as described in section 4 with a NIOS processor and three communication modules (see also Figure 4(b)). For the communication we have chosen the well known HDLC (High-level Data Link Control) protocol. In order to detect line errors, we have implemented the physical layer with a Manchester encoding. See Figure 4(b) for the configuration of a single ReCoNode.

The prototype implementation of the ReCoNet is shown in Figure 6. Several automotive applications are implemented on this ReCoNet. The three basic tasks for running a ReCoNet are implemented as described in Section 3 and 4. These tasks form the OS layer of the fault tolerant distributed reconfigurable hardware.

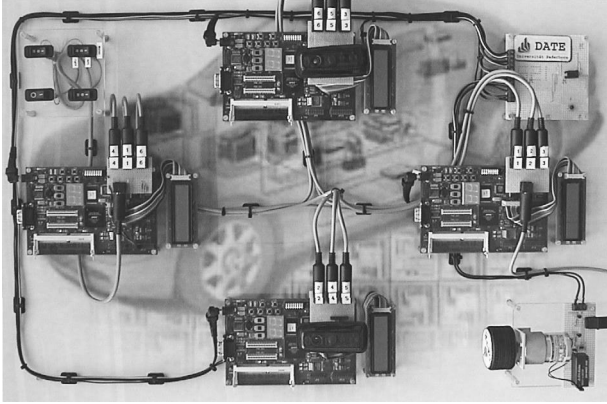


Figure 6. A prototype implementation of a distributed reconfigurable system that supports repartitioning.

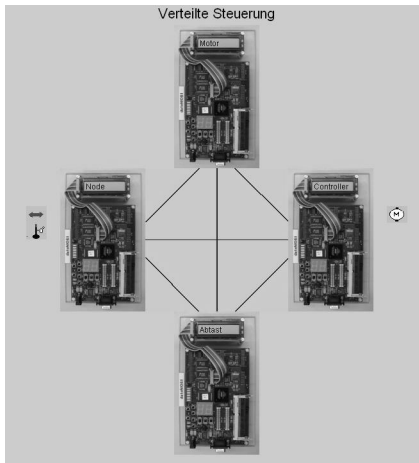


Figure 7. Monitoring tool for the prototype implementation. It shows the active connections and the process distribution.

In order to visualize the state of the network as well as the distribution of the applications, we have written a small monitoring JAVA program. This program communicates to a single Excalibur board over the debug port. A screenshot of the monitoring program is shown in Figure 7.

6 Conclusions and Future Work

In this paper, we have presented a model and an implementation of a fault tolerant distributed reconfigurable hardware system, a so-called *ReCoNet*. In order to run a *ReCoNet* three basic tasks must be considered. Two of these tasks deal with the fault tolerance of such a system. While the *rerouting* is used to compensate communication errors, we can compensate the defect of a node by *rebinding*. These two tasks together with the ability of *hardware reconfigura-*

tion allow to build dynamical hardware-software systems. A first implementation of these basic tasks in a distributed reconfigurable system consisting of four Altera FPGAs was presented here.

Important issues in the future are the optimization (online/offline) of distributed reconfigurable systems. Furthermore, the configuration management has to be considered as well as the real-time behavior of distributed reconfigurable hardware.

References

- [1] Altera. *Quartus Programmable Logic Development System & Software Data Sheet*, May 1999. <http://www.altera.com>.
- [2] Altera. *Excalibur Development Kit Data Sheet*, June 2000. <http://www.altera.com>.
- [3] Altera. *Nios Soft Core Embedded Processor Data Sheet*, June 2000. <http://www.altera.com>.
- [4] V. Baumgarten, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. In *ERSA*, Las Vegas, June 2001.
- [5] T. Blicke, J. Teich, and L. Thiele. System-Level Synthesis Using Evolutionary Algorithms. In R. Gupta, editor, *Design Automation for Embedded Systems*, 3, pages 23–62. Kluwer Academic Publishers, Boston, Jan. 1998.
- [6] Chameleon Systems. *CS2000 Reconfigurable Communications Processor, Family Product Brief*, 2000.
- [7] R. Dick and N. Jha. CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems. In *Proc. of ICCAD'98*, pages 62–68, 1998.
- [8] T. Dumitraş, S. Kerner, and R. Mărculescu. Towards On-Chip Fault-Tolerant Communication. In *Proceedings of the Asia and South Pacific Design Automation Conference 2003*, Kitakyushu, Japan, Jan. 2003.
- [9] S. P. Fekete, E. Köhler, and J. Teich. Optimal FPGA module placement with temporal precedence constraints. In *Proc. DATE 2001, Design, Automation and Test in Europe*, pages 658–665, Munich, Germany, March 13-16 2001. IEEE Computer Society Press.
- [10] R. Feldmann, C. Haubelt, B. Monien, and J. Teich. Fault Tolerance Analysis of Distributed Reconfigurable Systems Using SAT-Based Techniques. In *Proceedings of 13th International Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, Sept. 2003. to appear.
- [11] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, Massachusetts 02061 USA, 1997.
- [12] I. Ouais, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri. An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures. In *IPPS/SPDP Workshops*, pages 31–36, 1998.
- [13] A. Tanenbaum. *Computer Networks*. Prentice Hall, 2002.
- [14] H. Walder and M. Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Proceedings of Design, Automation and Test in Europe (DATE03)*, pages 290–295, Mar. 2003.
- [15] <http://www.xilinx.com>.