Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2006, Article ID 42168, Pages 1–15
DOI 10.1155/ES/2006/42168

Modeling and Design of Fault-Tolerant and Self-Adaptive Reconfigurable Networked Embedded Systems

Thilo Streichert, Dirk Koch, Christian Haubelt, and Jürgen Teich

Department of Computer Science 12, University of Erlangen-Nuremberg, Am Weichselgarten 3, 91058 Erlangen, Germany

Received 15 December 2005; Accepted 13 April 2006

Automotive, avionic, or body-area networks are systems that consist of several communicating control units specialized for certain purposes. Typically, different constraints regarding fault tolerance, availability and also flexibility are imposed on these systems. In this article, we will present a novel framework for increasing fault tolerance and flexibility by solving the problem of hardware/software codesign online. Based on field-programmable gate arrays (FPGAs) in combination with CPUs, we allow migrating tasks implemented in hardware or software from one node to another. Moreover, if not enough hardware/software resources are available, the migration of functionality from hardware to software or vice versa is provided. Supporting such flexibility through services integrated in a distributed operating system for networked embedded systems is a substantial step towards self-adaptive systems. Beside the formal definition of methods and concepts, we describe in detail a first implementation of a reconfigurable networked embedded system running automotive applications.

1. INTRODUCTION

Nowadays, networked embedded systems consist of several control units typically connected via a shared communication medium and each control unit is specialized to execute certain functionality. Since these control units typically consist of a CPU with certain peripherals, hardware accelerators, and so forth, it is necessary to integrate methods of fault-tolerance for tolerating node or link failures. With the help of reconfigurable devices such as field-programmable gate arrays (FPGA), novel strategies to improve fault tolerance and adaptability are investigated.

While different levels of granularity have to be considered in the design of fault-tolerant and self-adaptive reconfigurable networked embedded systems, we will put focus on the system level in this article. Different to architecture or register transfer level, where the 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 the topic of this contribution. A central issue of this contribution is *online hardware/software partitioning* which describes the procedure of binding functionality onto resources in the network at runtime. In order to allow for moving functionality from one node to another and execute it either on hardware or software resources, we will introduce the concepts of *task migration* and *task morphing*. Both task migration as well as task morphing require *hardware* and/or *software checkpointing* mechanisms and an extended design flow for providing an application engineer with common design methods.

All these topics will be covered in this article from a formal modeling perspective, the design methodology perspective, as well as the implementation perspective. As a result, we propose an operating system infrastructure for networked embedded systems, which makes use of dynamic hardware reconfiguration and is called ReCoNet.

The remainder of the article is structured as follows. Section 2 gives an overview of related work including dynamic hardware reconfiguration and checkpointing strategies. In Section 3, we introduce our idea of fault-tolerant and self-adaptive reconfigurable networked embedded systems by describing different scenarios and by introducing a formal model of such systems. Section 4 is devoted to the challenges when designing a ReCoNet-platform, that is, the architecture and the operating system infrastructure for a ReCoNet. Finally, in Section 5 we will present our implementation of a ReCoNet-platform.

2. RELATED WORK

Recent research focuses on operating systems for single FPGA solutions [1–3], where hardware tasks are dynamically assigned to FPGAs. In [1] the authors propose an online scheduling system that assigns tasks to block-partitioned devices and can be a part of an operating system for a reconfigurable device. For hardware modules with the shape of an arbitrary rectangle, placement methodologies are presented in [2, 3]. A first approach to dynamic hardware/software partitioning is presented by Lysecky and Vahid [4]. There, the authors present a *warp configurable logic architecture* (WCLA) which is dedicated for speeding up critical loops of embedded systems applications. Besides the WCLA, other architectures on different levels of granularity have been presented like PACT [5], Chameleon [6], HoneyComb [7], and dynamically reconfigurable networks on chips (DyNoCs) [8] which were investigated intensively too. Different to these reconfigurable hardware architectures, this article focuses too on platforms consisting of field-programmable gate arrays (FPGA) hosting a softcore CPU and free configurable hardware resources.

Some FPGA architectures themselves have been developed for fault tolerance targeting on two objectives. One direction is towards enhancing the chip yield during production phase [9] while the other direction focuses on fault tolerance during runtime. In [10] an architecture for the latter case that is capable of fault detection and recovery is presented. On FPGA architectures much work has been proposed to compensate faults due to the possibility of hardware reconfiguration. An extensive overview of fault models and fault detection techniques can be found in [11]. One approach suitable for FPGAs is to read back the configuration data from the device while comparing it with the original data. If the comparison was not successful the FPGA will be reconfigured [12]. The reconfiguration can further be used to move modules away from permanently faulty resources. Approaches in this field span from remote synthesis where the place and route tools are constrained to omit faulty parts from the synthesized module [13] to approaches, where design alternatives containing holes for overlying some faulty resources have been predetermined and stored in local databases [14, 15].

For tolerating defects, we additionally require checkpointing mechanisms in software as well as in hardware. An overview of existing approaches and definitions can be found in [16]. A *checkpoint* is the information necessary to recover a set of processes from a stored fault-free intermediate state. This implies that in the case of a fault the system can resume its operation not from the beginning but from a state close before the failure preventing a massive loss of computations. Upon a failure this information is used to *rollback* the system. Caution is needed if tasks communicate asynchronously among themselves as it is the case in our proposed approach. In order to deal with known issues like the *domino effect*, where a rollback of one node will require a rollback of nodes that have communicated with the faulty node since the last checkpoint, we utilize a *coordinated checkpointing* scheme [17] in our system. In [18] the impact of the checkpoint scheme on the time behavior of the system is analyzed. This includes the checkpoint overhead, that is, the time a task is stopped to store a checkpoint as well as the latencies for storing and restoring a checkpoint. In [19], it is examined how redundancy can be used in distributed systems to hold up functionality of faulty nodes under real-time requirements and resource constraints.

In the FPGA domain checkpointing has been seldomly investigated so far. Multicontext FPGAs [20–22] have been proposed, that allow to swap the complete register set (and therefore the state) among with the hardware circuit between a working set and one or more shadow sets in a single cycle. But due to the enormous amount of additional hardware overhead, they have not been used commercially. Another approach for hardware task preemption is presented in [23], where the register set of a preemptive hardware module is completely separated from the combinatorial part. This allows an efficient read and write access to the state by the cost of a low clock frequency due to routing overhead arising by the separation. Some work [24, 25] has been done to use the read back capability of Xilinx Virtex FPGAs in order to extract the state information in the case of a task preemption. The read back approach has the advantage that typically hardware design-flows are nearly not influenced. However, the long configuration data read back times will result in an unfavorable checkpoint overhead.

3. MODELS AND CONCEPTS

In this article, we consider networked embedded systems consisting of dynamically hardware reconfigurable nodes. The nodes are connected via point-to-point communication links. Moreover, each node in the network is able, but is not necessarily required, to store the current state of the entire network which is given by its current topology and the distribution of the tasks in the network.

3.1. ReCoNet modeling

For a precise explanation of scenarios and concepts an appropriate formal model is introduced in the following.

Definition 1 (ReCoNet). A ReCoNet $(g_t, g_a, \beta_t, \beta_c)$ is represented as follows.

(i) The *task graph* $g_t = (V_t, E_t)$ models the application implemented by the ReCoNet. This is done by communicating tasks $t \in V_t$. Communication is modeled by data dependencies $e \in E_t \subseteq V_t \times V_t$.

(ii) The *architecture graph* $g_a = (V_a, E_a)$ models the available resources, that is, nodes in the network $n \in V_a$ and bidirectional links $l \in E_a \subseteq V_a \times V_a$ connecting nodes.

(iii) The *task binding* $\beta_t : V_t \to V_a$ is an assignment of tasks $t \in V_t$ to nodes $n \in V_a$.

(iv) The *communication binding* $\beta_c : E_t \to E_a^i$ is an assignment of data dependencies $e \in E_t$ to paths of length i in the architecture graph g_a. A *path* p of length i is given by an i-tuple $p = (e_1, e_2, \ldots, e_i)$ with $e_1, \ldots, e_i \in E_a$ and $e_1 = \{n_0, n_1\}, e_2 = \{n_1, n_2\}, \ldots, e_i = \{n_{i-1}, n_i\}$.

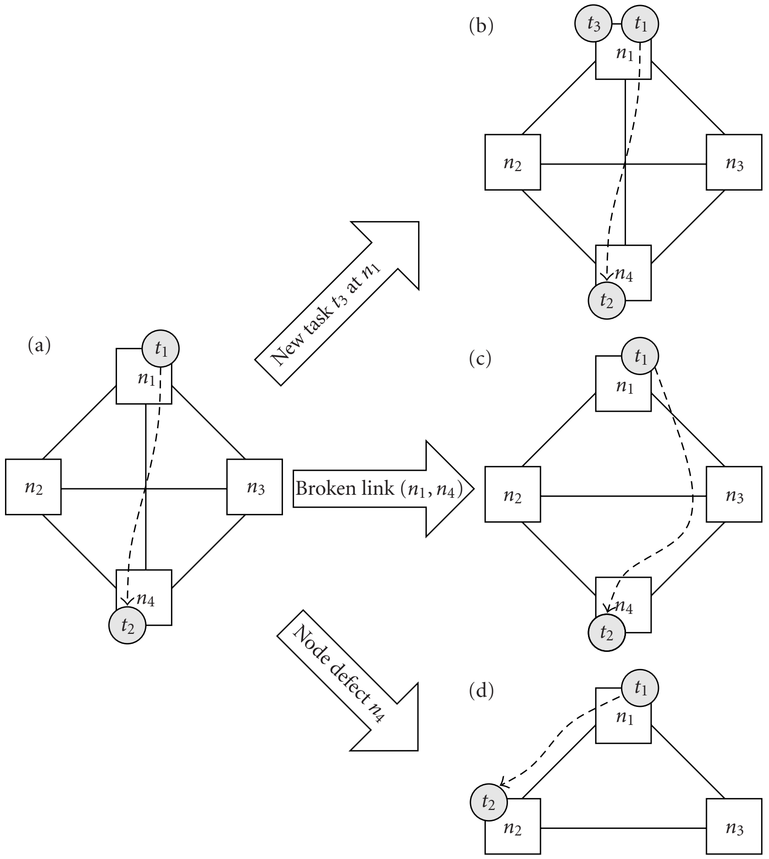


Figure 1: Different scenarios in a ReCoNet. (a) A ReCoNet consisting of four nodes and six links with two communicating tasks. (b) An additional task $t3$ was assigned to node $n1$. (c) The link $(n1, n4)$ is broken. Thus, a new communication binding is mandatory. (d) The defect of node $n4$ requires a new task and communication binding.

Example 1. In Figure 1(a), a ReCoNet is given. The task graph g_t is defined by $V_t = \{t1, t2\}$ and $E_t = \{(t1, t2)\}$. The architecture graph consists of four nodes and six links, that is, $V_a = \{n1, n2, n3, n4\}$ and $E_a = \{\{n1, n2\}, \{n1, n3\}, \{n1, n4\}, \{n2, n3\}, \{n2, n4\}, \{n3, n4\}\}$. The shown task binding is $\beta_t = \{(t1, n1), (t2, n4)\}$. Finally, the communication binding is $\beta_c = \{((t1, t2), (\{n1, n4\}))\}$.

Starting from this example, different scenarios can occur. In Figure 1(b) a new task $t3$ is assigned to node $n1$. As this assignment might violate given resource constraints (number of logic elements available in an FPGA or number of tasks assigned to a CPU), a new task binding β_t can be demanded. A similar scenario can be induced by deassigning a task from a node.

Figure 1(c) shows another important scenario where the link $(n1, n4)$ is broken. Due to this defect, it is necessary to calculate a new communication binding β_c for the data dependency $(t1, t2)$ which was previously routed over this link. In the example shown in Figure 1(c), the new communication binding is $\beta_c((t1, t2)) = (\{n1, n3\}, \{n3, n4\})$. Again a similar scenario results from reestablishing a previously broken link.

Finally, in Figure 1(d) a node defect is depicted. As node $n4$ is not available any longer, a new task binding β_t for task $t2$ is mandatory. Moreover, changing the task binding implies the recalculation of the communication binding β_c. The ReCoNet given in Figure 1(d) is given as follows: the task graph g_t with $V_t = \{t1, t2\}$ and $E_t = \{(t1, t2)\}$, the architecture graph consisting of $V_a = \{n1, n2, n3\}$ and $E_a = \{\{n1, n2\}, \{n1, n3\}, \{n2, n3\}\}$, the task binding $\beta_t = \{(t1, n1), (t2, n2)\}$ and communication binding $\beta_c = \{((t1, t2), (\{n1, n2\}))\}$.

From these scenarios we conclude that a ReCoNet given by a task graph g_t, an architecture graph g_a, the task binding β_t, and the communication binding β_c might change or might be changed over time, that is, $g_t = g_t(\tau)$, $g_a = g_a(\tau)$, $\beta_t = \beta_t(\tau)$, and $\beta_c = \beta_c(\tau)$, where $\tau \in R_0^+$ denotes the actual time. In the following, we assume that a change in the application given by the task graph as well as a change in the architecture graph is indicated by an event e. Appropriately reacting to these events e is a feature of *adaptive and fault tolerant systems*.

The basic factors of innovation of a ReCoNet stem from (i) *dynamic rerouting*, (ii) *hardware and software task migration*, (iii) *hardware/software morphing*, and (iv) *online partitioning*. These methods permit solving the problem of hardware/software codesign online, that is, at runtime. Note that this is only possible due to the availability of dynamic and partial hardware reconfiguration. In the following, we discuss the most important theoretical aspects of these methods. In Section 4, we will describe the basic methods in more detail.

3.2. Online partitioning

The goal of *online partitioning* is to equally distribute the computational workload in the network. To understand this particular problem, we have to take a closer look at the notion of *task binding* β_t and *communication binding* β_c. We therefore have to refine our model. In our model, we distinguish a finite number of the so-called *message types* M. Each message type $m \in M$ corresponds to a communication protocol in the ReCoNet.

Definition 2 (message type). M denotes a finite set of message types $m_i \in M$.

In a ReCoNet supporting different protocols and bandwidths, it is crucial to distinguish different demands. Assume a certain amount of data has to be transferred between two nodes in the ReCoNet. Between these nodes are two types of networks, one which is dedicated for data transfer and supports multicell packages and one which is dedicated for, for example, sensor values and therefore has a good payload/protocol ratio for one word messages. In such a case, the data which has to be transferred over two different networks would cause a different traffic in each network. Hence, we associate with each data dependency $e \in E_t$ the so-called *demand values* which represent the required bandwidth when using a given message type.

Definition 3 (demand). With each pair $(e_i, m_j) \in E_t \times M$, associate a real value $d_{i,j} \in \mathbb{R}_0^+$ (possibly ∞ if the message type cannot occur) indicating the *demand* for communication bandwidth by the two communicating tasks t_1, t_2 with $e_i = (t_1, t_2)$.

Example 2. Figure 2 shows a task graph consisting of three tasks with three demands. While the demand between $t1$ and $t2$ as well as the demand between $t1$ and $t3$ can be routed over all two message types ($|M| = 2$), the demand between $t2$ and $t3$ can be routed over the network that can transfer message type $m2$ only.

On the other hand, the supported bandwidth is modeled by the so-called *capacities* to each message type $m \in M$ associated with a link $l \in E_a$ in the architecture graph g_a.

Definition 4 (capacity). With each pair $(l_i, m_j) \in E_a \times M$, associate a real value $c_{i,j} \in \mathbb{R}_0^+$ (possibly 0, if the message type cannot be routed over l_i) indicating the *capacity* on a link l_i for message type m_j.

In the following, we assume that for each link $l_i \in E_a$ exactly one capacity c_i is greater than 0.

Example 3. Figure 2 shows a ReCoNet consisting of four nodes and four links. While $\{n1, n3\}$ and $\{n3, n4\}$ can transfer the message type $m1$, $\{n2, n3\}$ and $\{n2, n4\}$ can handle message type $m2$. As the data dependency $(t1, t3)$ is bound to path $(\{n1, n3\}, \{n3, n2\})$, node $n3$ acts as a gateway. The gateway converts a message of type $m1$ to a message of type $m2$. Note that only capacities with $c > 0$ and demands with $d < \infty$ are shown in this figure. In our model, we assign exactly one capacity with $c > 0$ to each communication link $l \in E_a$ in the architecture graph g_a and at least one demand with $d < \infty$ to the data dependencies $e \in E_t$ in the task graph g_t.

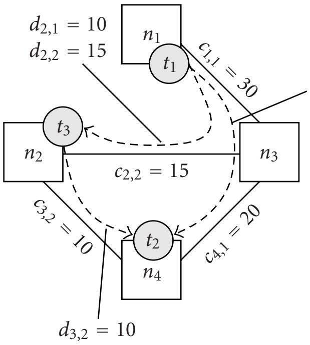


FIGURE 2: Demands are associated with pairs of data dependencies and message types while capacities are associated with pairs of links and message types.

Depending on the type of capacity, a demand of the corresponding type can be routed over such an architecture graph link. With this model refinement of a ReCoNet, it is possible to limit the routing possibilities, and moreover, to assign different demands to one problem graph edge.

Beside the communication, tasks have certain properties which are of most importance in embedded systems. These can be either soft or hard, either periodic or sporadic, have different arrival times, different workloads, and other constraints, see, for example, [26]. For *online partitioning* a precise definition of the workload is required which is known to be a complex topic. As we are facing dynamically and partially reconfigurable architectures, we have to consider two types of workload, *hardware workload* and *software workload*, which are defined as follows.

Definition 5 (software workload). The *software workload* $w^S(t, n)$ on node n produced by task t implemented in software is the fraction of execution time to its period.

This definition can be used for independent periodic and preemptable tasks. Buttazzo [26] proposed a load definition where the load is determined dynamically during runtime. The treatment of such definitions in our algorithm is a matter of future work.

Definition 6 (hardware workload). The *hardware workload* $w^H(t, n)$ on node n produced by task t is defined as a fraction of the required area and maximal available area, respectively, configurable logic elements in case of FPGA implementations.

As a task t bound to node n, that is, $(t, n) \in \beta_t$, can be implemented partially in hardware and partially in software, different implementations might exist.

Definition 7 (workload). The *workload* $w_i(t, n)$ on node n produced by the ith implementation of task t is a pair $w_i(t, n) = (w_i^H(t, n), w_i^S(t, n))$, where $w_i^H(t, n)$ ($w_i^S(t, n)$) denotes the hardware workload (software workload) on node n produced by the ith implementation of task t.

The overall hardware/software workload on a node n in the network is the sum of all workloads of the t_ith implementation of tasks bound to this node, that is, $w(n) = \sum_{(t,n)\in\beta_t} w_{t_i}(t, n)$. Here, we assume constant workload demands, that is, for all $t \in T, w_i(t, n) = w_i(t)$.

With these definitions we can define the task of online partitioning formally.

Definition 8 (online partitioning). The task of *online partitioning* solves the following multiobjective combinatorial optimization problem at runtime:

$$\min \begin{pmatrix} \max\left(\Delta_n(w^H(n)), \Delta_n(w^S(n))\right) \\ \left| \sum_n w^H(n) - \sum_n w^S(n) \right| \\ \sum_n w^H(n) + w^S(n) \end{pmatrix}, \tag{1}$$

such that

$$\begin{gathered} w^H(n), w^S(n) \leq 1, \\ \beta_t \text{ is a feasible task binding,} \\ \beta_c \text{ is a feasible communication binding.} \end{gathered} \tag{2}$$

The first objective describes the workload balance in the network. With this objective to be minimized, the load in the network is balanced between the nodes, where hardware and software loads are treated separately with $\Delta_n(w^H(n)) = \max_n(w^H(n)) - \min_n(w^H(n))$ and $\Delta_n(w^S(n)) = \max_n(w^S(n)) - \min_n(w^S(n))$.

The second objective balances the load between hardware and software. With this strategy, there will always be a good *load reserve* on each active node which is important for achieving fast repair times in case of unknown future node or link failures.

The third objective reduces the total load in the network. Finally, the constraints imposed on the solutions guarantee that not more than 100% workload can be assigned to a single node. The two feasibility requirements will be discussed in more detail next.

A feasible binding guarantees that communications demanded by the problem graph can be established in the allocated architecture. This is an important property in explicit modeling of communication.

Definition 9 (feasible task binding). Given a task graph g_t and an architecture graph g_a, a *feasible task binding* β_t is an assignment of tasks $t \in V_t$ to nodes $n \in V_a$ that satisfies the following requirements:

(i) each task $t \in V_t$ is assigned to exactly one node $n \in V_a$, that is, for all $t \in V_t$, $|\{(t, n) \in \beta_t \mid n \in V_a\}| = 1$;

(ii) for each data dependency $e \in (t_i, t_j) \in E_t$ with $(t_i, n_i), (t_j, n_j) \in \beta_t$ a path p from n_i to n_j exists.

This definition differs from the concepts of feasible binding presented in [27] in a way that communicating processes require a *path* in the architecture graph and not a direct link for establishing this communication. This way, we are able to consider *networked embedded systems*. However, considering multihop communication, we have to regard the capacity of connections and data demands of communication. This step will be named *communication binding* in the following.

Definition 10 (feasible communication binding). The task of *communication binding* can be expressed with the following ILP formulation. Define a binary variable with

$$x_{i,j} = \begin{cases} 1 & \text{data dependency } e_i \text{ is bound on link } l_j, \\ 0 & \text{else,} \end{cases} \tag{3}$$

and a mapping vector $\overrightarrow{m_i} = (m_{i,1}, \dots, m_{i,|V_a|})$ for each data dependency $e_i = (t_k, t_j)$ with the elements

$$m_{i,l} = \begin{cases} 1 & \text{if } (t_k, n_l) \in \beta_t, \\ -1 & \text{if } (t_j, n_l) \in \beta_t, \\ 0 & \text{else.} \end{cases} \tag{4}$$

Then, the following two kinds of constraints exist.

(i) For all $i = 1, \dots, |E_t|, C \cdot \overrightarrow{x_i} = \overrightarrow{m_i}$, with C being the incidence matrix of the architecture graph and $\overrightarrow{x_i} = (x_{i,j}, \dots, x_{i,|E_a|})^T$.
This constraint literally means that all incoming and outgoing demands of a node have to be equal. If a demand producing or consuming process is mapped onto an architecture graph node, the sum of incoming demands differs from the sum of outgoing demands.

(ii) The second constraint restricts the sum of demands $d_{i,j}$ bound onto a link l_j to be less than or equal to the edge's capacity c_j, where $d_{i,j}$ is the demand of the data dependency e_i. For all $j = 1, \dots, |E_a|, \sum_{i=1}^{|E_t|} d_{i,j} \cdot x_{i,j} \leq c_j$.

The objective of this ILP formulation is to minimize the total flow in the network: $\min(\sum_{i=1}^{|E_t|} \sum_{j=1}^{|E_a|} d_{i,j} \cdot x_{i,j})$. A solution to this ILP assigns data dependencies e in the task graph g_t to paths p in the architecture graph g_a. Such a solution is called a *feasible communication binding* β_c.

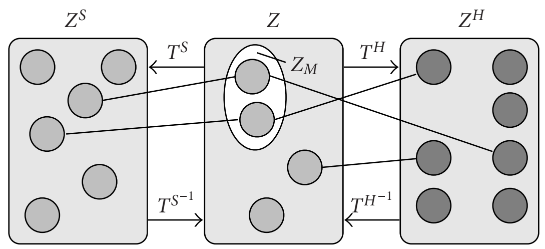


Figure 3: Hardware/software morphing is only possible in the morph states $\mathbb{Z}_M \subseteq \mathbb{Z}$. These states permit a bijective mapping of refined states ($\mathbb{Z}^S$ and $\mathbb{Z}^H$) of task t to $\mathbb{Z}$.

3.3. Task migration, task morphing, and replica binding

In order to allow online partitioning, it is mandatory to support the *migration* and the *morphing* of tasks in a ReCoNet. Note that this is only possible by using dynamically and partially reconfigurable hardware.

A possible implementation to migrate a task $t \in V_t$ bound to node $n \in V_a$ to another node $n' \in V_a$ with $n \neq n'$ is by duplicating t on node n' and removing t from n, that is, $\beta_t \leftarrow \beta_t \setminus \{(t,n)\} \cup \{(t,n')\}$. The duplication of a task t requires two steps: first, the implementation of t has to be instantiated on node n' and, second, the current context $\mathcal{C}(t)$ of t has to be copied to the new location.

In hardware/software morphing an additional step, the transformation of the context $\mathcal{C}^H(t)$ for a hardware implementation of t to an appropriate context $\mathcal{C}^S(t)$ for the software implementation of t or vice versa, is needed. As a basis for hardware/software morphing, a task $t \in V_t$ is modeled by a deterministic finite state machine m.

Definition 11. A *finite state machine* (*FSM*) m is a 6-tuple $(I, O, S, \delta, \omega, s_0)$, where I denotes the finite set of inputs, O denotes the finite set of outputs, S denotes the finite set of states, $\delta : S \times I \rightarrow S$ is the state transition function, $\omega : S \times I \rightarrow O$ is the output function, and s_0 is the initial state.

The state space of the finite state machine m is described by the set $Z \subseteq E \times O \times S$. During the software build process and the hardware design phase, state representations, $\mathbb{Z}^S$ for software and $\mathbb{Z}^H$ for hardware, are generated by transformations T^S and T^H, see Figure 3, for instance. After the refinement of $\mathbb{Z}$ in $\mathbb{Z}^S$ or $\mathbb{Z}^H$ it might be that the states $z \in \mathbb{Z}$ do not exist in $\mathbb{Z}^S$ or $\mathbb{Z}^H$. Therefore, hardware/software morphing is only possible in equivalent states existing in both, $\mathbb{Z}^H$ and $\mathbb{Z}^S$. For these states, the inverse transformation $T^{H^{-1}}$, respectively, $T^{S^{-1}}$ must exist. The states will be called *morph states* $\mathbb{Z}_M \subseteq \mathbb{Z}$ in the following (see Figure 3). Note that a morph state is part of the context $\mathcal{C}(t)$ of a task t.

In summary, both task migration and hardware/software morphing are based on the idea of context saving or *checkpointing*, respectively. In order to reduce recovery times, we create *one replica* t' for each task $t \in V_t$ in the ReCoNet. In case of task migration, the context $\mathcal{C}(t)$ of task t can be transferred to the replica t' and t' can be activated, assuming that the replica is bound to the node n' the task t should be migrated to. Thus, our ReCoNet model is extended towards a so-called *replica task graph* g_t'.

Definition 12 (replica task graph). Given a task graph $g_t = (V_t, E_t)$, the corresponding *replica task graph* $g_t' = (V_t', E_t')$ is constructed by $V_t' = V_t \cup \widetilde{V}_t$ and $E_t' = E_t \cup \widetilde{E}_t$. $\widetilde{V}_t$ denotes the set of replica tasks, that is, for all $t \in V_t$ there exists a unique $t' \in \widetilde{V}_t$ and $|V_t| = |\widetilde{V}_t|$. $\widetilde{E}_t$ denotes the set of edges representing data dependencies (t, t') resulting from sending checkpoints from a task t to its corresponding replica t', that is, $\widetilde{E}_t \subset V_t \times \widetilde{V}_t$.

The replica task graph g_t' consists of the task graph g_t, the replica tasks $\widetilde{V}_t$, and additional data dependencies $\widetilde{E}_t$ which result from sending checkpoints from tasks to their replica. With the definition of the replica task graph g_t', we have to rethink the concept of online partitioning. In particular, the definition of a *feasible task binding* β_t must be adapted.

Definition 13 (feasible (replica) task binding). Given a replica task graph g_t' and a function $r : V_t \mapsto \widetilde{V}_t$ that assigns a unique replica task $t' \in \widetilde{V}_t$ to its task $t \in V_t$. A *feasible replica task binding* is a feasible task binding β_t as defined in Definition 9 with the constraint that

$$\forall t \in V_t, \beta_t(t) \neq \beta_t(r(t)). \tag{5}$$

Hence, a task t and its corresponding replica $r(t)$ must not be bound onto the same node $n \in V_a$. In the following, we use the term *feasible task binding* in terms of *feasible replica task binding*.

3.4. Hardware checkpointing

Checkpointing mechanisms are integrated for task migration as well as morphing to save and periodically update the context of a task. In [16], checkpoints are defined to be consistent (fault-free) states of each task's data. In case of a fault or if the tasks' data are inconsistent, each task restarts its execution from the last consistent state (checkpoint). This procedure is called *rollback*. All results computed until this last checkpoint will not be lost and a distributed computation can be resumed. As mentioned above, several tasks have to go back to one checkpoint if they depend on each other. Therefore, we define checkpoint groups.

Definition 14 (checkpoint group). A checkpoint group is a set of tasks with data dependencies. Within such a group, one leader exists which controls the checkpointing.

For each checkpoint group the following premise holds: (1) each member of a checkpoint group knows the whole group, (2) the leader of a checkpoint group is not necessarily known to all the others in a group, and (3) overlapping checkpoint groups do not exist. As the developer knows the structure of the application, that is, the task graph g_t at design time, checkpoint groups can be built a priori. Thus, protocols for establishing checkpoint groups during runtime are not considered in this case.

Model of Consistency

Assume a task graph g_t with a set of tasks $V_t = \{t_0, t_1, t_2\}$ running on different nodes in a ReCoNet. The first task t_0 produces messages and sends them to the next task t_1 which performs some computation on the message's content and sends them further to task t_2. Our communication model is based on message queues for the intertask communication. Due to rerouting mechanisms, for example, in case of a link defect, it is possible that messages were sent over different links. Hence, the order of messages in general cannot be assured to stay the same.

But if messages arrive at a task t_j, we have to ensure that these were processed in the same order they have been created by task t_{j-1}. As a consequence, we assign a consecutive identifier i to every generated message. Let us assume that the last processed message by a task t_j was m_i produced at task t_{j-1}, then task t_j has to process message m_{i+1} next. If the message order arranged by task t_{j-1} has changed during communication this will be recognized at task t_j by an identifier larger than the one to be processed next. In this case all messages m_{i+k}, for all $k > 1$ will be temporarily stored in the so-called *local data set* of task t_j to be processed later in correct order.

If task t_j receives a message to store a checkpoint by the leader of a checkpoint group it will stop to process the next messages and consequently t_j will stop to produce new output messages for node t_{j+1}. In the following, all tasks of this checkpoint group will start to move incoming messages into their local data set. In addition, all tasks of the checkpoint group will store their internal states on the local data set. As a consequence, all tasks of the checkpoint group will reach a consistent state.

Hence, we define a checkpoint as follows.

Definition 15 (checkpoint). A checkpoint is a set of local data sets. It can be produced if all tasks inside a checkpoint group are in a consistent state. This is when (i) all message producing tasks are stopped and (ii) after all message queues are empty.

The checkpoint is stored in a distributed manner in the local data sets of all tasks belonging to their checkpoint group. All tasks $t \in V_t$ of the task graph g_t will have to copy their current local data set to their corresponding replica task $t' \in V_t'$ of the replica task graph g_t'. If a node hosting a task t fails, the corresponding replica task t_0 takes over the work of t and all tasks of the checkpoint group will perform a rollback by restoring their last checkpoint.

Hardware checkpointing

As we model tasks' behavior by finite state machines and we have seen how to handle input and output data to keep checkpoints consistent, we are now able to present a new model for hardware checkpointing. An FSM m that allows for saving and restoring a checkpoint can also be modeled by an FSM cm. Subsequently, we denote cm as *checkpoint FSM* or for short *CFSM*. In order to construct a corresponding CFSM cm for a given FSM m, we have to define a subset of states $S_c \subseteq S$ that will be used as a checkpoint. Using $S_c \subset S$ might be useful due to optimality reasons. First, we define a CFSM formally.

Definition 16. Given an FSM $m = (I, O, S, \delta, \omega, s_0)$ and a set of checkpoints $S_c \subseteq S$, the corresponding *checkpoint FSM* (*CFSM*) is an FSM $cm = (I', O', S', \delta', \omega', s_0')$, where

$$\begin{gathered} I' = I \times S_c \times I_{\text{save}} \times I_{\text{restore}} \quad \text{with } I_{\text{save}} = I_{\text{restore}} = \{0,1\}, \\ O' = O \times S_c, \qquad S' = S \times S_c. \end{gathered} \tag{6}$$

In the following, it is assumed that the current state is given by $(s, s') \in S'$. The current input is denoted by i'. The state transition function $\delta' : S' \times I' \rightarrow S'$ is given as

$$\delta' = \begin{cases} (\delta(s,i), s') & \text{if } i' = (i, -, 0, 0), \\ (\delta(s,i), s) & \text{if } i' = (i, -, 1, 0) \wedge s \in S_c, \\ (\delta(s,i), s') & \text{if } i' = (i, -, 1, 0) \wedge s \notin S_c, \\ (\delta(i_c,i), s') & \text{if } i' = (i, i_c, 0, 1), \\ (\delta(i_c,i), s) & \text{if } i' = (i, i_c, 1, 1) \wedge s \in S_c, \\ (\delta(s,i), s') & \text{if } i' = (i, i_c, 1, 1) \wedge s \notin S_c. \end{cases} \tag{7}$$

The output function ω' is defined as

$$\omega' = \begin{cases} (\omega(s,i), s') & \text{if } i' = (i, -, -, 0), \\ (\omega(i_c,i), s') & \text{if } i' = (i, i_c, -, 1). \end{cases} \tag{8}$$

Finally, $s_0' = (s_0, s_0)$.

Hence, a CFSM cm can be derived from a given FSM m and the set of checkpoints S_c. The new input to cm is the original input i and additionally an optional checkpoint to be restored as well as two control signals i_{save} and i_{restore}. These additional signals are used in the state transition function δ'. In case of $i_{\text{save}} = i_{\text{restore}} = 0$, cm acts like m. On the other hand, we can restore a checkpoint $s_c \in S_c$ if $i_{\text{restore}} = 1$ and using s_c as additional input, that is, $i_c = s_c$. In this case, i_c is treated as current state, and the next state is determined by $\delta(i_c, i)$. It is also possible to save a checkpoint by setting $i_{\text{save}} = 1$. In this case, the current state s is set to the latest saved checkpoint. Therefore, the state space of cm is given by the current state and the latest saved checkpoint ($S \times S_c$). Note that it is possible to swap two checkpoints by setting $i_{\text{save}} = i_{\text{restore}} = 1$. The output function is extended to output also the latest stored checkpoint s. The output function is given by the original output function ω and s' as long as no checkpoint should be restored. In case of a restore ($i_{\text{restore}} = 1$), the output depends on the restored checkpoint i_c and the input i. The initial state s_0' of cm is the initial state s_0 of m where s_0 is used as latest saved checkpoint, that is, $s_0' = (s_0, s_0)$.

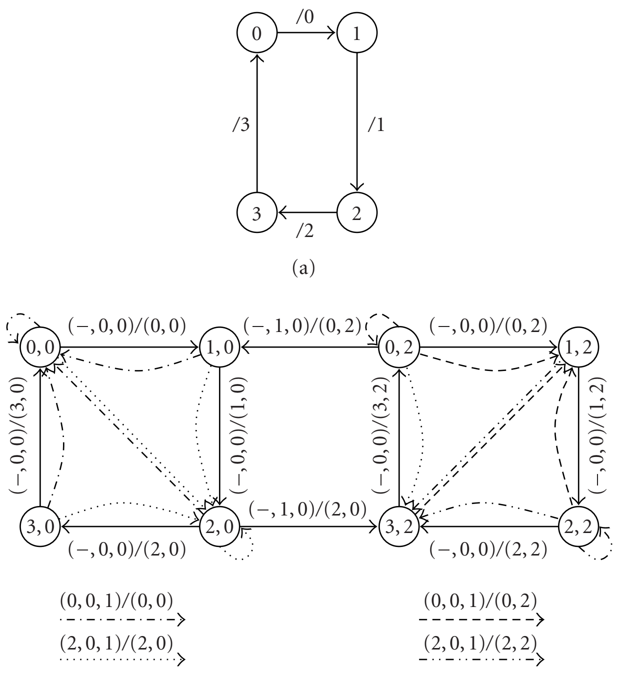


Figure 4: (a) FSM of a modulo-4-counter. (b) Corresponding CFSM for $S_c = \{0, 2\}$, that is, only in states 0 and 2 saving of the checkpoint is permitted. The state space is given by the actual state and the latest saved checkpoint.

Example 4. Figure 4(a) shows a modulo-4-counter. Its FSM m is given by $I = \emptyset, O = S = \{0, 1, 2, 3\}$, $\delta(s) = (s + 1)\%4$, $\omega(s) = s$, and $s_0 = 0$. The corresponding CFSM cm for $S_c = \{0, 2\}$ is shown in Figure 4. For readability reasons, we have omitted the swap state transitions. The state space has been doubled due to the two possible checkpoints. To be precise, there are two copies of m, one representing $s' = 0$ to be the latest stored checkpoint and one representing $s' = 2$ being the latest stored checkpoint. We can see that there exist two state transitions connecting these copied FSMs when saving a checkpoint, that is, $((2, 0), (3, 2))$ and $((0, 2), (1, 0))$. Of course it is possible to save the checkpoints in the states $(0, 0)$ and $(2, 2)$ as well. But the resulting state transitions do not differ from the normal mode transitions. The restoring of a checkpoint results in additional state transitions.

4. ARCHITECTURE AND OPERATING SYSTEM INFRASTRUCTURE

All previously mentioned mechanisms for establishing a fault-tolerant and self-adaptive reconfigurable network have to be integrated in an OS infrastructure which is shown in Figure 5. While the reconfigurable network forms the physical layer consisting of reconfigurable nodes and communication links, the top layer represents the application that will be dynamically bound on the physical layer. 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. The service provided by task migration mechanisms are required for moving tasks from one node to another while the hardware/software morphing allows for 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. Basic network services for addressing nodes, detecting link failures, and sending/receiving messages are discussed in [28]. In connection with the local operating system the hardware reconfiguration management has to be considered. Recent publications [3, 29, 30] have presented algorithms for placing hardware functionality on a reconfigurable device.

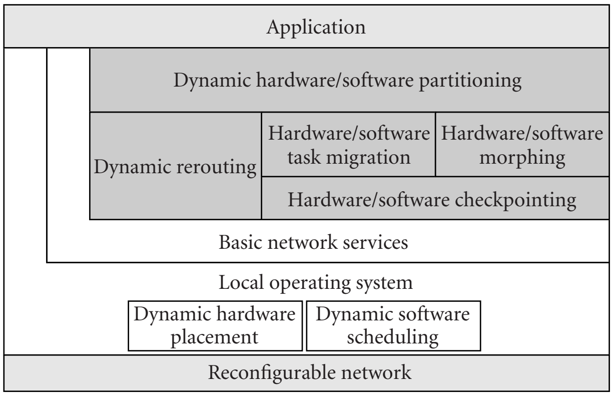

Application
Dynamic hardware/software partitioning
Dynamic rerouting
Hardware/software task migration
Hardware/software morphing
Hardware/software checkpointing
Basic network services
Local operating system
Dynamic hardware placement
Dynamic software scheduling
Reconfigurable network

Figure 5: Layers of a fault-tolerant and self-adaptive network. In order to abstract from the hardware, a local operating system runs on each node. On top of this local OS, basic network tasks are defined and used by the application to establish the fault-tolerant and self-adaptive reconfigurable network.

4.1. *Online partitioning*

The binding of tasks to nodes is determined by a so-called online hardware/software partitioning algorithm which has to (1) run in a distributed manner for fault-tolerance reasons, (2) work with local information, and (3) improve the binding concerning objectives presented in the following. In order to determine a binding of processes to resources, we will introduce a two-step approach as shown in Figure 6. The first step performs a fast repair that reestablishes the functionality and the second step tries to optimize the binding of tasks to nodes such that the system can react upon a changed resource allocation and newly arriving tasks.

Fast repair

Two of the three scenarios presented in Figure 1 will be treated during this phase. In case of a newly arriving task the decision of task binding is very easy. Here, we use discrete

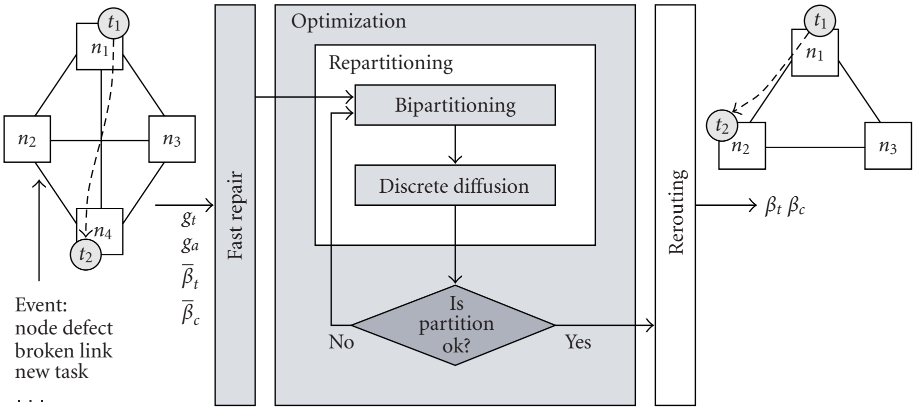


FIGURE 6: Phases of the two-step approach: while the *fast repair* step reestablishes functionality under timing constraints, the optimization phase aims on increasing fault tolerance.

diffusion techniques that will be explained later. Due to the behavior of these techniques, the load of all nodes is almost equally balanced. Hence, the new task can be bound on an arbitrary node.

In the third scenario a node defect occurs. So, tasks bound onto this node will be lost and replicas will take over the functionality. A replicated task t' will be hosted on a different node than its main task $t \in V_t$. Periodically, a replicated task receives a checkpoint by the main task and checks whether the main task is alive. If the main task is lost, the replicated task becomes a main task, restores the last checkpoint, and creates a replica on one node in its neighborhood. The main task checks either if its replicated task is still alive. If this is not the case, a replica will be created in the neighborhood again.

Bipartitioning

The applied heuristic for local bipartitioning first determines the load ratio between a hardware and a software implementation for each task $t_i \in V_t$, that is, $w^H(t_i)/w^S(t_i)$. According to this ratio, the algorithm selects one task and implements it either in hardware or software. If the hardware load is less than the software load, the algorithm selects a task which will be implemented in hardware, and the other way round. Due to the competing objectives that (a) the load on each node's hardware and software resources should be balanced and (b) the total load should be minimized, it is possible that tasks are assigned, for example, to software although they would be better assigned to hardware resources. These tasks which are suboptimally assigned to a resource on one node will be migrated to another node at first during the diffusion phase.

Discrete diffusion

While bipartitioning assigns tasks to either hardware or software resources on one node, a decentralized discrete diffusion algorithm migrates tasks between nodes, that is, changing the task binding β_t. Characteristic to the class of diffusion algorithms, first introduced by Cybenko [31] is that iteratively each node is allowed to move any size of load to each of its neighbors. The quality of such an algorithm is measured in terms of the number of iterations that are required in order to achieve a balanced state and in terms of amount of load moved over the edges of the graph.

Definition 17 (local iterative diffusion algorithm). A local iterative load balancing algorithm performs iterations on the nodes of g_a determining load exchanges between adjacent nodes. On each node $n_i \in V_a$, the following iteration is performed:

$$\begin{aligned} y_c^{k-1} &= \alpha\left(w_i^{k-1} - w_j^{k-1}\right) \quad \forall c = \{n_i, n_j\} \in E_a, \\ x_c^k &= x_c^{k-1} + y_c^{k-1} \quad \forall c = \{n_i, n_j\} \in E_a, \\ w_i^k &= w_i^{k-1} - \sum_{c=\{n_i,n_j\}\in E_a} y_c^{k-1}. \end{aligned} \tag{9}$$

In (9), w_i denotes the total load on node n_i, y is the load to be transferred on a channel c, and x is the total transferred load during the optimization phase. Finally, k denotes the integer iteration index.

In order to apply this diffusion algorithm in applications where we cannot migrate a real-valued part of a task from one node to another, an extension is introduced. With this extension, we have to overcome two problems.

(1) First of all, it is advisable not to split one process and distribute it to multiple nodes.

(2) Since the diffusion algorithm is an alternating iterative balancing scheme, it could occur that negative loads are assigned to computational nodes.

In our approach [32], we first determine the real-valued continuous flow on all edges to the neighboring nodes. Then, the node tries to fulfill this real-valued continuous flow for each incident edge, by sending or receiving tasks, respectively. By applying this strategy, we have shown theoretically and by experiment [32, 33] that the discrete diffusion algorithm

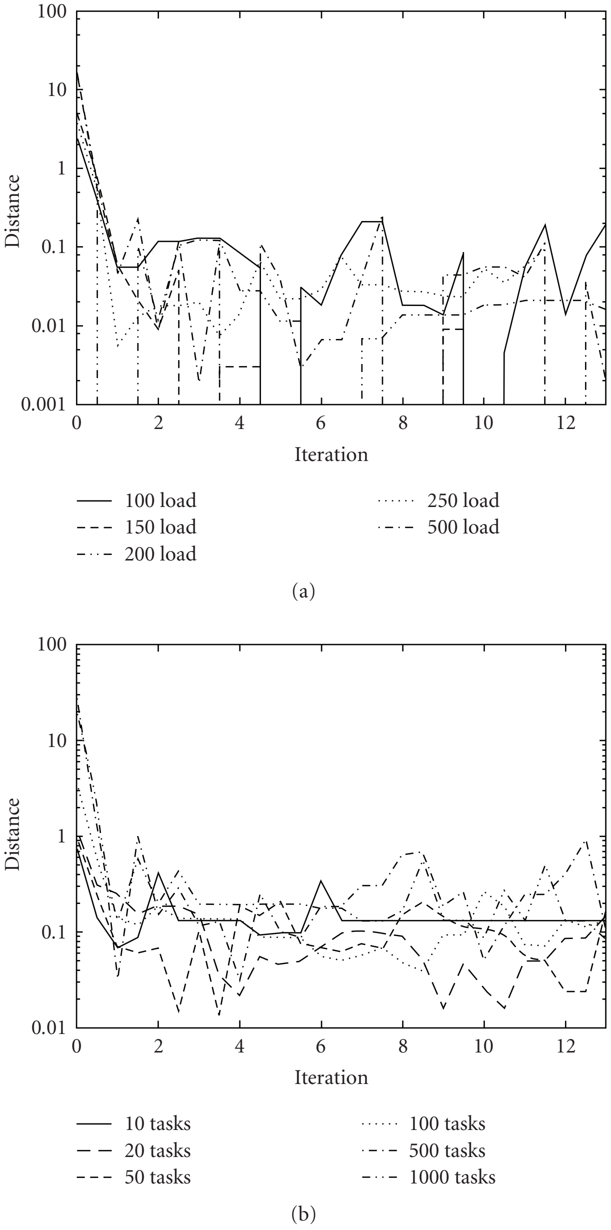


Figure 7: Presented is the distance between the solutions of our distributed online hardware/software partitioning approach and an algorithm with global knowledge. In (a) tasks are bound to network nodes such that each node has a certain load. In (b) a certain number of tasks is bound to each node and each task is implemented in the optimal implementation style.

converges within provable error bounds and as fast as its continuous counterpart.

In Figure 7 the experimental results are shown. There, our distributed approach has been evaluated by comparing it with a centralized methodology that possesses global knowledge. The centralized methodology is based on evolutionary algorithms and determines a set R of reference solutions and calculates the shortest normalized distance $d(s)$ from the solution s found by the online algorithm to any reference solution $r \in R$:

$$d(s) = \min_{r \in R} \left\{ \left| \frac{s_1 - r_1}{r_1^{\max}} \right| + \left| \frac{s_2 - r_2}{r_2^{\max}} \right| \right\}. \tag{10}$$

In the first experiment, we are starting from a network which is in an optimal state such that all tasks are implemented optimally according to all objectives. Now, we assume that new software tasks arrive on one node. Starting from this state, Figure 7(a) shows how the algorithm performs for different load values. In the second experiment, the initial binding of tasks and load sizes were determined randomly. For this case, which is comparable to an initialization phase of a network, we generated process sets with 10 to 1000 processes, see Figure 7(b). In this figure, we can clearly see that the algorithm improves the distribution of tasks already with the first iteration leading to the best improvement. We can see in Figure 7 that the failure of one node causes a high normalized error. Interestingly, the algorithm finds global optima but due to local information our online algorithm cannot decide when it finds a global optimum.

4.2. Hardware/software task migration

In case of software migration, two approaches can be considered. (1) Each node in the network contains all software binaries, but executes only the assigned tasks, or (2) the binaries are transferred over the network. Note that the second alternative requires that binaries are relocatable in the memory and only relative branches are allowed. With these constraints, an operating system infrastructure can be kept tiny. Besides software functionality, it is desired to migrate functionality implemented in hardware between nodes in the reconfigurable network. Similar to the two approaches for software migration, two concepts for hardware migration exist. (1) Each node in the network contains all hardware modules preloaded on the reconfigurable device, or (2) FPGAs supporting partial runtime reconfiguration are required. Comparable to location-independent software binaries, we demand that the configuration data is relocatable, too. In [34], this has been shown for Xilinx Virtex E devices and in [35], respectively, for Virtex 2 devices. Both approaches modify the address information inside the configuration data according to the desired resource location.

4.3. Hardware/software morphing

Hardware/software morphing is required to dynamically assign tasks either to hardware or software resources on a node. Naturally, not all tasks can be morphed from hardware to software or vice versa, for example, tasks which drive or read I/O-pins. But those tasks that are migratable need to fulfill some restrictions as presented in Section 3.3.

Basically, the morph process consists of three steps. At first, the state of a task has to be saved by taking a checkpoint in a morph state. Then, the state encoding has to be transformed such that the task can start in the transformed state with its new implementation style in the last step.

A requirement to morphable tasks is that they have to be equivalent such that the surrounding system does not recognize the implementation style of the morphable task. Also the transformation depends heavily on the implementation which especially leads to problems when transforming data types. While it is possible to represent numbers in hardware with almost arbitrary word width, current processors perform computations on 16 bit or 32 bit wide words. Thus, the numbers have to be extended or truncated. This modification causes again difficulties if numbers are presented in different representations. The representation which can either be one's complements, two's complement, fixed point, or floating point numbers needs to be transformed, too. Additional complexity arises if functionality requires a sequential computation in software and a parallel computation in hardware. Due to these implementation-dependent constraints, we currently support an automated morph-function generation only for bit vectors in the hardware that are interpreted as integers in the software. The designer needs to give information about the possible morph states and together with the help of the automated insertion of checkpoints into hardware/software tasks, the morphing becomes possible.

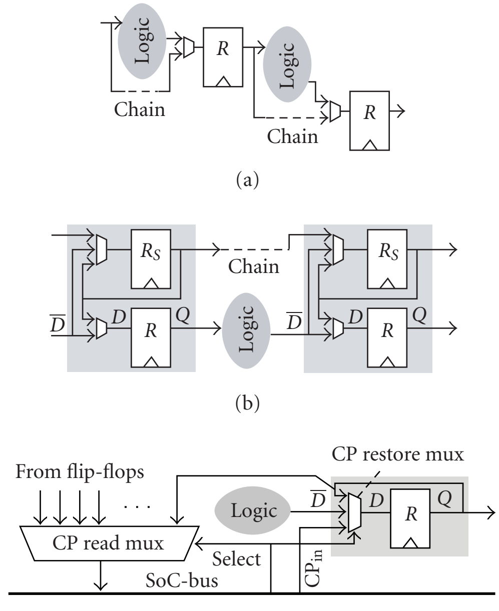
4.4. Hardware checkpointing

In Section 3, we have shown how to model checkpoints for tasks modeled by FSMs. Here, we are introducing and analyzing the overhead of three possibilities for extracting the state of a hardware module.

(i) *Scan chain*. As shown in Figure 8(a), an extra scan multiplexer in front of each flip-flop in the circuit switches between a *regular execution mode* and a *scan mode*. In the latter one, the registers are linked together to form a shift register chain. If the output of the register chain is connected to the input forming a ring shift, the module can continue regular execution immediately after the checkpoint has been read. In the case of a rollback, the last error-free state is shifted into the module.

(ii) *Scan chain with shadow registers*. Each flip-flop of the original circuit is duplicated and connected to a chain, see Figure 8(b). The multiplexer in front of the main flip flop can either propagate the value of the combinatorial circuit or the value of the corresponding shadow register. Hence, it is possible to store, restore, or swap a checkpoint within one single clock cycle.

(iii) *Memory mapping*. As shown in Figure 8(c), each flip-flop is directly accessible by the CPU via an address and a data bus. Depending on the data bus width several flip-flops can be grouped together to one word.



(a)

(b)

(c)

Figure 8: Hardware checkpointing methodologies: (a) the flip-flops are connected to form a scan chain, (b) each flip flop is replicated with a so-called shadow register which are connected to a scan chain again, (c) a set of flip-flops can be directly accessed via an address and a data port of a CPU.

All three state extraction architectures can be used for automatically modifying a given RTL design. But due to the optimization during the synthesis process from an RTL to a netlist description, it is advantageous to integrate the hardware checkpointing techniques on netlist level. Starting from the netlist, we can directly identify flip-flops by the instantiated primitives. These primitives are replaced with primitives for dedicated extended flip-flops that support saving and restoring a checkpoint, see Figure 9. Finally, the connections between the replaced flip-flops and the interface have to be determined and integrated.

In our experiments, we used the Synopsis design compiler to generate an EDIF netlist consisting of GTECH primitives. The identified GTECH flip-flops are replaced by our extended flip-flops allowing for hardware checkpointing. For our approach, we evaluated the different state extraction mechanisms discussed above according to the following properties.

(i) *Checkpoint hardware overhead*. The *checkpoint hardware overhead H* specifies the amount of additional resources required by a certain checkpointing mechanism. Here, we distinguish H_L and H_F being the checkpoint hardware overhead in terms of look up tables and flip-flops, respectively.

(ii) *Checkpoint performance reduction*. The *checkpoint performance reduction R* specifies the reduction of the maximal achievable clock frequency. As additional logic needs to be included into the original control and data paths, routing distances will slightly increase leading to a reduced clock frequency of the design.

(iii) *Checkpoint overhead*. The *checkpoint overhead C* specifies the amount of time a module is interrupted when storing a checkpoint which leads to an increase in the execution time.

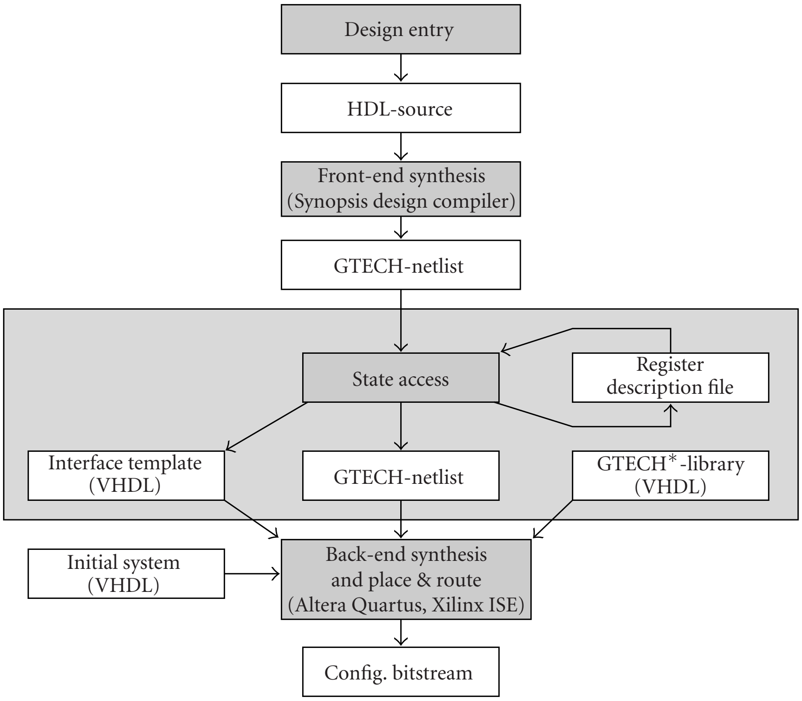

Design entry
HDL-source
Front-end synthesis (Synopsis design compiler)
GTECH-netlist
State access
Register description file
Interface template (VHDL)
GTECH-netlist
GTECH*-library (VHDL)
Initial system (VHDL)
Back-end synthesis and place & route (Altera Quartus, Xilinx ISE)
Config. bitstream

Figure 9: Design flow for integrating hardware checkpoints. Starting from the netlist, the StateAccess tool replaces flip-flops in the design by extended flip-flops that support saving and restoring of checkpoints.

(iv) *Checkpoint latency*. The *checkpoint latency L* specifies the amount of time required until the complete checkpoint data has arrived at the node hosting the specific replica task.

Table 1 presents measured values of a DES cryptographic hardware module from [36] that was automatically modified for checkpointing and tested on an Altera NIOS2 system. The table points out that each state extraction strategy is optimal in the sense of one of the defined properties. The shadow scan chain method leads in the case of high checkpoint rates to a higher throughput by the cost of almost doubling of the required logic resources. The simple scan chain approach demonstrates that it is possible to enhance a hardware module to be capable of checkpointing with an overhead of about 20% as compared to the original module.

Table 1: Results obtained by our approach by implementing the different state extraction mechanisms: scan chain, scan chain with shadow registers, and memory mapping.

	#LUTs/H_L	#Flip-Flops/H_F	
Original DES	2015/100%	984/100%	
Scan chain	2414/120%	1138/116%	
Shadow chain	3937/195%	2023/205%	
Memory mapped	2851/141%	1026/105%	
	F_{max} [MHz] / P	C	L
Original DES	116/100%	—	—
Scan chain	110/95%	10354	24979
Shadow chain	99/85%	0	16813
Memory mapped	107/92%	1306	16931

5. IMPLEMENTATION AND APPLICATION

The previously described methods have been implemented on the basis of a network consisting of four FPGA-based boards with a CPU and configurable logic resources. As an example, we implemented a driver assistant system that warns the driver in case of an unintended lane change and is implemented in a distributed manner in the network. As shown in Figure 10, a camera is connected to node *n*4. The camera's video stream is then processed in basically three steps: (a) preprocessing, (b) segmentation, and (c) lane detection. Each step is implemented as one task. The result of the lane detection is evaluated in a control task that gets in addition the present state of the drop arm switch. If the driver changes the lane without switching on the correct turn signal, an acoustic signal will warn the driver of an unintended lane change.

5.1. *Architecture and local OS*

As depicted in Figure 10, our prototype implementation of a ReCoNet consists of four fully connected FPGA boards. Each node is configured with a NIOS-II softcore CPU [37] running MicroC/OS-II [38] as a local operating system. The

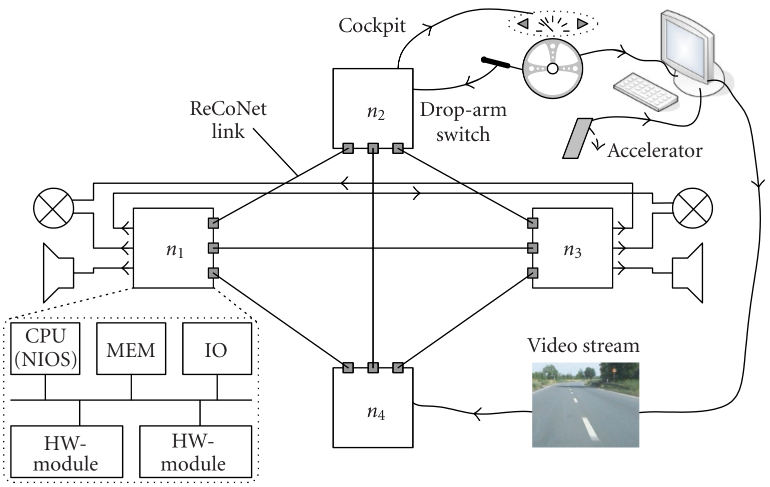


FIGURE 10: Schematic composition of a ReCoNet demonstrator: on the basis of four connected FPGA boards, we implemented a distributed operating system infrastructure which executes a lane detection algorithm. This application warns the driver acoustically in case of an unintended lane change.

local OS supports multitasking through preemptive scheduling and has been extended to a message passing system. On top of this extended MicroC/OS-II, we implemented the different layers as depicted in Figure 5. In detail, these are functions for checkpointing, task migration, task morphing and online hardware/software partitioning.

As Altera FPGAs do not support dynamic partial hardware reconfiguration, we configured each node with a set of hardware modules. This allows us to emulate the dynamic reconfiguration processes by selectively enabling hardware modules.

Although the MicroC/OS-II has no runtime system that permits dynamic task creation, we enabled the software task migration by transferring binaries to other nodes and linking OS functions to the same address such that tasks can access these functions on each node. This methodology reduces the amount of transferred binary data drastically compared to the alternative that the OS functions are transferred either. Also, this methodology avoids implementing a complex runtime system and the operating system keeps tiny.

5.2. *Communication*

For fault-tolerance reasons, the ReCoNet is based on a point-to-point (P2P) communication protocol [28]. As compared to a bus, we will produce some overhead by the routing on the one side while omitting the problem of bus arbitration.

The routing allows us to deal with link failures by changing the routing tables in such a way that data can be sent via alternative paths. Besides the fault tolerance, P2P networks have the advantage of an extremely high total bandwidth. In the present implementation, we set the physical data transfer rate of a single link to 12.5 Mbps and measured a maximum throughput of 700 kbps allowing even to transfer the video stream in our driver assistant application.

Each node stores a so-called *task resolution table* that allows a mapping from the task layer to the network layer, where the communication is performed with respect to the given node addresses. The task resolution is the key function for the task-2-task communication, allowing tasks to communicate among themselves regardless of there present hosting node. In the case of links, we have to distinguish between intermediate and long term failures. A single bit flip, for example, is an intermediate failure that will not demand additional care with respect to the routing, while a link down should be recognized as fast as possible in order to determine a new routes. As the link state is recognized in the transceiver ports of our implementation, we chose the advantageous variant to perform the line detection in hardware.

6. CONCLUSIONS

In this article, we presented concepts of self-adaptive networked embedded systems called ReCoNets. The particularities and novelties of such self-balancing and self-healing architectures stem from three central algorithmic innovations that have been proposed here for the first time and verified on a real platform for real applications, namely;

(i) fully decentralized online partitioning algorithms for hardware and software tasks;
(ii) techniques and overhead analysis for migration of hardware and software tasks between nodes in a network; and finally
(iii) morphing of the implementation style of a task from hardware to software and vice versa.

Although some of these techniques rely on existing principles of fault tolerance such as checkpoint mechanisms, we believe that their extension and the combination of the above three mechanisms is an important step towards self-adaptive and organic computing networks.

ACKNOWLEDGMENT

This work was supported in part by the German Science Foundation (DFG) under project Te/163-ReCoNets.

REFERENCES

[1] H. Walder and M. Platzner, "Online scheduling for block-partitioned reconfigurable devices," in *Proceedings of Design, Automation and Test in Europe (DATE '03)*, pp. 290–295, Munich, Germany, March 2003.

[2] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich, "Task scheduling for heterogeneous reconfigurable computers," in *Proceedings of the 17th Symposium on Integrated Cicuits and Systems Design (SBCCI '04)*, pp. 22–27, Pernambuco, Brazil, September 2004.

[3] A. Ahmadinia, C. Bobda, and J. Teich, "On-line placement for dynamically reconfigurable devices," *International Journal of Embedded Systems*, vol. 1, no. 3/4, pp. 165–178, 2006.

[4] R. Lysecky and F. Vahid, "A configurable logic architecture for dynamic hardware/software partitioning," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, pp. 480–485, Paris, France, February 2004.

[5] V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—a self-reconfigurable data processing architecture," in *Proceedings of 1st International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '01)*, Las Vegas, Nev, USA, June 2001.

[6] Chameleon Systems, *CS2000 Reconfigurable Communications Processor, Family Product Brief*, 2000.

[7] A. Thomas and J. Becker, "Aufbau- und Strukturkonzepte einer adaptive multigranularen rekonfigurierbaren Hardwarearchitektur," in *Proceedings of Organic and Pervasive Computing, Workshops (ARCS '04)*, pp. 165–174, Augsburg, Germany, March 2004.

[8] C. Bobda, D. Koch, M. Majer, A. Ahmadinia, and J. Teich, "A dynamic NoC approach for communication in reconfigurable devices," in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL '04)*, pp. 1032–1036, Antwerp, Belgium, August-September 2004.

[9] Altera, "FLEX 10K Devices," November 2005, http://www.altera.com/products/devices/flex10k/f10-index.html.

[10] P. Zipf, *A fault tolerance technique for field- programmable logic arrays*, Ph.D. thesis, Siegen University, Siegen, Germany, November 2002.

[11] A. Doumar and H. Ito, "Detecting, diagnosing, and tolerating faults in SRAM-based field programmable gate arrays: a survey," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 11, no. 3, pp. 386–405, 2003.

[12] CERN, "FPGA Dynamic Reconfiguration in ALICE and beyond," November 2005, http://alicedcs.web.cern.ch/alicedcs/.

[13] W. Xu, R. Ramanarayanan, and R. Tessier, "Adaptive fault recovery for networked reconfigurable systems," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, p. 143, IEEE Computer Society, Los Alamitos, Calif, USA, April 2003.

[14] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Efficiently supporting fault-tolerance in FPGAs," in *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 105–115, ACM Press, Monterey, Calif, USA, February 1998.

[15] W.-J. Huang and E. J. McCluskey, "Column-based precompiled configuration techniques for FPGA," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 137–146, IEEE Computer Society, Rohnert Park, Calif, USA, April-May 2001.

[16] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[17] K. M. Chandy and L. M. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[18] N. H. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 942–947, 1997.

[19] S. Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, Kluwer Academic, Boston, Mass, USA, 1996.

[20] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proceedings of 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, pp. 22–29, IEEE Computer Society, Napa Valley, Calif, USA, April 1997.

[21] S. M. Scalera and J. R. Vázquez, "The design and implementation of a context switching FPGA," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, p. 78, IEEE Computer Society, Napa, Calif, USA, April 1998.

[22] K. Puttegowda, D. I. Lehn, J. H. Park, P. Athanas, and M. Jones, "Context switching in a run-time reconfigurable system," *Journal of Supercomputing*, vol. 26, no. 3, pp. 239–257, 2003.

[23] G. Brebner, "The swappable logic unit: a paradigm for virtual hardware," in *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds., pp. 77–86, IEEE Computer Press, Napa Valley, Calif, USA, April 1997.

[24] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA coprocessors," in *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL '00)*, pp. 121–130, Villach, Austria, August 2000.

[25] H. Simmler, "Preemptive Multitasking auf FPGA Prozessoren," Dissertation, University of Mannheim, Mannheim, Germany, 2001, page 279.

[26] G. C. Buttazzo, *Hard Real-Time Computing Systems*, Kluwer Academic, Boston, Mass, USA, 2002.

[27] T. Blickle, J. Teich, and L. Thiele, "System-level synthesis using evolutionary algorithms," in *Design Automation for Embedded Systems*, R. Gupta, Ed., vol. 3, pp. 23–62, Kluwer Academic, Boston, Mass, USA, January 1998.

[28] D. Koch, T. Streichert, S. Dittrich, C. Strengert, C. D. Haubelt, and J. Teich, "An operating system infrastructure for fault-tolerant reconfigurable networks," in *Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS '06)*, pp. 202–216, Frankfurt/Main, Germany, March 2006.

[29] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[30] H. Walder and M. Platzner, "Fast online task placement on FPGAs: free space partitioning and 2D-hashing," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03) / Reconfigurable Architectures Workshop (RAW '03)*, p. 178, Nice, France, April 2003.

[31] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279–301, 1989.

[32] T. Streichert, C. D. Haubelt, and J. Teich, "Distributed HW/SW-partitioning for embedded reconfigurable systems," in *Proceedings of Design, Automation and Test in Europe Conference and Exposition (DATE '05)*, pp. 894–895, Munich, Germany, March 2005.

[33] T. Streichert, C. D. Haubelt, and J. Teich, "Online hardware/software partitioning in networked embedded systems," in *Proceedings of Asia South Pacific Design Automation Conference (ASP-DAC '05)*, pp. 982–985, Shanghai, China, January 2005.

[34] E. L. Horta, J. W. Lockwood, and S. T. Kofuji, "Using parbit to implement partial run-time reconfigurable systems," in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications (FPL '02)*, pp. 182–191, Springer, Montpellier, France, September 2002.

[35] H. Kalte, G. Lee, M. Porrmann, and U. Rückert, "REPLICA: a bitstream manipulation filter for module relocation in partial reconfigurable systems," in *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium—Reconfigurable Architectures Workshop*, p. 151, Denver, Colo, USA, April 2005.

[36] OpenCores, 2005, http://www.opencores.org.

[37] Altera, "Nios II Processor Reference Handbook," July 2005.

[38] J. Labrosse, *Micro-C/OS-II*, CMP Books, Gilroy, Calif, USA, 2nd edition, 2002.

Thilo Streichert received the Diploma degree in electrical engineering and computer science from the University of Hannover, Germany, in 2003. Beside his studies, he gained industrial research experience at the Multimedia Research Labs of NEC in Kawasaki (2002), Japan, and in the Semiconductor and ICs Advanced Engineering-Design Methodology Group of Bosch (2003), Germany. He is currently a Ph.D. degree candidate in the Department of Computer Science at the University or Erlangen-Nuremberg, Germany. His research interests include reconfigurable computing and networked embedded systems.

Dirk Koch received his Diploma degree in electrical engineering from the University of Paderborn, Germany, in 2002. During his studies, he worked on neural networks on coarse-grained reconfigurable architectures at Queensland University of Technology, Brisbane, Australia. In 2003, he joined the Department of Computer Science of the University of Erlangen-Nuremberg, Germany. His research interests are distributed reconfigurable embedded systems and reconfigurable hardware architectures.

Christian Haubelt received his Diploma degree in electrical engineering from the University of Paderborn, Germany, in 2001, and received his Ph.D. degree in computer science from the Friedrich-Alexander University of Erlangen-Nuremberg, Germany, in 2005. He leads the System-Level Design Automation Group in the Department of Hardware-Software Codesign at the University of Erlangen-Nuremberg. He serves as a Reviewer for several well-known international conferences and journals. His special research interests focus on system-level design, design space exploration, and multiobjective evolutionary algorithms.

Jürgen Teich received his Master's degree (Dipl. Ing.) in 1989 from the University of Kaiserslautern (with honors). From 1989 to 1993, he was a Ph.D. student at the University of Saarland, Saarbrücken, Germany, from where he received his Ph.D. degree (summa cum laude). His Ph.D. thesis entitled "A compiler for application-specific processor arrays" summarizes his work on extending techniques for mapping computation intensive algorithms onto dedicated VLSI processor arrays. In 1994, he joined the DSP Design Group of Prof. E. A. Lee and D. G. Messerschmitt in the Department of Electrical Engineering and Computer Sciences (EECS) at UC Berkeley, where he was working in the Ptolemy Project (postdoc). From 1995 to 1998, he held a position at the Institute of Computer Engineering and Communications Networks Laboratory (TIK) at ETH Zürich, Switzerland, finishing his habilitation entitled "Synthesis and optimization of digital hardware/software systems" in 1996. From 1998 to 2002, he was a Full Professor in the Electrical Engineering and Information Technology Department of the University of Paderborn, holding a Chair in computer engineering. Since 2003, he is appointed a Full Professor in the Computer Science Institute of the Friedrich-Alexander University Erlangen-Nuremberg holding the Chair of Hardware-Software Codesign. He has been a Member of multiple program committees of well-known conferences and workshops. He is a Member of the IEEE and the author of a textbook edited by Springer in 1997. His research interests are massive parallelism, embedded systems, codesign, and computer architecture. Since 2004, he also has been an elected reviewer for the German Science Foundation (DFG) for the area of computer architecture and embedded systems. He is involved in many interdisciplinary national basic research projects as well as industrial projects. He is supervising 19 Ph.D. students currently.