An Operating System Infrastructure for Fault-Tolerant Reconfigurable Networks

Dirk Koch, Thilo Streichert, Steffen Dittrich, Christian Strengert, Christian D. Haubelt, and Jürgen Teich

Department of Computer Science 12
University of Erlangen-Nuremberg, Germany

Abstract. Dynamic hardware reconfiguration is becoming a key technology in embedded system design that offers among others new potentials in dependable computing. To make system designers benefit from this new technology, powerful infrastructures and programming environments are needed. In this paper, we will propose new concepts of an operating system (OS) infrastructure for reconfigurable networks that allow to efficiently design fault-tolerant systems. For this purpose, we consider a hardware/software solution that supports *dynamic rerouting*, *hardware* and *software task migration*, *hardware/software task morphing*, and *online partitioning*. Finally. we will present an implementation of such a reconfigurable network providing this OS infrastructure.

1 Introduction

Nowadays, embedded systems are typically networked at different levels of granularity. This can be either at system level, where different controllers cooperate with each other, like in sensor networks or body area networks, or at chip level where different processor cores and dedicated hardware modules are implemented on a single die. The motivation to design such a networked embedded system as can be found in automotive and avionic industries ranges from the computational power over reliability to flexibility aspects.

In this paper, we will present the possibilities that distributed dynamic hardware reconfiguration offers in the context of fault tolerance and flexibility. In the following, we will use the term *reconfigurable network* to denote a networked embedded system consisting of dynamically hardware reconfigurable nodes (FPGAs). In such a reconfigurable network, it becomes possible to migrate hardware and software tasks from one node to another during runtime. Thus, resource faults can be compensated by *rebinding* of tasks to fully functional nodes of the network. This task of rebinding is also known as *online partitioning* [1, 2]. However, in order to allow system designers to benefit from this new technology, powerful infrastructures and programming environments are needed.

Recent research focuses on operating systems for single FPGA solutions [3–6] where hardware tasks are dynamically assigned to FPGAs. On the other hand, architectures for networked reconfigurable solutions like PACT [7], Chameleon [8], HoneyComb [9], and dynamically reconfigurable networks on chips (DyNoCs) [10] were investigated intensively. Nevertheless, the former omits the fact that more and more embedded systems become networked, whereas the latter does not account the support of basic hardware task management for *online scheduling* and *online placement*. Moreover, only by considering both aspects simultaneously, the problem of designing fault-tolerant and flexible or even self-optimizing embedded systems can be solved.

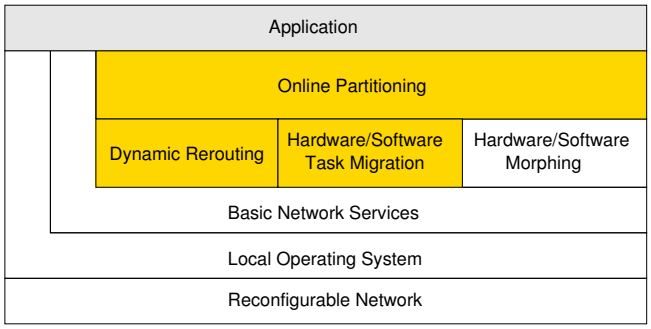

Fig. 1. Basic OS features provided by a ReCoNet. On each node in the reconfigurable network a local OS is needed. Based on these local OSs, basic network services can be implemented. On top of the basic network services, the basic OS features *dynamic rerouting*, *task migration*, and *morphing* are defined. In order to increase the fault tolerance of a ReCoNet, *online partitioning* must be supported as well. Finally, an application can be build on top of all these layers.

In this contribution, we close this gap by proposing new concepts for an operating system (OS) infrastructure for reconfigurable networks that allow for designing dependable computing systems efficiently. This operating system provides an efficient infrastructure and programming environment by providing the basic tasks known as *dynamic rerouting*, *hardware* and *software task migration*, *hardware/software task morphing*, and *online partitioning*. This is shown in Fig. 1. In this paper, we focus on dynamic rerouting, task migration, and online partitioning, Whereas hardware/software task morphing is covered only briefly. In the following, we denote a reconfigurable network which supports these basic tasks as ReCoNet.

The first three features that are provided by the OS in a ReCoNet are (1) *dynamic rerouting*, (2) *task migration*, and (3) *task morphing*. Note that these features that deal with erroneous resources have to be implemented on each reconfigurable node such that they run in a distributed manner in the network. Thus, we are able to compensate line errors by computing a new routing for broken communications and we can migrate tasks from one node in the network to another during runtime. Moreover, we are able to morph the implementation style of a task. Especially the task of fault tolerant communication in networked reconfigurable systems is of outer importance and will be discussed in this paper comprehensively. This communication protocol is the basis for an efficient rerouting algorithm for a ReCoNet. Based on the three tasks *dynamic rerouting*, *task migration*, and *task morphing*, we define feature (4) called *online partitioning* as the process of optimally binding tasks to nodes in the network during runtime. Feature (4) guarantees the fault tolerance of the ReCoNet.

In summary, the paper contributes with an OS infrastructure to design modern fault-tolerant and flexible embedded systems covering reconfigurable networks or even organic computing systems. The rest of the paper is organized as follows: In Section 2, an in depth discussion of the basic OS tasks *dynamic rebinding*, *task migration*, and *task morphing* will be done. Section 3 is devoted to the topic of *fault tolerance* and *online optimization*. Finally, Section 4 presents our implementation of a ReCoNet and the most important figures related to this implementation. The focus in this section will be on the implementation of a communication protocol that supports dynamic rerouting.

2 Basic OS Features

This section describes the basic features needed for running a ReCoNet. Before defining these features, we will take a closer look on the underlying architecture. In this paper, we consider small networked embedded systems consisting of dynamically hardware reconfigurable nodes. The main aspects of the architecture are:

- small: Each node in the network is able, but is not necessarily required, to store the current state of the entire network. The state of a network is given by its current typology consisting of all available nodes, of available links, and of the distribution of the tasks in the network.
- dynamic hardware reconfiguration: Allows the implementation of arbitrary functions in hardware. Thus, it accelerates the computation of the corresponding functions required in the network.
- embedded: requires the optimization of different objectives, like power consumption, cost, etc. simultaneously.

These are the fundamental properties of a reconfigurable network that we call a ReCoNet. Furthermore, in order to increase the degree of fault tolerance and flexibility of a ReCoNet, it must support *online partitioning* of tasks in the network. For this purpose, we have to implement four basic OS features. In order to compensate errors in the hardware infrastructure, we implemented the OS features *dynamic rerouting*, *hardware* and *software task migration*, and *hardware/software task morphing*. Network connectivity faults are compensated by the computation of a new routing and faults of a complete node are compensated by migrating tasks to other nodes. Finally, the task morphing allows for changing the implementation style of a task from hardware to software and vice versa at runtime. On top of these features, we implemented an additional, fourth OS feature named *online partitioning* which will be discussed in Section 3. This online partitioning may use task migration and task morphing due to optimality reasons.

In order to describe these OS features formally, we need an appropriate model. The application implemented by the reconfigurable network is given by n tasks $T = \{t_1, t_2, \ldots, t_n\}$ running on m possible nodes $R = \{r_1, r_2, \ldots, r_m\}$. Tasks may communicate with each other modeled by so-called *data dependencies* $D = \{d_1, d_2, \ldots, d_k\} \subseteq T \times T$. Moreover, the ReCoNet structure is given by l links $C = \{c_1, c_2, \ldots, c_l\}$ between the nodes where $C \subseteq R \times R$. Each task $t_i \in T$ can be mapped onto an arbitrary set of nodes. Moreover, a task can be implemented in either hardware (HW) or software (SW). We therefore model all possible bindings as a set M, where $M \subseteq T \times R \times \{hw, sw\}$. The actual binding of tasks to resources is called $\beta \subseteq M$. A task t_i bound to hardware $(t_i, r_j, hw) \in \beta$ produces a hardware load (number of resources occupied on the FPGA) of $w^H(t_i)$. The same task implemented in software produces a software load (CPU utilization) of $w^S(t_i)$. The OS features are triggered if a resource fault is detected. In the following, we reveal the basic OS features *dynamic rerouting*, *task migration*, and *task morphing* in detail. Online partitioning will be discussed in Section 3 in the scope of fault tolerance.

2.1 Rerouting

The first OS feature to be defined is the task of *dynamic rerouting*. Rerouting is required if a connection ($c_f \in C$) in the network fails. All data dependencies

routed over this connection has to be redirected. There are several publications dealing with this issue where recent work was mainly focused on probabilistic approaches [11].

Here, we consider a high-level fault tolerant approach. Dynamic rerouting itself can be decomposed in three subproblems:

1. Line detection: Is a link $c_i = (r_j, r_k)$ between two nodes r_j and r_k available or not?
2. Network state distribution: If a connection between two nodes (r_j, r_k) fails, all nodes using link c_f must be informed.
3. Routing of broken communications (data dependencies).

Note that communication takes place between tasks and the binding of a task to a network node can change at run-time. Therefore, the rerouting is much more complex as in static network where communication takes place between nodes. This will be discussed comprehensively in Section 4 where we present an efficient algorithm for dynamic rerouting.

2.2 Hardware and Software Task Migration

Task migration describes the rebinding of hardware and software tasks $t_i \in T$ from one node r_j in the network to another r_k. Assuming t_i implemented in hardware, i.e., $(t_i, r_j, hw) \in \beta$. After the rebinding $(t_i, r_k, hw) \in \beta$. To perform this step, we need $\{(t_i, r_j, hw), (t_i, r_k, hw)\} \in M$. Note, that if we configure the reconfigurable nodes r_i, r_k with a processor, it may be possible to morph a hardware task ($(t_i, r_j, hw) \in \beta$) to a software task ($(t_i, r_j, sw) \in \beta$) and vice versa, too. For a node r_j this will further need $\{(t_i, r_j, hw), (t_i, r_j, sw)\} \in M$.

But when should the task migration take place? In this paper, we focus on the aspect of dependability and, in particular, on fault tolerance. Thus, task migration is used to compensate resource faults, i.e., if a node r_f in the network fails, all tasks running on r_f must be migrated to other nodes. Thus, task migration can be divided into two subproblems:

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

The first subproblem can be solved by observing if a node r_f has at least one working connection to any of its neighbors ($\{r_i \mid (r_f, r_i) \in C\}$). Otherwise, this node is called *isolated*. An isolated node cannot be used for process execution any longer and all tasks bound to this node must be migrated.

An important question is how to perform a save task migration, i.e., how to keep track on the current state of a task. For this purpose, we use so-called *checkpoints* as discussed in the following section. Moreover, we must answer the question of how to optimally bind a task t_i after some resource fault. This is especially necessary in the context of embedded systems where multiple objectives have to be optimized simultaneously while meeting several constraints. This topic will also be discussed in subsequent section in the context of fault tolerance.

2.3 Hardware/Software Task Morphing

Hardware/software task morphing describes the switching of the implementation style of a task from hardware to software or vice versa. Assuming that functionality can be implemented for the available hardware and software resources, the

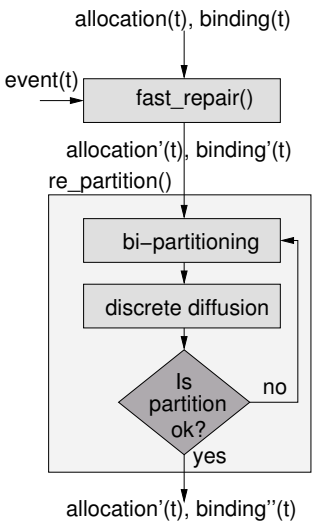

Fig. 2. Phases of the online partitioning. In case of certain events, indicating e.g., temporary or permanent resource and link faults, a fast reassignment of tasks to available resources is found by *fast_repair()*. In the second phase, a diffusion-based *dynamic repartitioning* is applied where an actual temporal bi-partition is iteratively improved by migrating tasks to other active resources and by possibly changing their implementation style.

morphing phase needs several steps, e.g., for extracting states of the functionality implemented in either hardware or software and transforming these state such that they can be loaded in their functional counterpart. Due to space limitations, we will omit an in depth discussion.

3 Fault Tolerance

With the above presented methods for *dynamic rerouting*, *task migration*, and *task morphing*, we are now able to investigate new concepts for increasing the fault tolerance in a ReCoNet. The problem we face is the following: Suppose a node r_f in the reconfigurable network fails and with this node all its functionality will be lost. All tasks $t_i \in T$ with $(t_i, r_f, \{hw, sw\}) \in \beta$ must be migrated to a fully-functional node $r \in R \backslash \{r_f\}$. Obviously, this should be done quickly in order to compensate the resource fault. However, the new task binding β' might be suboptimal or even miss some constraints imposed on the system, such that we have to perform an optimization of the system at runtime.

The described scenario is sketched in Fig. 2. The *online partitioning* basically consists of two phases: (i) a *fast repair phase* that reestablishes the functionality of a defect node and (ii) a *repartitioning phase* that optimizes the binding of tasks in the network.

In order to guarantee a *fast repair*, we propose the use of *self replication of tasks* in combination with *checkpointing*. Checkpointing is responsible for saving a task's state in order to recover this state, whereas replicating tasks assures that a fast migration decision can be made in case of some node fault. Hence, after a resource fault, the replicated tasks take over control of the computation

and restore the last state saved from this computation. In a second step, the task is replicated as well in order to guarantee a fast migration step in case of an additional node failure.

During the replication of tasks, we might step into another problem, which will be solved by so-called *dynamic repartitioning*. The new tasks that will be produced during the replication phase has to be bound to free resources. Unfortunately, it can not be guaranteed during design time, that exactly the resource for binding either a task implemented in hardware or software is free during runtime. Hence, a novel strategy for dynamic repartitioning, that decides whether a tasks will be implemented in hardware or software and on which node a task will be executed.

In the following, we will discuss the three aspects *self replication*, *checkpointing*, and *dynamic repartitioning* separately.

3.1 Self Replication

For the purpose of redundancy, a *regular task* replicates itself on another node, such that an execution of a task can be continued if one of these nodes fails. Self replication is task migration to an adjacent node without stopping/removing the regular task. This replicated and migrated task will be called *shadow task* subsequently. Thus, there exists a shadow task for each regular task in the reconfigurable network. A regular task and its corresponding shadow task observe each other by periodically sending and requesting so-called *keep alive messages* from each other. To permit this mutual monitoring a task to task communication protocol (Task2Task) must be supported by a ReCoNet (cf. Sec. 4.2). In Task2Task communication so-called *task addresses* (TAD) are resolved to physical *node addresses* (NAD) such that messages can be routed between nodes in the network.

To avoid confusions during the resolution of a TAD to a NAD, TADs must be unique in a ReCoNet. Thus, a designer has to define unique TADs for a regular task and the corresponding shadow task at design time.

If a shadow task becomes a regular task, i.e., if the former regular task is not available any longer, it takes the TAD of the former regular task. On the other hand, if the shadow task fails, a new shadow task is created by the regular task and the TAD of the shadow task will be assigned. This shadow task is bound on the first adjacent node, which has free capacities for an additional task.

A problematic scenario is the decomposition of a network in two parts and the reconnection of two subnetworks. Suppose the regular task is running in one part of the network and its shadow task is running in the other part. The regular task does not receive any keep alive messages from its shadow task and therefore creates a new one. The same holds for the shadow task. This task in turn assumes that the regular task is out of order and sets itself to be the regular task. Producing another shadow task, leads to a situation in which two TADs of the regular and two TADs of the shadow task exist in the decomposed network.

In order to resolve this particular problem when two parts of a network are reconnected, we make use of the concept of so-called *checkpoint sessions*, which will be introduced in more detail later on. Here, it is only important to understand, that after each creation of a checkpoint or each rollback to a checkpoint, a session number will be increased. Thus, the session number is representative for the lifetime of a task. Now, if two tasks have the same TAD, the tasks with the shortest lifetime survives. If they have the same lifetime, the task on the node with the lowest NAD will be removed.

With this concept, it can be ensured that always one regular and one shadow task exist, if an adjacent node provides the necessary resources for execution.

3.2 Checkpointing

Without a sufficient mechanism for saving and restoring states of processes, the migration and replication of tasks is not applicable. For this purpose, *checkpointing* mechanisms are integrated which contain a context of a task and are periodically updated. In [12] checkpoints are defined as:

Definition 1. *A* checkpoint *denotes a consistent (fault-free) state of each task's data. In case of a fault or the tasks' data are inconsistent, each task that belongs to a checkpoint starts 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 continued.*

As mentioned above, several tasks have to go back to one checkpoint if they depend on each other. Therefore, we define checkpoint groups:

Definition 2. *A checkpoint group is a set of tasks that belongs to a certain functionality. Thus, a checkpoint group is characterized by a set of TADs. Within such a group, one leader exists which controls the checkpointing.*

For each checkpoint group the following premise holds:

- Each member of a checkpoint group knows the whole group.
- The leader of a checkpoint group is not necessarily known to all the others in a group. It is sufficient, if the leader knows its special purpose.
- Overlapping checkpoint groups do not exist. If a task is required by two functionalities, these are combined to one functionality and a new checkpoint group. In general this assumption causes no problem, because tasks are usually dedicated for one function.

We assume that a developer knows the structure of functionality at design time and can build checkpoint groups a priori. Thus, protocols for establishing checkpoint groups during runtime are not considered in this case. Also, the leader of a group has to be chosen a priori, but without any restrictions.

Model of Consistency Assume a number of tasks $t_i \in T$. The first task t_0 produces messages and sends them to the next task t_1 which performs some computation on the messages content and sends them to the next node t_{i+1}. The communication model is based on message queues where every message can be sent with a different priority. Hence, the order of the messages can in general not be assured to stay the same and also the time a message arrives at a task is non deterministic.

But if a message arrives at a task, the task saves it in its *local data set* that this message has been processed. Then, a state of a checkpoint group consists of the (i) message queues' content and (ii) the local data set which save the processed messages and the rest of a task's state. In order to respect the message queues' content that cannot be restored due to the mentioned nondeterminism, a consistent checkpoint can only be produced if all message generating tasks are stopped and all message queues are empty. Hence, we extend and redefine a checkpoint as follows:

Definition 3. *A checkpoint is a set of local data sets. It can be produced if all tasks are in a consistent state and all message queues are empty after all message producing tasks are stopped.*

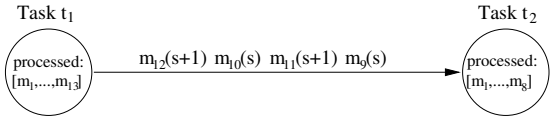

Fig. 3. Task t_1 has finished session s with checkpoint $[m_1, \ldots, m_{13}]$. Although the message order of message m_{10} and m_{11} has been changed, task t_2 is able to determine the correct checkpoint of session s.

Due to this consistency model, certain messages are assigned to certain checkpoints. In Fig. 3, two tasks are shown, that update their checkpoints after every tenth message. After each update procedure, a *checkpoint session number* is incremented. So, up to message m_{10}, all messages belong to session s and after message m_{10} has been processed, the checkpoint is updated.

Due to the nondeterminism of the message order it can occur that messages arrive belonging to the next checkpoint session. Thus, all messages have to contain the checkpoint session number they belong to. Assume that one node fails and it restores the last checkpoint with a certain session number. Then, this node has to tell its neighbors which is their corresponding checkpoint session number, such that the state of the system can be restored correctly.

The checkpointing mechanisms can be used in order to keep the shadow task updated. Every time a task generates or updates its checkpoint, it transmits this checkpoint also to the shadow task. When the shadow task takes control and becomes a regular task, it detects that its local data is inconsistent and causes a rollback. Note, that it is applicatory not possible to hold consistent local data at a shadow task. This would otherwise result in a multiplication of communicated messages in the network. Each task of a checkpoint group would have to duplicate its messages and send them to the receiving task and to its shadow task.

3.3 Dynamic Repartitioning

After a node fails in a network, new shadow tasks are started and restored, such that they are in a consistent state with their corresponding regular tasks. In [2], this phase has been called *fast repair* phase and aims a network with full functionality and the same redundancy as before the failure. This phase has to be passed in a certain time, so that we can make assumptions about the real-time behavior.

The second phase, described in this section, tries to find an optimal binding of processes to nodes under the respectation of certain objectives. In this phase, the decision whether a task will be implemented on hardware resources or software resources as well as the distribution of tasks in the network will be done iteratively until a satisfying binding β concerning the objectives is found. Note, that due to reasons of fault tolerance this approach has to run in a distributed manner in the network.

Our overall approach to *dynamic repartitioning* tries to find a binding of tasks to nodes, such that the load reserves of hardware as well as of software resources are maximized on all nodes. Optimizing the binding of tasks to resources in this way, increases the probability that a shadow task can be bound onto an adjacent node.

In [13], we proposed a two step methodology consisting of a *diffusion phase* and a local *hardware/software partitioning phase*:

Diffusion Phase: During the diffusion phase nodes exchange tasks according to their load differences on the nodes. Characteristic to a diffusion-based algorithm, introduced first by Cybenko [14], is that iteratively each node is allowed to move any size of load to each of its neighbors. Communication is only allowed along point-to-point connections. The quality of such an algorithm may be measured in terms of the number of iterations that are required in order to achieve a balanced state and in terms of the amount of load moved over the edges of the graph. In [2], we have presented an extended diffusion algorithm, that exchanges only whole tasks between nodes and thus, only discrete load entities. Anyway, it has been shown theoretically and by experiment that our proposed version of the diffusion algorithm does not exceed optimality constraints concerning the optimization flow of its continuous counterpart and moreover, we are able to show theoretically maximal deviations with respect to the quality of the load balance. Note that the diffusion phase makes use of the OS feature *hardware* and *software task migration*.

Local Bi-Partitioning: The local bi-partitioning supposes that each task $t_i \in T$ can be either implemented in hardware or software. Each implementation style causes certain costs or load on the node's resources $w^{H/S}(t_i)$ and upon these costs a ratio is determined for each task: $w^H(t_i)/w^S(t_i)$. According to this ratio the bi-partitioning algorithm selects on task and implements it either in hardware or software. Due to such a local strategy, we can guarantee that the total load will be minimized, but to reach an optimal hardware/software balance, we calculate the total software load and the total hardware load on one node. 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 these competing objectives (balanced hardware/software load and minimization of total load), tasks with a ratio larger than one can be assigned to hardware and tasks with a ratio less than one are assigned to software. Of course it is possible that tasks are assigned to a resource, such that they are implemented suboptimal. But during the diffusion phase, we diffuse these tasks at first. Therefore, we introduce two priority lists, one for software and one for hardware tasks. In these lists, we collect all tasks in the reverse order as they are assigned to a hardware or software resource. Thus, the last task which was, e.g., assigned to software is the first task which will be diffused if the node has to send tasks via the network. Therefore, suboptimal partitioned tasks will have a higher mobility, which leads to an improvement concerning the convergence speed. Note that the implementation of the local bi-partitioning needs the OS feature *hardware/software task morphing*.

4 Prototype Implementation

In this section, we will present our implementation of a ReCoNet. Here, we will focus on the *communication infrastructure* and *dynamic rerouting* algorithm.

4.1 Architecture and Local OS

Our prototype implementation of a ReCoNet consists of four fully connected Altera Cyclone FPGA boards [15]. Each Board is configured with a NIOS soft-core CPU [16] running microC/OS-II [17] as local operating system. The local OS permits, multi-tasking by priority-based preemptive scheduling. We extend the microC/OS-II, by a new C++-API for task creation and an Inter Process

Communication (IPC) infrastructure based on message passing. As Altera FPGAs do not support dynamic hardware reconfiguration, we configure each node with a set of hardware modules implementing selected tasks. These hardware modules can be activated during runtime to emulate dynamic reconfiguration. Beside the NIOS processor and the application dependent hardware modules, each Cyclone board is configured with a number of new designed communication ports. These communication ports permit line detection which is a basic functionality for *dynamic rerouting*. Moreover, a novel communication protocol was developed providing many features to support *dynamic rerouting* and *task migration*. This communication protocol will be discussed in detail next.

4.2 Communication

Our ReCoNet approach demands a specialized network infrastructure. In order to obtain a high degree of fault tolerance, we cannot allow busses that are based on a shared physical medium. Even a doubling of the bus medium in order to get a parallel redundant communication path is based on a too restricted fault model. One faulty node can prohibit the communication inside the entire network by randomly sending unintentional data. Point-to-point (P2P) networks on the other side demand some routing overhead to channel packages through the network. In the case of a faulty link, data can be sent via alternative paths. Beside the fault tolerance, P2P networks have the advantage of an extreme high total bandwidth. Thus, ReCoNet uses P2P communication.

For our ReCoNet we implemented a new communication protocol which supports dynamic rerouting as well as hardware and software task migration. The communication protocol works on different layers. Firstly, a node-to-node (N2N) protocol at the transport layer is defined. The N2N protocol is responsible for reliable communication between nodes (multi-hop). Secondly, a task-to-task (T2T) protocol is implemented that handels the task resolution in the network. The most important features of the communication protocol can be summarized as follows:

- *Priorities* are used to achieve different service levels in the network in order to prevent low priority messages blocking high priority messages.
- *Different sizes* are supported to allow the efficient transfer of simple sensor values as well as the efficient transfer of large binary (configuration) data. The size of the payload field can vary between 4 and 20 bytes per cell.
- *Celling* is used by the network driver to determine if the cell is a fraction of a multi-cell package or not.
- *Cut through* for small multi-hop communication latencies.
- *Data transfer rate* of $12.5Mb/s$.

Task2Task Communication One of the major design goals of the ReCoNet is to decouple structure from functionality. This means that a task is not forced to run on a predefined resource in the network if it does not demand special resources that are only available on specific nodes (e. g., a sensor). Thus, a *task resolution* mechanism has been integrated with the following requirements:

- Fast assignment of task addresses (TADs) to node addresses (NADs). Note, due to the rapid change of the TAD to NAD assignment, the resolution has to take place after each reconfiguration.
- Resolution of conflicts in case of multiple TADs which can occur in the context of task replication.
- Task resolution is an operating system task which is not visible to the user and does not affect the design style.

Line Detection If we want to compensate failures, we have to build failure detection mechanisms into a ReCoNet. 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 route for packages to be transferred over this link. As the link state is recognized in the transceiver ports of our implementation we chose the advantageous variant to perform the line detection completely in hardware.

All links of a ReCoNet support full duplex mode. If for example one transceiver port fails the failure is recognized in the adjacent node and not in the faulty one itself. Hence, the recognizing node needs to send its receive link state back to the adjacent node with the faulty transceiver. If this happens the hardware tries to reestablish the link by itself. If this was not successful, the hardware generates an interrupt to the CPU to switch over to an alternative routing. This will be described in the following section. The complete process takes place in less then a millisecond. In case that both lines will go down at the same time, this is recognized by an inactivity on the link. If no traffic is demanded by the application the transceivers generate keep alive message for their adjacent neighbors. If a link is down, the hardware tries to setup the connection periodically. This allows to include new links at runtime (e. g., after the repair of a faulty link).

Routing and Rerouting The main design objective described in this paper is to achieve a high degree of fault tolerance by self-optimizing network components. Consequently, we demand that the self-optimizing process itself has to be fault-tolerant. Hence, all network management processes have to operate distributed without global knowledge and robust with respect to faulty nodes.

The routing is based on a hierarchical approach and has some similarities to distance vector routing (also known as Bellman-Ford algorithm). Routing is defined as the problem of finding an output port for an associated node specified by its address. In our approach we have hardware routers each with a local routing table evaluating the primary routing function by a lookup. This permits the routing with a small latency. In addition, every node maintains a second routing function that determines an alternative port in software that has to differ from the port selected by the primary routing function. As a consequence, this allows for a fast reaction on link failures when packages have to be sent over alternative routes. In the case of a fault, it is mandatory to determine new routes as fast as possible. In this phase it is not important to find the best alternative route. The secondary routing function determines the alternative port before the fault occurs. For this purpose, we need the so-called *reachability set* (RS_p) for each output port p that contains the nodes that can be reached from p.

The routing is based on a special addressing scheme allowing a node to send a package to a neighbor node without knowing the target address. With UP being the set of ports connected to active nodes and $cost_r$ the cost function defined as the distance given by the number of hops to reach a node $r \in R$, then the routing algorithm initializes each node $\bar{r}$ as follows:

1: $UP := \emptyset$ \\at the beginning there is no port available
2: $\forall r \in R \backslash \{\bar{r}\} : ROUTE\_1st(r) := -1$ \\we have no primary route
3: $\forall r \in R \backslash \{\bar{r}\} : ROUTE\_2nd(r) := -1$ \\we have no 2nd route
4: $\forall p \in outputs(\bar{r}) : RS_p := \emptyset$ \\no neighbors
5: $\forall r \in R \backslash \{\bar{r}\} : cost_r := -1$ \\we don't know the cost to any node

If a link to a neighbor node $\widetilde{r}$ is established, we do the following in node $\bar{r}$:

1: $UP := UP \cup \{p\}$ \\put new link to set of connected ports
2: $HELLO(p, \bar{r}, 1)$ \\send via port p that r̄ \\is reachable with cost 1

The incoming $HELLO(p, \bar{r}, cost)$ message on port $\widetilde{p}$ starts then on the neighbor node $\widetilde{r}$ an update of the primary routing function:

1: $RS_{\widetilde{p}} := RS_{\widetilde{p}} \cup \{\bar{r}\}$ \\put neighbor node r̄ into the \\reachability set of port $\widetilde{p}$
2: $ROUTE\_1st(\bar{r}) := \widetilde{p}$ \\set route on port $\widetilde{p}$ for node r̄
3: $cost_{\bar{r}} := cost$ \\update cost function
4: $\forall q \in UP \backslash \{\widetilde{p}\} : ROUTE(q, \bar{r}, cost + 1, \widetilde{r})$ \\propagate routing
5: $ROUTE(\widetilde{p}, \widetilde{r}, 1))$ \\inform new node with own identity
6: $\forall r \in R|_{ROUTE\_1st(r) \neq -1} : ROUTE(\widetilde{p}, r, cost_r + 1, \widetilde{r})$ \\inform new node with all known routes

In line 4 we propagate the new route to node $\bar{r}$ to all connected neighbor ports different from the new one. The value of the cost function has to be incremented by 1 if the traffic is routed through node $\widetilde{r}$. If these nodes receive a route with lower cost they will update their own table and only in this case the routing information is propagated further through the network until all routing tables can be updated with a better route. In line 5 we set the new route in backward direction whereas in line 6 we inform the new neighbor about the complete routing information we know until now. As a result, the neighbor will test for each route if there was a better one or not. In the former case the local routing table is updated and the result is propagated further.

Because of the optimality principle, the route written to the primary routing table is optimal with respect to the number of hops. The secondary routing function stores the port with the second best cost function and is used when the hardware detects that the link of a specific port is down. Note that this algorithm needs no information about the network topology. Even the ports of a single router can be used in any order.

In the case of a link failure the primary routing table gets the values from the second routing table for all nodes that were originally routed through the now faulty port. This alternative table has usually a higher cost. Therefore, we have to propagate the new cost to the remaining neighbor ports that can locally decide whenever to update a routing table or not. Every node has only to store locally the two best routing alternatives. In the worst case, we have to propagate the routing information over the complete network. This can take up to $|R| - 1$ time steps with $|R|$ being the number of nodes in the ReCoNet. An additional delay can only occur if the node degree k is larger than two. But a node with degree $2 + k$ will reduce the longest possible path in a ReCoNet at least by k. Therefore, the sum of links passed by a package will always be less than $|R| - 1$. The proposed routing algorithm will not execute more than two consecutive commands (e. g., $HELLO$ and $ROUTE$) in a single node to update all routing tables. As a consequence, the routing is finished in $O(2 \cdot (|R| - 1)) = O(|R|)$ time steps.

Fig. 4 gives an example of the rerouting algorithm for the case that node A sends a package to node D. In the error free case I), the message will pass node B and C. If the link from B to C fails, node B will look into the second routing table guiding the package via the nodes G and H instead of node C. The route is invalidated by writing a -1 into the cost function table for the broken ports. If next node G fails its neighbor node B will have the only possibility to reach D via A. As a consequence node A will invalidate its port to node B and a new route is established via the nodes E, F, C.

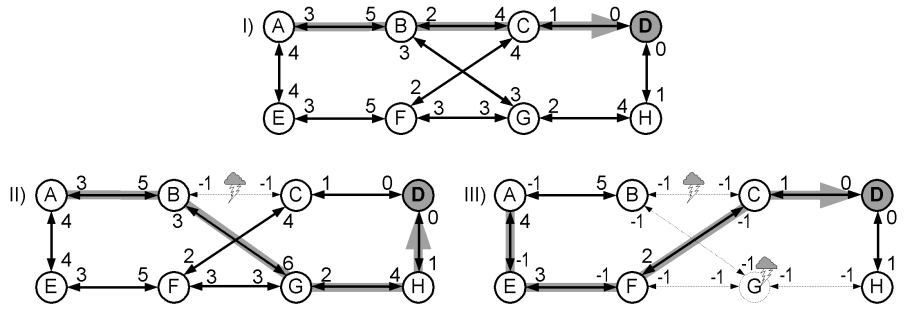

Fig. 4. Rerouting of a route $A \rightarrow D$ in the case of a link failure II) or the failure of a complete node III). The numbers specify the cost for each transceiver port to reach port D. If a transceiver port is not capable to reach D then the cost is set to -1.

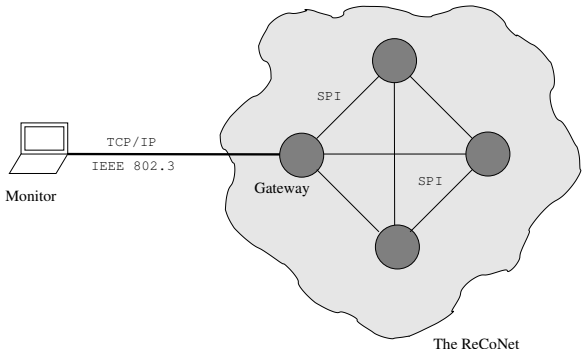

Fig. 5. In a network of several nodes, one node is acting as the gateway to the network monitor. The gateway collects all data and sends them via a TCP/IP connection to a host computer.

4.3 Online Analysis and Experimental Results

For analyzing the performance of the presented methods, a network monitoring system has been designed and integrated. It basically consists of a gateway collecting the data of the nodes in the network and a host computer for interpreting and displaying the collected data (see Fig. 5). The gateway is integrated on one dedicated node and each other node sends periodically its own status to the gateway. The information displayed by the monitoring system contains

- the binding of tasks to nodes in the network and the implementation style (hardware/software),
- a time line for each task, such that it can be analyzed when a task has been started on a node and when it migrated to another node,
- the data traffic on a link over the time,
- the topology of the network and
- the content of routing tables.

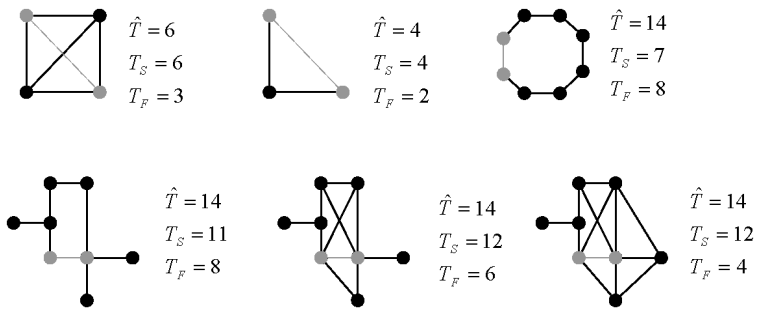

Fig. 6. Several scenarios are presented containing one broken link each. The given time steps denote a theoretical upper bound of routing time $\hat{T}$, the measured time after startup T_S and the measured time after a link defect T_F, respectively.

Routing The routing algorithm has been tested with different network topologies. Initially, the nodes had to discover routes after start-up. Then, during run-time links are disconnected and reestablished. In order to compare the results of the routing time after start-up T_S and in case of an error T_F with the theoretical upper bound $\hat{T} = 2(n-1)$, the results in Fig. 6 are presented in time steps. Note that the number of nodes in the network is denoted with n.

Performance of Task2Task Messaging As presented in Sec. 4.2 the data transfer rate depends on the message type which is chosen. Remember that dedicated messages are intended for e.g. sensor values with a good ratio between protocol data and small data entities. Another message type is intended for high volume data transfers having again a good ratio between protocol data and large data entities. With the constraint of a 50MHz CPU and a physical layer that supports a data transfer rate of $12.5Mb/s$, a data transfer rate of

- $1MB/s$ is reached for multi cell packets and
- $400KB/s$ could be transfered for single cell data packets during task to task communication.

5 Conclusion and Future Work

In this paper, we presented a new operating system infrastructure for reconfigurable networks which allow for the efficient design of dependable computing systems. The scope of this paper is on fault tolerance and a novel strategy was presented which deals with permanent faults or defects of communication links and computational resources. To establish this task, the basic OS features *dynamic rerouting*, *hardware* and *software task migration*, *hardware/software task morphing*, and *online partitioning* were discussed and implemented. In particular, the online partitioning that consists of a fast repair phase and an optimization phase is a key contribution in the area of modern embedded system design covering *reconfigurable networks* as well as *organic computing systems*.

In future work, we will present an application from the automotive industry running on our ReCoNet. Moreover, in order to support dynamic hardware reconfiguration at full scale, we consider the integration of state-of-the-art Xilinx FPGAs into our ReCoNet.

References

1. Lysecky, R., Vahid, F.: A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. In: Proceedings of the Conference on Design, Automation and Test in Europe, Paris, France (2004) 480–485
2. Streichert, T., Haubelt, C., Teich, J.: Online and HW/SW-Partitioning in Networked Embedded Systems. In: Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC'05), Shanghai, China (2005) 982–985
3. Walder, H., Platzner, M.: Online Scheduling for Block-partitioned Reconfigurable Devices. In: Proc. of Design, Automation and Test in Europe (DATE03). (2003) 290–295
4. Ahmadinia, A., Bobda, C., Koch, D., Majer, M., Teich, J.: Task Scheduling for Heterogeneous Reconfigurable Computers. In: In Proceedings of the 17th Symposium on Integrated Circuits and Systems Design (SBCCI), Pernambuco, Brazil (2004) 22–27
5. Ahmadinia, A., Bobda, C., Teich, J.: On-line Placement for Dynamic Reconfigurable Devices. To appear in Int. Journal of Embedded Systems (IJES) (2005)
6. Hecht, R., Timmermann, D., Kubisch, S., Zeeb, E.: Network-on-Chip basierende Laufzeitsysteme fr dynamisch rekonfigurierbare Hardware. In: In Proceedings of ARCS 2004, Augsburg, Germany (2004) 185–194
7. Baumgarte, V., May, F., Nückel, A., Vorbach, M., Weinhardt, M.: PACT XPP - A Self-Reconfigurable Data Processing Architecture. In: ERSA, Nevada (2001)
8. Chameleon Systems: CS2000 Reconfigurable Communications Processor, Family Product Brief. (2000)
9. Thomas, A., Becker, J.: Aufbau- und Strukturkonzepte einer multigranularen rekonfigurierbaren Hardwarearchitektur. In: In Proceedings of ARCS 2004, Augsburg, Germany (2004) 165–174
10. Bobda, C., Koch, D., Majer, M., Ahmadinia, A., Teich, J.: A Dynamic NoC Approach for Communication in Reconfigurable Devices. In: In Proceedings of International Conference on Field-Programmable Logic and Applications (FPL), Antwerp, Belgium (2004) 1032–1036
11. Dumitraş, T., Kerner, S., Mărculescu, R.: Towards On-Chip Fault-Tolerant Communication. In: Proceedings of the Asia and South Pacific Design Automation Conference 2003, Kitakyushu, Japan (2003)
12. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Comput. Surv. **34** (2002)
13. Streichert, T., Christian, H., Teich, J.: Distributed HW/SW-Partitioning for Embedded Reconfigurable Systems. In: Proc. of DATE05, Munich, Germany (2005)
14. Cybenko, G.: Dynamic Load Balancing for Distributed Memory Multiprocessors. Journal of Parallel and Distributed Computing **7** (1989) 279–301
15. Altera: Nios Development Board - Reference Manual,Cyclone Edition (2005) http://www.altera.com.
16. Altera: Nios II Processor Reference Handbook (2005)
17. Jean Labrosse: micro-C/OS-II Second Edition (2002)