# Communication Aware Optimization of the Task Binding in Hardware/Software Reconfigurable Networks

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

Department for Computer Science 12, University of Erlangen-Nuremberg, Germany,
e-mail : {streichert,dirk.koch,haubelt,teich}@cs.fau.de

**ABSTRACT**

In this paper, a new methodology for tolerating link as well as node defects in self-adaptive reconfigurable networks will be presented. Currently, networked embedded systems need a certain level of redundancy for each node and link in order to tolerate defects and failures in a network. Due to monetary constraints as well as space and power limitations, the replication of each node and link is not an option in most embedded systems. Therefore, we will present a hardware/software partitioning algorithm for reconfigurable networks that optimizes the task binding onto resources at runtime such that node/link defects can be handled and data traffic on links between computational nodes will be minimized. This paper presents a new hardware/software partitioning algorithm, an experimental evaluation and for demonstrating the applicability, an implementation on a network of FPGA-based boards.

**Index Terms:** Reconfigurable network, Task placement, Dynamic hardware/software partitioning

## 1. INTRODUCTION

Available *networked embedded systems*, e.g., in the field of automotive networks bind functionality statically onto electronic control units (ECUs). Thus, if a node fails, the functionality hosted by the control unit will be lost, and due to data dependencies, other functions on working nodes may not operate either. The same holds true for erroneous communication links which may lead to an isolation of a node or changing routes in point-to-point networks. However, in order to tolerate defects of computational nodes, it is necessary to replicate computational nodes and links such that a certain degree of redundancy is available. Obviously, introducing redundancy into embedded networks has an essential drawback concerning monetary costs, power consumption, size, weight, etc. Therefore, we will present an approach for tolerating permanent faults like node or link defects by separation of functionality from the physical hardware and rebinding tasks from defect nodes onto working nodes.

When using *reconfigurable devices* such as FPGAs together with internal CPU cores, it will be possible to assign tasks implemented in either hardware or software, dynamically to the resources in the network. Besides these reconfigurable devices, computational nodes in a network contain dedicated analog hardware for driving sensors and actuators which leads to a certain heterogeneity in the network. This irregularity is typical for embedded systems that consist of specialized nodes for certain purposes. For such *reconfigurable networks,* we will show how to reduce the degree of redundancy on the one hand while increasing *fault tolerance* and *flexibility* on the other hand. Essential for achieving these objectives is a novel class of algorithms called *online hardware/software partitioning* which determines a binding of tasks to available resources in heterogeneous networks.

Binding tasks onto computational nodes has been investigated in many research fields. The offline approach towards hardware/software partitioning has been considered by many researchers [1][2][3]. For example, Blickle [1] synthesizes so-called *Pareto-optimal* systems out of many design alternatives with the help of Evolutionary Algorithms. Such an approach helps a system designer for an unbiased decision making.

Also, so-called *load balancing algorithms* have received a considerable interest and solve the problem of task binding with the objective of homogeneously distributing the task loads onto CPUs at runtime. Note that the term task is used in the same sense as

functionality. The goals of load balancing are a) the reduction of latency or average response time, b) to provide fairness and c) reduction of overheads due to many context switches on highly utilized nodes. Load balancing approaches like *Token Distribution* [4][5], *Diffusion* [6][7], and so-called *Balancing Circuits* [8] are distributed algorithms and are thus applicable for fault-tolerant networked systems.

Another field of placing functionality on resources has been opened by Vahid et al. Based on a platform consisting of reconfigurable hardware and a CPU [9], a profiler extracts critical code regions, decompiles them and synthesizes them to hardware. Achieving an average speedup of 2.6 [10] for different benchmarks, this approach to dynamic hardware/software partitioning shows the potential of dynamically assigning tasks to software or hardware resources.

A first approach to online hardware/software partitioning for reconfigurable networks [11] [12] which is based on a combination of diffusion algorithms and bi-partitioning balances the load between the resources and thus, maximizes the amount of free resources on each single node. With this strategy the likelihood that the load of defect nodes or newly arriving tasks may be adopted by every node is increased.

Unfortunately, all the presented approaches either do not consider hardware/software reconfigurability at all or provide no extension to reconfigurable networks.

Also, heterogeneities due to sensors and actuators attached to single nodes in the network are not respected by the algorithms but strongly affect the placing of functionality onto nodes drastically. Moreover, none of these approaches consider the minimization of data traffic on links between computational nodes.

To overcome these drawbacks, we will explain in Sec. 2 a network model in which certain sensors and actuators can only be connected to certain nodes and tasks reading the sensor values or controlling the actuators have limited binding possibilities. Different to the approaches in [11] where the binding of tasks to resources is done with the objective of minimizing the load on each single resource, the methodology presented here tries to minimize the congestion on the communication links while respecting utilization constraints of hardware and software resources. The entire hardware/software partitioning approach runs in a distributed manner in the network and is described in Sec. 3. In Sec. 4, we present an implementation of the online hardware/software partitioning algorithm as well as an evaluation and comparison of our methodology to an approach with global knowledge.

## 2. CONCEPTS AND MODELS

In this paper, networks are considered consisting of hardware/software reconfigurable nodes. The networks have a fixed topology which is only influenced by node and link defects. Different to ad-hoc networks the size and the dynamic effects are not arbitrary. Assuming that all network nodes are connected via point-to-point connections and having more than one incoming and outgoing communication link, the considered networks should be fault-tolerant against permanent and transient faults as well as babbling idiot failures. Presuming a network with reconfigurable nodes allows for implementing a task in either software and run it locally sequentially together with other software tasks or in hardware (e.g., using reconfigurable hardware technology). Typical for embedded systems are dedicated I/O-interfaces which might not be available on each node and lead to a heterogeneous network structure.

Exemplarily, Figure 1 shows a network topology with four *computational nodes* $c_i \in C$, *sensors* $s_i \in S$, *actuators* $a_i \in A$ and communication links represented by the edges between the nodes $c_i$. The sensors and actuators are not connected to all nodes in the network, but only to some. Thus, the presented methodology in Sec. III has to be able to bind functionality onto a heterogeneous network structure. Similar to the network structure, the functionality is modeled by a so-called *sensor-controller-actuator chain* graph and distinguishes between *sensor tasks* $t_i^s$, *controller tasks* $t_i^c$, and *actuator tasks* $t_i^a$.

While sensor tasks produce data which are processed by one or more controller tasks, actuator tasks consume data. In Figure 1, such a sensor-controller-actuator chain is represented by gray nodes and edges in between where the edges represent data dependencies. Annotated to these nodes and edges are the following attributes which are necessary for the online hardware/software partitioning approach:

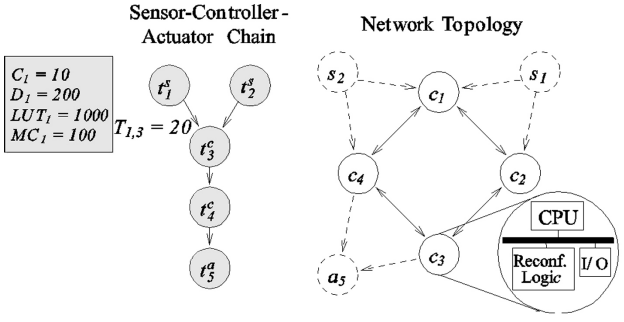


**Figure 1.** Functionality is modeled with a so-called *sensor-controller-actuator* chain. This functionality will be bound with certain restrictions onto the nodes of the network topology. Restrictions occur due to resource constraints where the parameters $C_1$, $D_1$, $LUT_1$, $MC_1$ denote the resource usage of a task.

**Execution Time ($C_i$) and Deadline ($D_i$):** In order to analyze the schedulability of a task $t_i^c$ on a CPU without violating deadlines, the execution time $C_i$ and its Deadline $D_i$ are required. With the help of schedulability analyses for real-time schedulers the utilization can be computed. Based on this utilization, it can be determined whether a task can be executed in software.

**Look-Up Tables (*LUT*), Memory-Cells (*MC*):** Before a task is assigned to hardware resources at runtime, it has to be checked if enough resources are available. Considering current FPGA-architectures, not only Look-Up Tables are required to implement the hardware functionality, moreover, memory cells or embedded RAM blocks are necessary. Additional placement constraints together with the shape of a hardware module might prevent the binding of tasks onto free hardware resources either. In summary, the feasibility of placing functionality onto hardware resources depends in our model not only on one parameter, but on a set of parameters. Note that due to this set of parameters, online hardware/software partitioning algorithms which are based on load balancing algorithms are not applicable. In general, load balancing algorithms have their legitimacy in architectures or topologies where only one parameter decides about executability.

**Migration Size ($M$):** This parameter is used to reduce the probability of migrating huge tasks between nodes. In FPGA-based architectures with a CPU, the migration size is given by the sum of the binary and the bit-stream size of task $t_i^c$.

**Data Traffic ($T$):** The data traffic $T_{i,j}$ produced by a Task $t_i^{\{s,c,a\}}$ and consumed by task $t_j^{\{s,c,a\}}$ is annotated to the edges $e_{i,j}$ between the node $t_i^{\{s,c,a\}}$ and $t_j^{\{s,c,a\}}$ in the sensor-controller-actuator chain.

Due to the heterogeneity caused by the sensors $s_i$ and actuators $a_i$ in the network topology, the binding of sensor tasks $t_i^s$ and actuator tasks $t_i^a$ is restricted. In particular, a sensor task $t_i^s$ is only allowed to be bound onto a corresponding node $s_i \in S$. In contrast, an actuator task $t_i^a$ is only allowed to be bound onto an actuator node $a_i$. We assume that all controller tasks $t_i^c$ may run on each computational node $c_i$.

Considering Figure 1, sensor task $t_1^s$ may be bound onto the sensor node $s_1$, but not onto $s_2$. Analogously, sensor task $t_2^s$ can run on $s_2$. For the controller tasks $t_3^c$ , $t_4^c$ , no binding restrictions exist. Thus, they are able to run on all computational nodes ($c_1$, $c_2$,$c_3$,$c_4$). For placing the actuator task $t_5^a$, only the actuator can be considered.

## A. Formal Model

The previously literally described model can be formally defined as follows:

DEFINITION 1 (NETWORK MODEL). *The entire system* $S(G^{tg}, G^{\{s,c,a\}})$ *consists of a* topology graph $G^{tg}$ *and a set of* sensor-controller-actuator chains $G^{\{s,c,a\}}$.

DEFINITION 2 (TOPOLOGY GRAPH). *The graph* $G^{tg}(S^{tg}, C^{tg}, A^{tg}, E^{tg})$ *consists of sensor nodes* $s_i \in S^{tg}$, *computational nodes* $c_i \in C^{tg}$ *and actuator nodes* $a_i \in A^{tg}$. *The edges* $e_i \in E^{tg} \subseteq S^{tg} \times C^{tg} \cup C^{tg} \times C^{tg} \cup C^{tg} \times A^{tg}$ *represent the connections between the three kinds of nodes.*

The nodes of the topology graph can be refined as:

DEFINITION 3 (COMPUTATIONAL NODE). *A computational node has ports* $p_i \in P$ *and* $|P| = \deg(c_j) + 1$. *While the ports* $p_i : i = 1 \ldots \deg(c_j)$ *are dedicated for communication between sensor nodes, computational nodes or actuator nodes, the port* $p_0$ *is dedicated for the node internal communication.*

For modeling the functionality, we define so-called *sensor-controller-actuator chains.*

DEFINITION 4 (SENSOR-CONTROLLER-ACTUATOR CHAIN). *The sensor-controller-actuator chain* $G^{sca}(T^s, T^c, T^a, E^{sca})$ *consists of sensor tasks* $t_i^s \in T^s$, *controller tasks* $t_i^c \in T^c$, *and actuator tasks* $t_i^a \in T^a$,. *The edges* $e_i^{sca} \in E^{sca} \subseteq T^s \times T^c \cup T^c \times T^c \cup T^c \times T^a$ *represent the data dependencies between the three kinds of tasks.*

Annotated to the edges and nodes can be different parameters which do not belong explicitly to the model. The parameters required by our online hardware/software partitioning approach were presented above.

In order to express, where a task $t_i^s$, $t_i^c$, $t_i^c$ is executed, we define a binary variable $b_{i,j}^{task}$:

DEFINITION 5 (TASK BINDING).

$$b_{i,j}^{task} = \begin{cases} 1 & : \textit{if } t_i^{\{s,c,a\}} \textit{ is executed by} \\ & \quad s_j \in S, c_j \in C \textit{ or } a_j \in A \\ 0 & : \text{else} \end{cases}$$

Equivalent to the *task binding*, we define a *traffic routing.*

DEFINITION 6 (TRAFFIC ROUTING).

$$b_{i,j}^{task} = \begin{cases} 1 & : \textit{if } e_i^{s,c,a} \textit{ is routed over } e_k^{tg} \textit{ and} \\ & \quad e_k^{tg} \textit{ is connected to a port } p_j \\ 0 & : \text{else} \end{cases}$$

In order to obtain a feasible communication, it might be necessary that an edge $e_i^{s,c,a} = (t_k, t_l)$ is routed over many edges $e_j^{tg}$. In particular it is required that the path constructed by the edges $e_j^{tg}$ connects the resources where the tasks $t_k$ and $t_l$ are bound to.

DEFINITION 7 (BINDING RESTRICTIONS). *Sensor tasks $t_i^s \in T^s$ may only be bound onto sensor nodes $s_i \in S$. Controller tasks $t_i^c \in T^c$ may be bound onto all computational nodes $c_j \in C$. Actuator tasks $t_i^a \in T^a$ may only be bound onto actuator nodes $a_i \in A$.*

Note that additional binding restrictions may occur during the hardware/software partitioning process due to the attributes annotated to edges or nodes in the graphs $G^{tg}$ and $G^{sca}$.

### B. Problem Statement

Online hardware/software partitioning aims at binding tasks to free hardware or software resources at runtime. Typically, the hardware/software partitioning is executed during the design phase of an embedded system. But since dynamic effects like node or link defects as well as new arriving tasks corrupt an optimal binding, it is inevitable to determine a new binding online. Our approach to online hardware/software partitioning consists of two main steps of which the second step will be refined later on. In Figure 2, these two steps are shown in an exemplifying scenario. The presented network topology consists of four computational nodes $c_1$,...,$c_4$, a sensor $s_1$ and an actuator $a_4$. The controller tasks $t_2^c$, $t_3^c$are bound onto the computational nodes $c_1$, $c_4$, sensor task $t_1^s$ is bound onto sensor $s_1$ and actuator task $t_4^a$ is bound onto $a_4$. Additional to the tasks $t_i^c$, replicas $t_i'^c$ are bound onto the computational nodes $c_1$, $c_2 \in C$. A requirement to this replica binding is that a task $t_i^c$ and its replica $t_i'^c$ must not be bound onto the same computational node $c_j$. Next, the computational node $c_1$ fails in Figure 2 and thus, all tasks bound onto this node are lost. During the *fast-repair* phase, the replicated tasks $t_i'^c$ become the main task $t_i^c$ and new routes for the task-to-task communication have to be established. Obviously, the tasks are sub-optimally bound after the fast-repair phase which will be improved during the optimization phase. The optimization phase tries to find a binding of tasks $t_i^c$ to resources such that the data traffic on the communication links is minimized and constraints to the CPU utilization or the usage of hardware resources, respectively, are not violated. In order to tolerate another node defect, replicated tasks $t_i'^c$ need to be created and bound onto the computational nodes $c_j$.

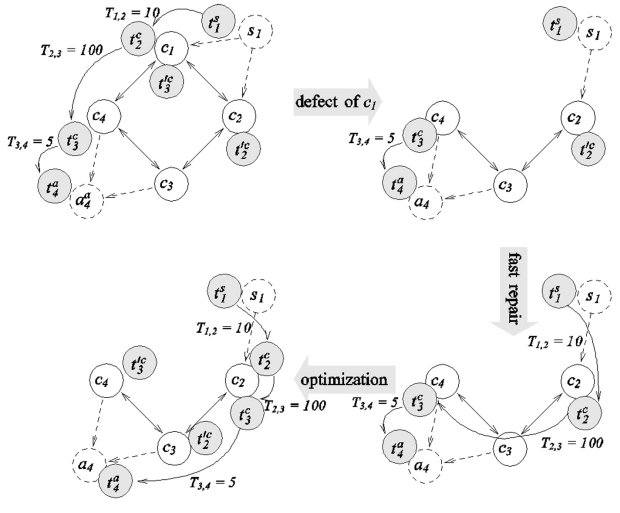


**Figure 2.** Four cases of a network: 1.) normal operation, 2.) after a node defect, 3.) after reestablishing communication and switching to replicated tasks, and 4.) after an optimization phase.

## 3. ONLINE HARDWARE/SOFTWARE PARTITIONING

As presented in Figure 3, the overall approach to hardware/software partitioning consists mainly of two phases. While the concepts and implementations of the first phase (fast repair) have been described in detail in [13], this paper concentrates on the second phase (optimization). Several constraints exist to this optimization phase:

- *distributed computation:* Due to fault-tolerance aspects, the binding of tasks has to be determined in a distributed manner at the computational nodes.

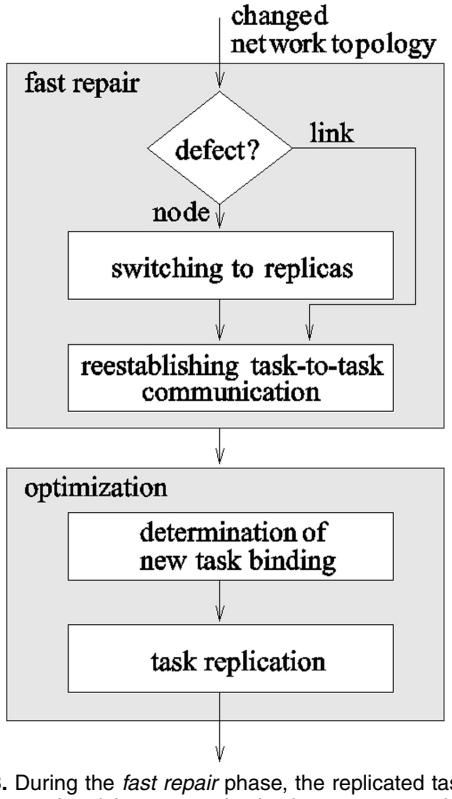


**Figure 3.** During the *fast repair* phase, the replicated tasks take over the control and the communication between two tasks will be reestablished. The optimization phase optimizes the binding of tasks and creates new replicas.

- *load definition:* No suitable equivalent to CPU utilization exists for the computation of hardware utilization in, e.g. FPGA-based architectures (ref. to Sec. 2).
- *data traffic:* Communicating tasks produce a certain amount of data that has to be transferred over links between the nodes. Therefore, a requirement to the algorithm is the optimization of traffic in the network
- *local knowledge:* Gathering of data to obtain global knowledge about the network is time consuming and produces communication overhead. Thus, it is desired to optimize the binding with limited information.

The next section shows how our approach fulfils these constraints by determining *improvement values* on each task and migrating tasks according to these values.

## A. Task Binding

The proposed methodology for determining an optimal binding is based on three improvement values: a) a *communication improvement* that tries to cumulate functionality with data dependencies, b) a *migration improvement* which reduces the overhead caused by the task migrations, and c) a *partitioning improvement* that tries to implement a task according to its favorite implementation style.

**Communication improvement:** The communication improvement $I_{i,j}^{com}$ is defined as the improvement for task $t_i^c$ if it is migrated from node $c_m$ over port $p_j$ to a neighboring computational node ($j \neq 0$):

$$I_{i,j}^{com} = \sum_{l=0}^{\deg(c_i)} \sum_{k=1}^{|T|} T_{i,k} \cdot r_{k,l} \qquad (1)$$

with

$$r_{k,l} = \begin{cases} -1 : \textit{if traffic } T_{i,k} \textit{ is routed over } p_l \textit{ withe } l \neq j \\ 1 : \textit{if traffic } T_{i,k} \textit{ is routed over } p_l \end{cases}$$

The outer sum adds $\deg(c_i) + 1$ terms because not only the traffic over the ports of the nodes but also the node internal traffic needs to be considered. Considering Figure 4 as an exemplifying binding where the communication improvement $I_{1,3}^{com}$ for migrating task $t_1^c$ over port $p_3$ should be computed, we will obtain the following:
$I_{1,3}^{com} = -80-20+10+100.$

Afterwards, the communication improvement $I_{i,j}^{com}$ has to be normalized. For this normalization, the maximal absolute value of $I_{i,j}^{com}$ of all tasks $t_i^c$ will be computed if migrated over a certain port $p_j$:

$$I_{\max}^{com} = \max_{\forall t_i \text{ at } c_m, \forall p_j \in P} |I_{i,j}^{com}| \qquad (2)$$

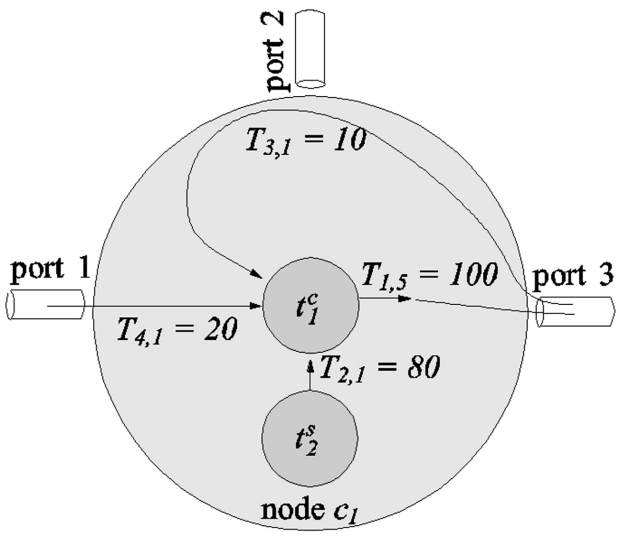


**Figure 4.** Two tasks $t_1^c$, $t_2^c$ at a computational node are shown. The inter task communication is denoted with directed edges to/from the ports or between $t_1^c$ and $t_2^c$. Annotated to each edge is the traffic between two tasks.

**Migration improvement:** For the determination of the migration improvement $I_i^{mig}$, the size $M_i$ of the bit-stream and binary of task $t_i^c$ which needs to be migrated is required.

Then, the migration improvement for migrating task $t_i^c$ over a port $p_j$ is simply defined as:

$$I_i^{mig} = M_i \qquad (3)$$

Note that this is not really an improvement of the task binding. It just avoids transferring huge data entities over the communication channels in the network. Again the improvement $I_i^{mig}$ needs to be normalized:

$$I_{\max}^{mig} = \max_{\forall t_i \text{ at } c_m} |I_i^{mig}| \qquad (4)$$

**Partitioning improvement:** The partitioning improvement $I_{i,j}^{par}$ is required for optimizing the implementation style (hardware/software) of a task $t_i^c$. For certain applications, e.g., video stream processing, it might be desirable to implement a task in hardware while alternatively, a state-machine might be efficiently executed in software. However, assuming that each task $t_i^c$ has a favorite implementation style, a likelihood value $l_i \in \mathbb{R}$ with $0 \leq l_i \leq 1$ will be defined at design time. The decision whether a task is better implemented in hardware or software can be taken based on resource utilization or a quality of service. The resulting improvement $I_{i,j}^{par}$ will be defined as:

$$I_{i,j}^{par} = l_i \cdot q_{i,j} \qquad (5)$$

with

$$q_{i,j} = \begin{cases} 1 : \text{if } t_i^c \text{ was implemented in its favorite style and can be implemented in its favorite style after migration over } p_j \\ -1 : \text{if } t_i^c \text{ was implemented in its favorite style and can only be implemented in its non favorite style after migration over } p_j \\ 0 : \text{else} \end{cases}$$

The resulting improvement $l_{i,j}$ for migrating a task $t_i^c$ over port $p_j$ to a neighboring computational node is:

$$I_{i,j} = \frac{I_{i,j}^{com}}{I_{\max}^{com}} - \frac{I_{i,j}^{mig}}{I_{\max}^{mig}} + I_{i,j}^{par} \qquad (6)$$

As shown in Figure 5, this improvement will be determined for all migratable tasks $t_i^c \in T_m \subseteq T$ and all ports $p_j$ of node $c_m$. After calculating the improvement values for the migratable tasks, negative improvement values might be in the list and can impair the current binding. Therefore, two possibilities exist, a) to remove all negative improvement values or b) to allow for negative improvement values depending on the migration count $mc_i$ of task $t_i^c$. In the next step, the algorithm selects the task $t_i^c$ with the highest improvement value $I_{i,j}$ and asks the neighboring computational node at port $p_j$ if the task can be scheduled on the CPU or bound onto the reconfigurable hardware device, respectively (see Figure 5). If enough resources are available for scheduling/placing the task, the task will be migrated and all improvement values will be deleted. Otherwise, only the improvement value $I_{i,j}$ for the considered port $p_j$ and task $t_i^c$ will be deleted. These two steps of selecting the task with the highest improvement value and trying to migrate it, is repeated locally until no improvement value $I_{i,j}$ remains. Note that the set of migratable tasks $T_m$ contains only tasks with a *migration counter* less than a certain limit: $mc_i \leq mc_{limit}$. The counter $mc_i$ is incremented after each migration of task $t_i^c$ and reset after a node or link defect. With this constraint, the algorithm will terminate by preventing an alternating behavior. All in all, our methodology runs asynchronously in the network, i.e., there are no periodic migration rounds. Since the routing needs to be fixed before calculating the improvement values of the tasks on a node, it is not possible to migrate tasks on different nodes simultaneously. Therefore, a token will be placed onto an arbitrary node. If a node or link defect occurs, the node with the token will start with the calculation of improvement values and migrates a task to a neighboring node. Along with this migrated task a token will be transferred and the node which receives the token may start the calculation of improvement values. If the node will not migrate a task, the token will be passed to an arbitrary neighboring node. This strategy is derived from the class of hill climbing algorithms, where optimization runs are repeatedly started from arbitrary initial points. The algorithm stops after the token has been transferred a certain number of times.

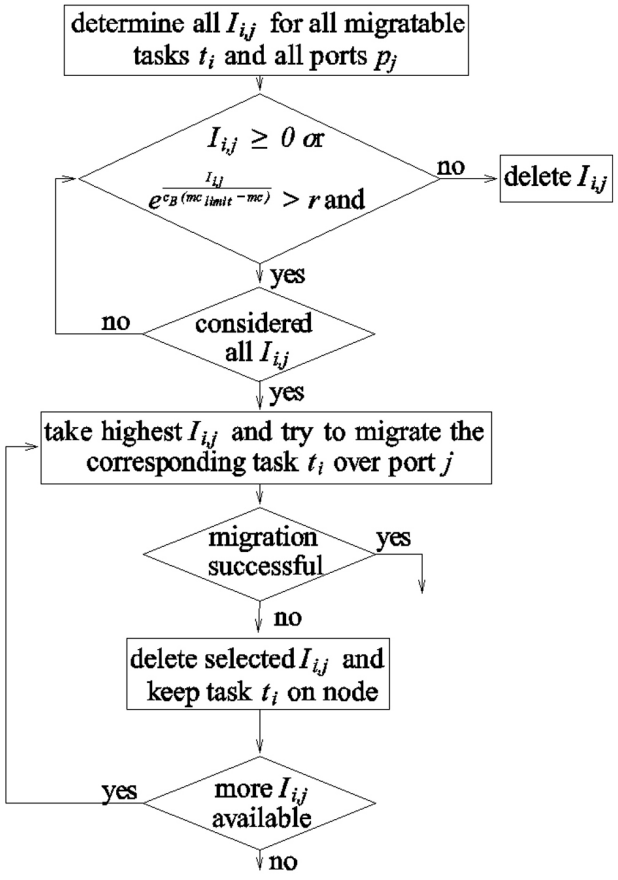


**Figure 5.** The flow diagram shows the complete process of binding optimization that will be locally determined on each computational node in the network.

## 4. RESULTS

In this section a detailed evaluation of our contribution is given. The objectives of the two phases in Figure 3 are diverse:

While the fast repair phase aims at reestablishing the functionality, the optimization phase targets at improving the binding of tasks to network nodes.

**Evaluation of the Fast Repair Phase:** We implemented our approach to distributed online hardware/software partitioning on a network of four reconfigurable FPGA-based boards incorporating a RISC-CPU and additional logic for implementing

hardware. The operating system *microC-OS II* [14] has been extended such that node and link defects are automatically detected [15]. Additionally, a *task manager* has been designed and implemented which gathers information about the task binding and locally decides where to bind the tasks. This decision will be taken by our approach to online hardware/software partitioning.

On top of this network infrastructure, a *driver assistance* application has been implemented. With the help of pattern recognition algorithms, the application tracks the lane and in case of an unintended lane change, the assistant sets off an acoustic warning. The entire driver assistant runs on the network in a distributed manner. Thus, if one node fails, the tasks have to be dynamically reassigned to free resources in the network. With this implementation, we determined the worst case repair times of the *fast repair phase* for network topologies with varying diameter.

In the first case, the time for determining new routes after a node or link defect is evaluated. The experimental results as presented in Figure 6 show that the rerouting time increases linearly with the network diameter and about 0.89ms are required in average for each hop. If a task is not accessible any more a replica needs to be activated and the information about this activation has to be distributed to the other nodes in the network. In Figure 7, the time for activating a replica and distributing the information is presented. The activation of replicas takes about 4ms in our implementation and afterwards the information is broadcasted which depends linearly on the network diameter.

**Evaluation of the Optimization Phase:** For a detailed evaluation of our approach to online hardware/software partitioning, we implemented a behavioral model of the previously described network, too. This model has been supplied with nine different scenarios where each scenario consists of a sensor-controller-actuator-chain and a network topology. Three different scenarios were created with 40 tasks and 10 computational nodes. The next three scenarios had 80 tasks and 20 nodes and the last three scenarios had 200 tasks and 50 nodes. Our distributed approach started from an arbitrary initial binding of tasks onto computational nodes. For each scenario, 10 initial bindings were determined such that in total 90 test cases were examined. Starting with an arbitrary binding of the tasks onto the computational nodes of the network topology, the algorithm tries to improve the binding by migrating functionality between the hardware and software resources in the network. After each migration step, we determine the overall traffic $\boldsymbol{T}$ in the network and the fraction of tasks which are executed in their non-favorite implementation style $\boldsymbol{N}$:

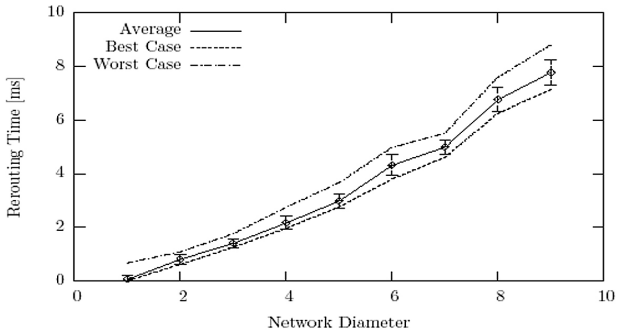


**Figure 6.** Time for determining new routes after a link defect in dependency of the network diameter. Vertical bars indicate the standard deviation.

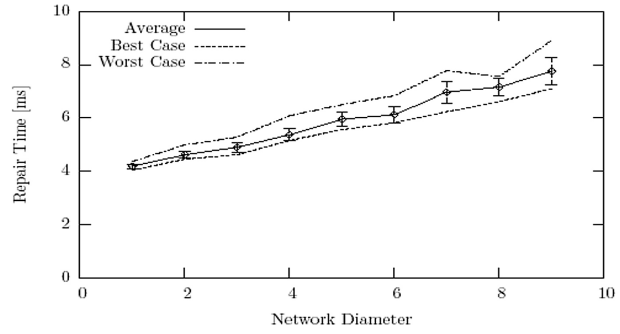


**Figure 7.** Time for activating a replica and distributing the information about this activation to other nodes.

$$T = \sum_{l=1}^{|E_{sca}||E_{tg}|} \sum_{k=1} T_{i,j} \cdot b_{i,j}^{traffic}\ ,\ N = \frac{\sum_{i=1}^{|T^{\{sca\}}|} n_i}{|T^{\{sca\}}|} \qquad (7)$$

with

$$n_i = \begin{cases} 1: \textit{if } t_i^{\{s,c,a\}} \textit{ is implemented in its non} - \textit{favorite style} \\ 0: \text{else} \end{cases}$$

We compared the solutions $s_i = (T,N)$, $s_i \in S$ of each optimization run with a hardware/software partitioning algorithm based on Evolutionary Algorithms (EA) [16] that incorporates global knowledge. Note that our algorithm tries to optimize the binding only with local knowledge. The EA-based approach, in contrast determines a reference set $R_{EA}$ of so-called Pareto-optimal solutions $(T,N) = r_{EA} \in R_{EA}$. The minimal normalized distance $d(s)$ between each $s \in S$ and $R_{EA}$ is then calculated as follows:

$$d(s) = \min_{r_{EA} \in R_{EA}} \left\{ \sqrt{\frac{|s_T - r_T|}{r_T^{max} - r_T^{min}} + \frac{|s_N - r_N|}{r_N^{max} - r_N^{min}}} \right\} \qquad (8)$$

where a smaller distance indicates a better solution.

For each locally determined solution $s \in S$, we computed the distance $d(s)$ to the reference set $R_{EA}$ with the Pareto-optimal solutions. The distance between the Pareto- front determined by the EA-based

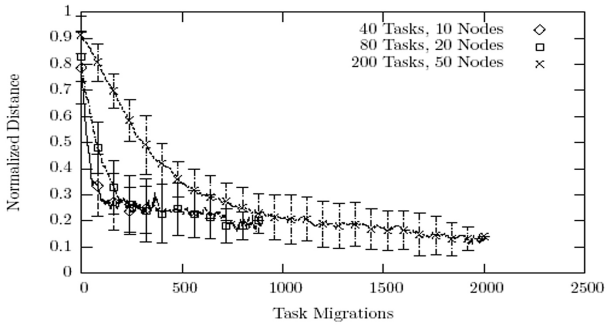


**Figure 8.** Distance and its standard deviation between the Pareto-optimal partitions determined by an EA and the online partitioner over time (number of task migrations).

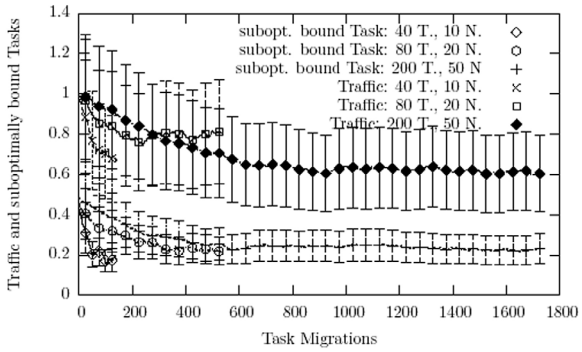


**Figure 9.** Normalized traffic and percentage of suboptimally bound tasks over time (number of task migrations).

approach and the solution *s* after each task migration is shown in Figure 8 and denotes how close the solutions found by the proposed algorithm converged towards the reference solutions. Each plot in Figure 8 represents one test case with either 40 Tasks/10 Nodes, 80 Tasks/20 Nodes or 200 Tasks/50 Nodes. Due to the migration counter, the smaller test cases terminate earlier than the bigger test cases, but it can be clearly seen that our methodology improves the initial partitioning and approaches a global optima. In Figure 9, the two objectives (traffic $T$ and percentage of suboptimally implemented tasks $N$) after each task migration are shown. For these plots, we normalized the traffic by dividing by the maximal traffic of each optimization run. Interestingly, our algorithm is able to reduce the traffic $T$ by at least 20%.

Additionally, the number of suboptimally implemented tasks $N$ which has been about 50% at the beginning has been reduced to 25% in average.

## CONCLUSIONS

Online hardware/software partitioning aims at binding functionality onto free resources at run-time. While other approaches solved this partitioning problem offline or just assign software tasks dynamically to network nodes, our approach solves the partitioning problem at run-time. Moreover, it runs in a distributed manner, requires only local knowledge and respects various resource limitations on the nodes. While assigning functionality to nodes, our algorithm successfully minimizes the congestion in the network.

All in all, we presented an online hardware/software partitioning approach for FPGA-based or general reconfigurable networks.

## ACKNOWLEDGEMENTS

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

## REFERENCES

[1] T. Blickle, J, Teich, L. Thiele, “System-Level Synthesis Using Evolutionary Algorithms”, Design Automation for Embedded Systems, Kluwer Academic Publishers, Boston, 3, pp. 23-62, 1998.
[2] V. Kianzad, S.S. Bhattacharyya, “CHARMED: A Multi-Objective Co-Synthesis Framework for Multi-Mode Embedded Systems”, Proc. of the 15th IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP’04), Galveston, U.S.A., pp. 28-40, 2004.
[3] M. López-Vallejo, J.C. López, “On the Hardware-Software Partitioning Problem: System Modeling and Partitioning Techniques”, ACM Transactions on Design Automation of Electronic Systems, 8(3):269-297, 2003.
[4] F. Meyer auf der Heide, B. Oesterdiekho_, R. Wanka,“Strongly adaptive token distribution”, Algorithmica, 15:413-427, 1996.
[5] D. Peleg, E. Upfal, “The token distribution problem”, ORSA Journal on Computing, 18:229-243, 1989.
[6] J. E. Boillat, “Load balancing and poisson equation in a graph”, Concurrency: Practice and Experience, 2:289-313, 1990.
[7] G. Cybenko, “Dynamic Load Balancing for Distributed Memory Multiprocessors”, Journal of Parallel and Distributed Computing, 7:279-301, 1989.
[8] J. Aspens, M. Herlihy, N. Shavit, “Counting Networks”, Journal of ACM, 41:1020-1048, 1994.
[9] R. Lysecky, F. Vahid, “A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning”, Proc. of the conference on Design, automation and test in Europe (DATE’04), pp. 480-485, 2004.
[10] G. Sitt, R. Lysecky, F. Vahid, “Dynamic Hardware/Software Partitioning: A First Approach”, Proceedings of Design Automation Conference 2003, Anaheim, California, 2003.
[11] T. Streichert, C. Haubelt, J. Teich, “Distributed HW/SW-Partitioning for Embedded Reconfigurable Systems”, In Proc. of Design, Automation and Test in Europe (DATE’05), Munich, Germany, 2005.
[12] T. Streichert, C. Haubelt, J. Teich, “Online Hardware/Software Partitioning in Networked Embedded Systems”, In Proc. of Asia and South Pacific Design, Automation and Test Conference (ASP-DAC’05), Shanghai, China, pp. 982-985, 2005.
[13] D. Koch, T. Streichert, S. Dittrich, C. Strengert, C. Haubelt, J. Teich, “An Operating System Infrastructure for Fault-Tolerant Reconfigurable Networks”, In Proc. of Architecture of Computing Systems (ARCS’06), Frankfurt (Main), Germany, 2006.
[14] “microC-OS II”, http://www.micrium.com/.
[15] “ReCoNets-Demonstrator”, www.reconets.de.
[16] C. Haubelt, “Automatic Model-Based Design Space Exploration for Embedded Systems – A System Level Approach”, Univ. of Erlangen-Nuremberg, Germany, Dr. Köster, Berlin, ISBN 3-89574-572-3, 2005.