Online Hardware/Software Partitioning in Networked Embedded Systems *

Thilo Streichert, Christian Haubelt, Jürgen Teich
University of Erlangen-Nuremberg, Germany
{streichert,haubelt,teich}@cs.fau.de

Abstract— Today's embedded systems are typically *distributed* and more often confronted with time-varying demands. Existing methodologies that optimize the partitioning of computational tasks to hardware (HW) and software (SW) at compile-time become obsolete or inefficient in this context as the optimal use of existing resources cannot be foreseen. Here, we investigate a discrete iterative algorithm that balances the load of a *HW/SW partition* online: Once there are changing computational demands, the system will dynamically assign tasks to reconfigurable HW or SW resources and migrates tasks to other nodes if necessary. For this purpose an *Evolutionary Algorithm* combined with a discrete version of a *diffusion algorithm* is presented. Concerning the diffusion algorithm, we will show theoretically and by experiment that our version is run-time optimal in a linear number of steps.

I. Introduction

Distributed and adaptive embedded platforms [3, 5] are becoming more and more important for applications in the area of body area networks, ambient intelligence and automotive technology. In order to be able to cope with different and time-variant application demands, these systems must be adaptive to changing requirements. In the context of networked embedded systems the computational requirements can often be unpredicted and changing over time such that an offline partitioning methodology can only provide tradeoff-solutions with respect to conflicting objectives such as equal load, high performance, low power, etc. that are not efficient for every scenario. Therefore, we need to rethink the HW/SW partitioning problem again for adaptive networked systems. Due to unforeseen demands, the system should be able to react to changing requirements by dynamically distributing its load among the available resources.

In this paper, we present a strategy for *online partitioning* and *dynamic load balancing.* which is based on a class of algorithms that are successfully applied in the context of dynamic load balancing called *diffusion algorithms* [2]. Here, we propose a discrete version of an optimal local iterative diffusion algorithm where only full task entities can be exchanged between nodes. and show theoretically as well as by experiment the quality of the algorithm.

To the best of our knowledge, there exists no previous work on *online HW/SW partitioning* for networked embedded systems yet. The work of Vahid et al. in [6] describes load balancing on a platform consisting of a microprocessor and reconfigurable HW but there is no straightforward extension of this methodology for networks with more than two nodes.

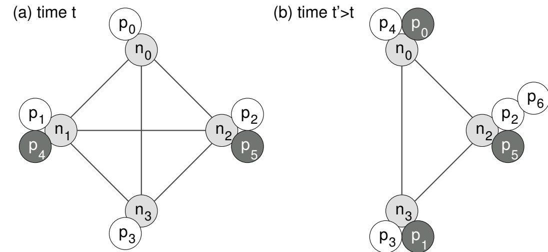

Fig. 1. Model of a ReCoNet at certain time instances t

This paper is organized as follows: Section II introduces the concept and models of the online partitioning problem. Section III introduces the diffusion-based algorithms for migration of tasks between nodes. Section IV proposes a discrete version of the diffusion schemes and also proves theoretic bounds imposed on our algorithm. Section V shows the quality of our new algorithm by experiment.

II. Concepts and Models

In this paper, we consider embedded systems consisting of *networked HW/SW reconfigurable nodes* with the following properties:

- interconnect: A network consists of computational nodes connected by bidirectional point to point links.
- embedded: This feature requires the optimization of competing objectives like power, latency or area.
- HW and SW reconfigurable: Tasks can be executed either in SW or in HW on a node.

In such a reconfigurable network (ReCoNet), two possible scenarios exist where a HW/SW (re)partitioning of processes becomes important or even necessary: The first is the possible failure of a link or of a computational node at a certain time. The second is for optimality reasons due to either finishing tasks or yet unknown arriving tasks. The structure of a network is given by a so-called *architecture graph* $g_{\mathrm{a}} = (N, C)$, where N is the finite set of nodes and $C \subseteq N \times N$ is the finite set of connections.

Example 1 *Figure 1(a) shows a specification of a ReCoNet at time t. Assigned to each resource node is a set of tasks where each task has two possible implementation styles (black nodes denote HW implementations/white nodes denote SW implementations). The architecture graph in Figure 1(a) consists of the nodes $N = \{n_0, n_1, n_2, n_3\}$. In Figure 1(b), the network is shown at time $t' > t$ where node n_1 has failed. The processes have been reassigned to the remaining nodes and due to optimality reasons the implementation style has changed. Additionally, a new task p_6 has arrived.*

It is therefore the requirement of the online partitioning algorithm to distribute the tasks again to nodes in the

*Supported in part by the German Science Foundation (DFG), SPP 1148 (Rekonfigurierbare Rechensysteme)

remaining network, e.g., to balance the load evenly between the resources or to satisfy performance constraints. Without loss of generality, we assume that the architecture graph is undirected in the following. For simplicity, we also assume that each computational task p_j may be assigned to each node in the network without restriction.

Definition 1 (Temporal partition) *Given is a set of processes $P = \{p_1, ..., p_n\}$ and an undirected architecture graph $g_{\mathrm{a}} = (N, C)$, where N is the finite set of nodes and $C \subseteq N \times N$ is the finite set of connections. At any instant of time t, $P(t) \subseteq P$ denotes the subset of active processes, and $g_{\mathrm{a}}(t)$ denotes the subgraph of g_{a} including all active resources $N(t)$ and links $C(t)$. Then, a* temporal HW/SW partition *at time t is an assignment of each task $p \in P(t)$ to a resource $N(t)$ as well as the indication whether the task is implemented in HW or SW.*

Here, the assignment of tasks to nodes in the network as well as their implementation in HW or SW can change over time. In order to allow the migration of a SW task to a HW task, the ability of HW reconfiguration of the node in a ReCoNet is required.

Definition 2 (Workload characterization) *Each task $p_j \in P(t)$ causes a unique load $w_j^H \in \mathbf{R}_0^+$ on resource $n_i \in N(t)$ if implemented in HW and a load of $w_j^S \in \mathbf{R}_0^+$ if implemented in SW. In HW, the load is defined as the fraction of required area and maximal available area. In SW, we define the load as the fraction of execution time and period.*

III. Basic Diffusion Algorithm

First, we will introduce the basics of *diffusion algorithms*. In Section 4, we will extend these algorithms to the discrete case for solving the online HW/SW partitioning problem. Given an architecture graph and an assignment of tasks to nodes, we want to move load across edges so that finally the load weight of each node is equal. Characteristic to a diffusion-based algorithm, introduced by Cybenko [2], is that iteratively each node is allowed to move any size of load to each of its neighbors along the edges $c \in C(t)$. The *quality* of an algorithm is measured in 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.

Definition 3 (local iterative diffusion algorithm) *A local iterative load balancing algorithm performs iterations on the nodes of g_{a}. The load exchanges between two adjacent nodes are determined in each iteration as follows:*

$$
\begin{aligned}
y_c^{k-1} &= \alpha(w_i^{k-1} - w_j^{k-1}) \quad \forall c = \{n_i, n_j\} \in C \qquad (1)\\
x_c^k &= x_c^{k-1} + y_c^{k-1} \quad \forall c = \{n_i, n_j\} \in C\\
w_i^k &= w_i^{k-1} - \sum_{c=\{n_i,n_j\} \in C} y_c^{k-1}
\end{aligned}
$$

In the above definition, y_c^k is the amount of load sent via edge c in step k with α being a parameter for the fraction of the load difference, x_c^k is the total amount of load sent via edge c until iteration k and w_i^k is the load of node i after the k-th iteration.

If arbitrary real-valued load portions may be sent at each iteration k, then it has been shown in [2] that the iteration converges to the average load $\bar{w}$. The number of iterations needed to obtain a certain error bound may be large and is in general not known a priori.

A slight modification of the above iteration scheme that works with changing values of α in each iteration k has shown that the convergence speed can be drastically improved to exactly $m-1$ iterations as follows without sacrificing the fact that the scheme is l_2-optimal in the total caused flow during the total number of iterations [4]. Simply choose $\alpha = \frac{1}{\lambda_k}$ in the iteration of Eq. (1) where λ_k, $1 \leq k \leq m-1$ denotes a certain numbering for the nonzero eigenvalues of L. L is called the Laplacian-matrix of the network and defined as $L = D - B$ where D contains the node degrees as diagonal entries and B is the adjacency matrix of the network. Hence, $\alpha = \alpha_k = \frac{1}{\lambda_k}$, and in each iteration k, a node n_i adds a flow of $\frac{1}{\lambda_k}(w_i^{k-1} - w_j^{k-1})$ to the flow of edge $\{n_i, n_j\}$, choosing a different eigenvalue for each iteration. We obtain an equally balanced load distribution after exactly $m-1$ iterations, while the created flow is also l_2-optimal.

IV. Online Hw/Sw-Partitioning

A. Multiobjective Optimization Flow

Due to the promising results presented later on, we propose the following 2-level *online partitioning algorithm* (see Figure 2):

Evolutionary Algorithm: In the first step (*init_pop()*), our algorithm creates an initial population with individuals, where each task is randomly assigned to HW or SW. Later on this bi-partition will be improved, using generic operators. This step is called *implementation_selection()*.

Diffusion: For a given initial binding of tasks, the initial loads w_i^0 of each active resource node n_i may be computed by summation of all loads of either HW or SW tasks. Then, run the diffusion algorithm (*discrete_diffusion()*) once for the HW and once for the SW tasks. This results in a new task assignment.

Objectives: Our heuristic introduced here, tries to find a bi-partition such that the load is balanced between HW and SW, i.e., we minimize $\left|\sum_{i=1}^{|N|} w_i^{\mathrm{S}} - \sum_{i=1}^{|N|} w_i^{\mathrm{H}}\right|$. With this strategy, a subsequent iterative diffusion will then create most likely task assignments that enable a good *load reserve* on each active node which is important with respect to achieve fast repair times in case of unknown future node or link failures. This balance cannot be reached by direct application of our discrete diffusion algorithm, introduced later on. The diffusion algorithm balances only the load between nodes, but it is not able to balance the HW/SW load. The second objective is the effective load balance. We try to minimize $|\bar{w} - \max\{\max_{\mathrm{i:n_i \in N}}\{w_i^S\}, \max_{\mathrm{i:n_i \in N}}\{w_i^H\}\}|$. This objective is zero for fully balanced loads. Here, we apply an *evolutionary algorithm* to encode the implementation selection as a binary string. This evaluation of the two objectives is denoted with *terminate?*.

B. Discrete Diffusion Algorithm

With this discrete diffusion algorithm, we have to overcome two problems:

First of all, it is advisable not to split one process and

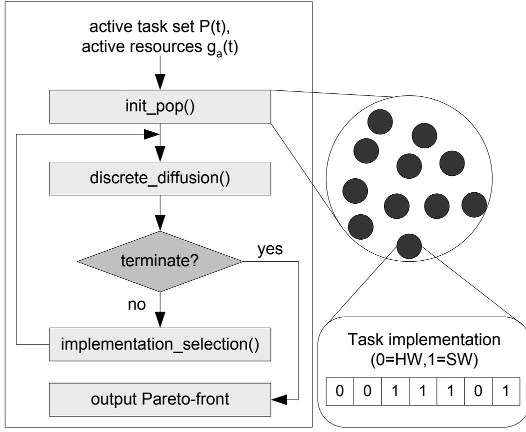

Fig. 2. Flowchart for online HW/SW partitioning

distribute it to multiple nodes. This increases the data traffic in the network.
Secondly, since the diffusion algorithm is an alternating iterative balancing scheme, it could occur that negative loads are assigned to computational nodes.

From now on, let $ycont_c^k$ be the real-valued continuous and $ydisc_c^k$ the discrete flow on one edge c in iteration k. Analog, all variables are extended with *cont* for the continuous and *disc* for the dicrete diffusion algorithm. Then the discrete diffusion algorithm works in each iteration k with $k = \{1, ..., m-1\}$ as follows:
In the first step of an iteration, the continuous flows $ycont_c^k$ on all edges are computed. In the next step, each node tries to fulfill this continuous flow for its incident edges. Thus, it sends or receives tasks, respectively. Here, we encounter another $\mathcal{NP}$-complete problem where we have to choose a certain number and size of tasks of one node in order to fulfill the optimal flow. Therefore, we randomly choose tasks as long as the discrete flow $ydisc_c^k$ doesn't exceed the continuous flow or no more tasks remain on the node:

$$ydisc_c^k \leq ycont_c^k + e_c^{k-1} \qquad \text{with} \qquad e_c^0 = 0 \tag{2}$$

In this equation, we already respect the error e_c^k done in the previous iteration, by not fulfilling the optimal real-valued flow. We are able to compute this error ${e_c}^k$ that occured in the current iteration step as follows:

$${e_c}^k = ycont_c^k + e_c^{k-1} - ydisc_c^k \quad \forall c = \{\mathrm{n_i}, \mathrm{n_j}\} \in C \tag{3}$$

In order to minimize the final error e_c^m, the error of the current iteration step is respected in the following iteration step, see Eq. (2). From here on, we start with a next iteration until the last iteration step $m-1$. After the last iteration step, the remaining error e_c^{m-1} is minimized in one additional adjustment step in which nodes exchange tasks according to the error after $m-1$ iterations:

$$e_c^m = e_c^{m-1} - y_c^{adj} \tag{4}$$

y_c^{adj} denotes the flow in this adjustment step.

Now, let us compare the behavior of our discrete diffusion algorithm and its continuous counterpart.

Theorem 1 *The discrete diffusion algorithm requires m steps.*

Proof 1 *We introduced just a single adjustment step.* □

Theorem 2 *The overall congestion by the discrete diffusion algorithm is less or equal than that caused by its continuous counterpart which is known to be l_2-optimal.*

Proof 2 *Let $xcont_c$ and $xdisc_c$ denote the whole load transmitted via edge c of the continuous and the discrete diffusion algorithm, respectively. Then we first show that*

$$xcont_c \geq xdisc_c \tag{5}$$

no matter what network, initial load and edge c. With

$$xcont_c = \sum_{k=1}^{m-1} ycont_c^k \qquad where \qquad ycont_c^k \geq 0 \tag{6}$$

$$and \quad xdisc_c = \sum_{k=1}^{m-1} ydisc_c^k + y_c^{adj} \tag{7}$$

where y_c^{adj} is the flow via edge c in the last adjustment step which is always less than or equal to e_c^{m-1}, see Eq. 2:

$$e_c^{m-1} \geq y_c^{adj} \tag{8}$$

Replacing $ycont_c^k$ in Eq. (6) with Eq. (3), leads to:

$$xcont_c = \sum_{k=1}^{m-1} \left(e_c^k - e_c^{k-1} + ydisc_c^k \right) \qquad with \qquad e_c^0 = 0 \tag{9}$$

Inserting $xcont_c$ from Eq. (9) and $xdisc_c$ from Eq. (7) in Eq. (5) leads to:

$$\sum_{k=1}^{m-1} \left(e_c^k - e_c^{k-1} + ydisc_c^k \right) \quad \geq \quad \sum_{k=1}^{m-1} ydisc_c^k + y_c^{adj} \tag{10}$$

$$\Leftrightarrow \sum_{k=1}^{m-1} \left(e_c^k - e_c^{k-1} \right) \quad \geq \quad y_c^{adj} \tag{11}$$

$$\Leftrightarrow e_c^{m-1} \quad \geq \quad y_c^{adj} \tag{12}$$

With respect to Eq. (8) we have proven that the discrete congestion does not exceed the congestion of the continuous diffusion algorithm in the network. □

Theorem 3 *The maximal bound for the deviation of the discrete to the average load $\bar{w}$ on node n_i is the product of the maximal load $Smax$ of a task $p \in P(t)$ and the degree d_i of the node n_i:*

$$d_i * S_{max} > |\bar{w} - w_i^m| \tag{13}$$

Proof 3

$$With \qquad \bar{w} = w_i^0 - \sum_{c=1}^{d_i} \sum_{k=1}^{m-1} ycont_c^k \tag{14}$$

$$and \qquad w_i^m = w_i^0 - \sum_{c=1}^{d_i} \left[\sum_{k=1}^{m-1} ydisc_c^k + y_c^{adj} \right] \tag{15}$$

we compute the average and the discrete load after running the discrete diffusion algorithm. In Eq. (14) and (15), we accumulate the flow on all edges of one node and add it to the load of the node n_i.
We already mentioned the last adjustment step after the $m-1$ iterations. After this adjustment step, we require the final error e_c^m to be less than the maximal task size:

$$e_c^m < S_{max} \tag{16}$$

Inserting Eq. (14) and (15) in (13), leads to:

$$d_i * S_{max} > \left| w_i^0 - \sum_{c=1}^{d_i} \sum_{k=1}^{m-1} ycont_c^k - w_i^0 + \sum_{c=1}^{d_i} \left[\sum_{k=1}^{m-1} ydisc_c^k + y_c^{adj} \right] \right| \tag{17}$$

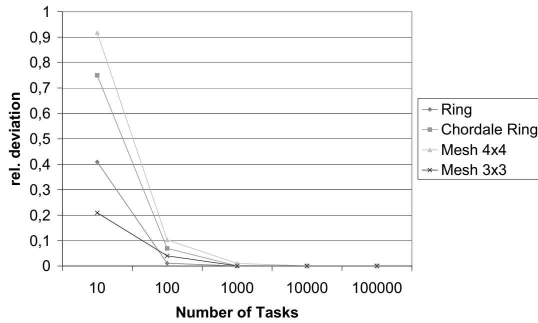

Fig. 3. Relative deviation between the optimal balanced and the real network over the number of tasks.

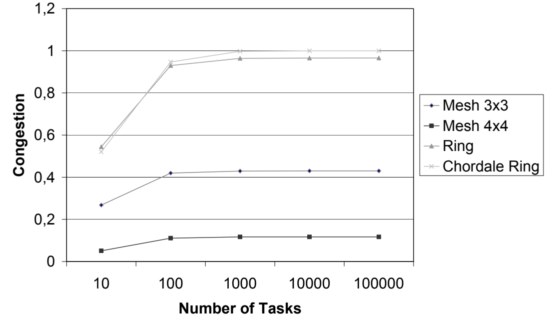

Fig. 4. Comparison of congestions caused by the discrete and the continuous diffusion algorithm.

Replacing $ydisc_c^k$ with the help of Eq. (3), results in:

$$d_i * S_{max} > \left| -\sum_{c=1}^{d_i}\sum_{k=1}^{m-1} ycont_c^k + \sum_{c=1}^{d_i}\left[\sum_{k=1}^{m-1}\left(ycont_c^k + e_c^{k-1} - e_c^k\right) + y_c^{adj}\right]\right| \quad (18)$$

$$\Leftrightarrow d_i * S_{max} > \left|\sum_{c=1}^{d_i}\left[\sum_{k=1}^{m-1}\left(e_c^{k-1} - e_c^k\right) + y_c^{adj}\right]\right| \quad (19)$$

$$\Leftrightarrow d_i * S_{max} > \left|\sum_{c=1}^{d_i} -e_c^{m-1} + y_c^{adj}\right| \quad (20)$$

with Eq. (4):

$$\Leftrightarrow d_i * S_{max} > \left|\sum_{c=1}^{d_i} -e_c^{m}\right| \quad (21)$$

According to Eq. (16) and (21), this assumption is correct. The maximal deviation between the continuous optimum and the discrete optimum of the load is therefore less than the maximal size of a task times the degree of a node, which is due to the nature of local iterative algorithms and thus, optimal in the discrete case. □

Note that all eigenvalues have to be computed prior to running the iterations.

V. Experiments

By experiment, we evaluated the discrete diffusion algorithm for different types of a network like meshes with 3x3 or 4x4 nodes, a ring and a chordal ring with 8 nodes.

In our experiment we use different networks with a varying number of tasks between 10 to 100000. The size of the tasks ranges from 1 to 100 and is randomly distributed over the number of tasks. In the beginning all tasks are mapped onto a single resource node, which is obviously a worst case scenario. The focus is set on the load error $|\bar{w} - w_i|$ and the congestion in the network.

In Figure 3, we present the arithmetic mean deviation v in the different networks over the number of tasks:

$$v = \frac{\sum_{i=1}^{|N|} |\bar{w} - w_i|}{|N|} \quad (22)$$

We can see two interesting properties: Firstly, the deviation in the mesh with 4x4 nodes is very high. This stems from the fact that more nodes exist than tasks (16 nodes, 10 tasks). Secondly, for an increasing number of tasks we approximate the continuous diffusion scheme.

In Figure 4, we illustrate the relation between the congestions c_R caused by the discrete diffusion and the continuous diffusion algorithm:

$$c_R = \frac{\sum_{i=1}^{n}\sum_{c=1}^{d_i} xcont_c}{\sum_{i=1}^{n}\sum_{c=1}^{d_i} xdisc_c} \quad (23)$$

If the relative congestion is less than 1 the discrete diffusion scheme performs better than its continuous counterpart, which is the case in all experiments.

Compared to the continuous diffusion algorithm, we have proven theoretically and shown practically that our discrete version is always better concerning the congestion. Moreover, we have theoretically deduced the upper bounds for the deviation of the optimal load distribution and the discrete load distribution. Compared to these theoretical bounds our algorithm produces better results in the presented examples than the theoretical bounds.

VI. Conclusions and Future Work

We have presented a first approach to solving the HW/SW partitioning problem online for embedded networked systems. In order to cope with faults and dynamic load over longer time intervals, an iterative optimization that finds optimal (re)partitions of tasks is applied. The idea is to increase the likelihood that future node and link failures may be compensated fast due to load reserves in the neighborhood of each node that may fail. Therefore, we developed a discrete diffusion algorithm based on continuous diffusion schemes and proved run-time optimality in linear number of steps. Thus, this paper presents theoretical and practical aspects to solve the HW/SW partitioning problem online.

References

[1] Douglas Chang, Malgorzata Marek-Sadowska, " Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs," *International Symposium on FPGAs*, 1998.

[2] G. Cybenko, " Dynamic Load Balancing for Distributed Memory Multiprocessors," *Journal of Parallel and Distributed Systems*, pp. 279-301, 1998.

[3] R.Dick, N. Jha, " CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems," *ICCAD'98*, 1998.

[4] R. Elsässer, A. Frommer, B. Monien, R. Preis, " Optimal and Alternating-Direction Loadbalancing Schemes," *Euro-Par 99, Parallel Processing*, 1999.

[5] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, R. Vemuri, " An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," *IPPS/SPDP Workshops*, 1998.

[6] C. Zhang, F. Vahid, R. Lysecky, " A Self-Tuning Cache Architecture for Embedded Systems," *DATE'04*, 2004.