OFF-LINE PLACEMENT OF TASKS ONTO RECONFIGURABLE HARDWARE CONSIDERING GEOMETRICAL TASK VARIANTS

Klaus Danne*, Sven Stühmeier
*University of Paderborn, Germany, *funded by DFG Graduate College 776*

Abstract: We consider off-line task placement onto reconfigurable hardware devices (RHDs), which are increasingly used in embedded systems. The tasks are modelled as three dimensional boxes given by their footprint times execution time which results into a three dimensional orthogonal packing problem. Unlike other approaches, we allow several alternative implementation variants for each task, which enables better placements. We apply modified heuristic methods from chip floorplanning to select and place the task variants. Our method computes a set of pareto placement solutions with the objectives to minimize the total execution time and the amount of required RHD area. We have evaluated the placement quality in first simulation experiments.

1. INTRODUCTION

Reconfigurable hardware devices (RHD), such as the prominent field-programmable gate array (FPGA) or coarse grain devises [1], are general-purpose devices that can be (re-)programmed after fabrication. SRAM-based FPGA variants can be reconfigured arbitrarily often, opening up the way to FPGA based computing. For a number of embedded applications, RHDs have been shown to outperform general-purpose processors, and even specialized processors [2]. Often the processing elements of RHDs are arranged as two dimensional arrays which can be reprogrammed (partially) during runtime. Therefore their resources can be reused for differed tasks over time. As modern devices have high densities, several tasks can be mapped onto the device at the same time, enabling true parallel execution. Combined, these features enable multitasking in space (e.g. parallel executed tasks) and time (e.g. sequentially executed tasks).

Placement and scheduling of task onto RHDs has attracted wide attention, e.g. in [3–5]. Surprisingly, most authors assume that the tasks

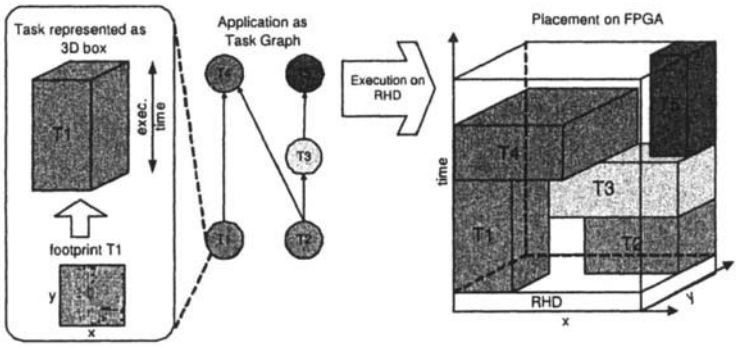

Figure 1. Placement of a task graph onto a RHD

have a fix rectangular shaped footprint given by *width* ×*length* and an unknown or fix execution time. The problem remains to place the tasks, which are represented by three dimensional objects *footprint* ×*execution time*, into a three dimensional container, given by the RHD array dimensions and the time dimension (see Figure 1).

In contrast to the model with fix tasks, we consider that several implementation variants for each task may exist, differing in the dimensions *width, length* and *execution time*. The problem is now given by selecting a proper variant for each task, assigning an position on the RHD and assigning a start time. The two objectives that should be optimized are the amount of required FPGA area and the total execution time of the task graph. Truly, the option to select variants for each task enlarges the design space, which enables new (potentially better) solutions. Also the problem gets more computational intensive, which makes optimal solving hopeless for a reasonable number of tasks. Therefore, we adopt heuristics form integrated circuit floorplanning. Specifically, we apply a bipartitioning method using slicing trees [6].

In Section 2 we formalize the our placement problem and defines valid solutions. Section 3 describes the applied placement method in detail. In Section 4 we report on first results and conclude the paper in Section 5.

2. PROBLEM MODELLING

We consider an application modeled as a directed acyclic task graph $G = (V, A)$, which consist of vertices V representing the task set $V = \{T_1, \ldots, T_n\}$ and directed arcs $A \subseteq V \times V$ where an arc $a_{i,k} = (T_i, T_k)$ represents an order constraints between the tasks T_i and T_k. For each task, a set of implementation variants exist $T_i = \{T_{i,1}, T_{i,2} \ldots\}$. Each variant $T_{i,k}$ is characterized by its footprint dimensions $x_{i,k} = x(T_{i,k})$ and $y_{i,k} = y(T_{i,k})$ which it occupies on the RHD when being executed, and by its execution time $t_{i,k} = t(T_{i,k})$. We assume, that the set of variants of each task T_i is a pareto set. I.e. for any given task variant

$T_{i,k} \in T_i$, no other task variant $T_{i,l}$ exists that has less or equal values in all three dimensions x, y and t than $T_{i,k}$.

Problem Definition. For a given task graph G, find a *selection function* $f : V \rightarrow \bigcup_{i=1}^{n} T_i$, which selects a variant for each task, and a *placement function* $p : V \rightarrow \mathbf{N}^3$, which assign an position for each task.

The placement function assigns x, y and t coordinates to each task by $p_x(T_i)$, $p_y(T_i)$ and $p_t(T_i)$. These coordinates define the starting time and the position onto the RHD of the front left vertex of the selected variant $T_{i,f(T_i)}$. Consequently, the placed variant occupies the RHD in x dimension from $p_x(T_i)$ to $p_x(T_i) + x(f(T_i))$, in the y dimension from $p_y(T_i)$ to $p_y(T_i) + y(f(T_i))$ and in the time dimension from $p_t(T_i)$ to $p_t(T_i) + t(f(T_i))$.

Valid solution. A valid solution for a graph G is a selection function f and a placement function p, such that the following predicates hold:

- f selects only existing task variants, i.e. $\forall T_i \in V : f(T_i) \in T_i$
- p places the task variants in such a way, that their three dimensional visualizations do not intersect (see Figure 1). This is the fact, iff all pairs of tasks are in series (non overlapping) either in the x, y or t dimension, which is expressed by equation 1.

$$\forall (T_i, T_k) \in V^2 : T_i \neq T_k \Rightarrow d^x_{i,k} \vee d^y_{i,k} \vee d^t_{i,k} \tag{1}$$

$$d^x_{i,k} = [p_x(T_i) > p_x(T_k) + x(f(T_k))] \vee [p_x(T_i) + x(f(T_i)) < p_x(T_k)]$$
$$d^y_{i,k} = [p_y(T_i) > p_y(T_k) + y(f(T_k))] \vee [p_y(T_i) + y(f(T_i)) < p_y(T_k)]$$
$$d^t_{i,k} = [p_t(T_i) > p_t(T_k) + t(f(T_k))] \vee [p_t(T_i) + t(f(T_i)) < p_t(T_k)]$$

- no precedence constraints are violated:

$$\forall a_{i,k} \in A : (p_t(T_i) + t(f(T_i)) \leq p_t(T_k)) \tag{2}$$

3. PLACEMENT METHOD

This section presents the method we use to find a good *selection* and *placement* of task variants of the input graph G. These is done in two phases. In the *partitioning phase*, a placement topology is created. We partition the three dimensional container given by the RHD array dimensions and the time dimension hierarchically into *rooms*, until one room for every task of V exist. This defines the relative position among the tasks, e.g T_1 will be placed left from T_2 in x dimension and in front of T_3 in y dimension. In the *sizing phase*, we compute the possible sizes of every room and finally the possible sizes of the entire container. We

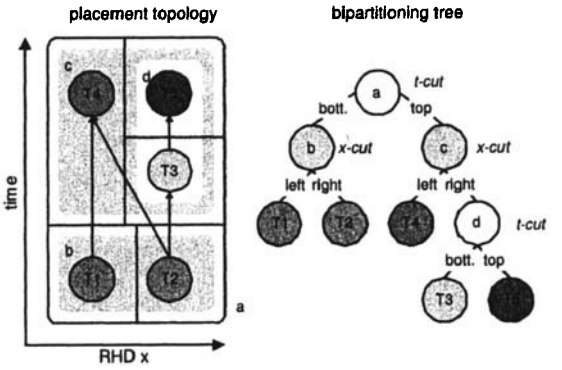

Figure 2. Placement topology and corresponding bipartitioning tree

chose the container size that minimizes our cost function and derive the variant selection f and the absolute task positions p.

3.1 Placement Topology by Bipartitioning

The partitioning phase creates a placement topology, that defines the relative position of every task. We restrict the placement topologies to topologies, which can be generated by a bipartitioning process. These are called *slicing placement topologies* (in the style of [6]). Figure 2 shows an example of a 2-dimensional slicing placement topology on the left side, and the corresponding bipartitioning tree on the right side. The initial placement is represented by an empty rectangle (room) labeled by a, which has the entire task set assigned $a = V$. The task set of room a is partitioned into two sub-rooms b and c by a horizontal cut and c is defined to be on top of b in the time dimension. Then, room b is cut into two sub-rooms by a vertical cut. This recursive cutting process stops, when an exclusive room for every task exists. The bipartitioning tree on the right side of Figure 2 fully defines the relative placement topology of the left side of Figure 2. Each tree node represents a room and its two direct child nodes represent the sub-rooms created by a cut. The leave nodes represent the final rooms, which hold only one task each. The labels at nodes define the cut *orientation* and the labels at the edges define the cut *order*. The orientation defines along which axis the room is sliced into sub-rooms and can be either vertical *(x-cut)* or horizontal *(t-cut)*. The order of the cut defines the order of the two sub-rooms in the corresponding dimension and is labeled with $\{left,right\}$ for x-cuts or $\{top,bottom\}$ for t-cuts respectively. Since in our placement problem we generally consider 3-dimensional objects, we define additional the *(y-cut)* orientation with the ordering labels $\{front,back\}$.

When the bipartitioning tree is created to define the relative placement for a given input task graph G, three heuristic decisions have to

be made for every cut. 1.) the node's task set has to be partitioned into the two subsets for the two child nodes. 2.) the orientation of the cut has to be defined. 3.) the relative order of the child nodes in the cut dimension has to be defined.

In the circuit layout, the partitioning into two subsets has usually the first goal to place heavily connected circuit blocks close together, while the second goal is generate sub-rooms of about the same size. Up to now, our model does not consider communication cost among the tasks and we only modeled precedence constraints which correspond to some communication. In future work, we will improve our model to consider communication costs and minimize these during the partitioning process. Until then, we make the partitioning only on basis of the size of the tasks in order to keep the sub-rooms of about the same size. Therefore, we define the average size of a task $\bar{S}(T_i)$ as the average over the product of the dimensions of all variants: $\bar{S}(T_i) = \sum_{k=1}^{m=|T_i|} \frac{x_{i,k} \times y_{i,k} \times t_{i,k}}{m}$. When the task set $V(a)$ of a room a should be partitioned into $V(b)$ and $V(c)$, we sort the tasks into a queue in order of decreasing average size. The first task of the queue is assigned to $V(b)$. After that, we keep assigning tasks from the queue to $V(c)$ until the size of $V(c)$ exceeds the size of $V(b)$. Then, we stop assigning tasks to $V(c)$ and start to assign tasks to $V(b)$ until the size of $V(b)$ exceeds the size of $V(c)$ and so on. The procedure terminates when the queue is empty and returns two partitions with about the same size.

The decision concerning the orientation and order of a cut can be made during the partitioning phase when the bipartitioning tree is created or can be made later during the sizing phase. We call that *oriented partitioning* and *unoriented partitioning* respectively. When these decisions are made in the fist phase, all nodes and edges of the tree get labeled with orientation and order labels respectively and the tree defines an *oriented ordered placement topology* like in the example in Figure 2. When these decisions are made in the sizing phase, the result of the partitioning phase is an unlabeled tree which defines an *unoriented placement topology.* Considering the example of Figure 2, an unoriented placement topology would define that e.g. task T_1 and T_2 have a common border, but it is neither defined in which dimension the tasks are side by side nor in which order.

Respecting precedence constraints. When precedence constrains among tasks have to be respected, the orientation and order of the cuts do matter. Let r_1 be a parent room that gets partitioned into r_2 and r_3. Obviously, an order constraints $a_{i,k} = (T_i, T_k)$ is satisfied, when room r_1 with $T_i, T_k \in r_1$ gets partitioned by a t-cut and r_2 with $T_i \in r_2$ becomes

the *bottom* room and r_3 with $T_k \in r_3$ the *top* room. The example in Figure 2 shows, that all dependency arcs among tasks are crossed by an t-cut. The precedence constrains modify our goal of the partitioning process. The first goal remains to slice a parent room r_1 into child rooms r_2, r_3 of about equal size. The second goal is, that their should be either no precedence edge being cut and we let the cut orientation *unoriented* or a maximum of precedence edges should be cut, all in the same direction (e.g from r_2 to r_3), and the cut is defined to be an *t-cut* (e.g. with r_2 being the bottom and r_3 the top room). Currently we use a simple heuristic strategy, where we archive an balanced initial bipartitioning as described above. If precedence edges in both directions have been cut, we modify the partitioning by moving tasks from one side to the other in order to find a partitioning with only edges being cut in one direction. In future work, we plan to use more goal oriented approaches like modified versions of the Kerninghan-Lin [5] and Fiduccia-Mattheyses heuristics that where adapted to directed graphs.

3.2 Sizing of the Placement Topology

The sizing phase takes the relative placement topology in form of bipartitioning tree of the first phase as input and computes an absolute position and size of every room, including the size of the container. Since in general there exist several variants for each task, there exist several variants for every room as well. To be able to compute the variants of a room resulting from the variants of its two sub-rooms, we define (similar to the sizing step functions used in circuit layout [6]) a sizing function for each task and room. In the two dimensional case, the sizing function of a task $s_{T_i} : \mathbf{R} \rightarrow \mathbf{R}$ is a monotonically decreasing step function of x, where $s_{T_i}(x)$ is the minimum of the execution times of all task variants that have a width smaller or equal to x (eq. 3). Figure 3(a) shows an example of a sizing function for the 2-dimensional case.

In the 3-dimensional case, $s_{T_i} : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ is a piecewise constant function, where $s_{T_i}(x, y)$ is the minimum of the execution times of all task variants with dimensions less or equal then x and y (eq. 4).

$$s_{T_i}(x) = \min_{\{T_{i,k} \in T_i | x_{i,k} \leq x\}} t_{i,k} \tag{3}$$

$$s_{T_i}(x, y) = \min_{\{T_{i,k} \in T_i | x_{i,k} \leq x \wedge y_{i,k} \leq y\}} t_{i,k} \tag{4}$$

The sizing functions of any other room can be computed based on the sizing functions of its two child nodes and as consequents the sizing function of the root node can be computed bottom up. The sizing of oriented and unoriented placement topologies have to be distinguished.

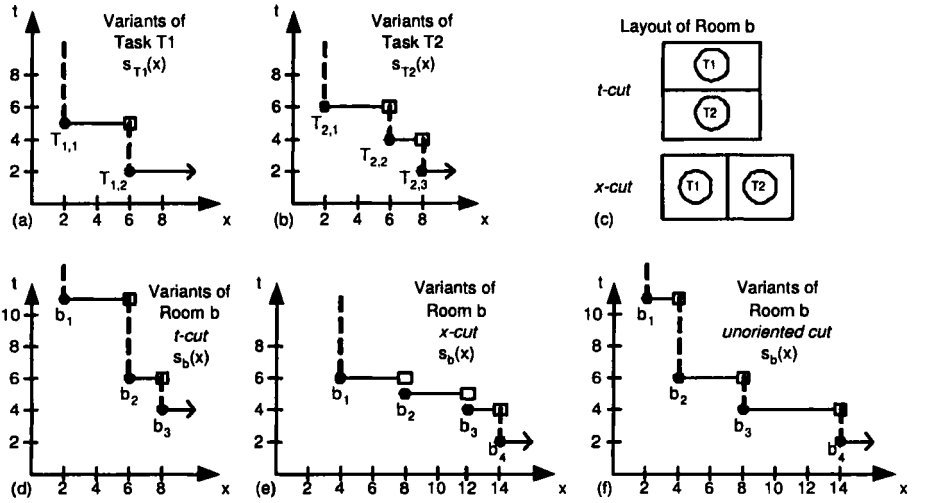

Figure 3. Sizing function tasks T_1 (a) and T_2 (b), possible room layouts (c) and the resulting sizing function s_b of room b for an *t-cut* (d), an *x-cut* (e) and an *unoriented-cut* (f) respectively.

Sizing oriented ordered placement topologies. In the case of oriented ordered placement topologies, the cut orientation of a room in its two sub-rooms is already defined. Let $r1$ be a room that is sliced by a *t-cut* into $r2$ and $r3$, than we obtain the sizing function of r_1 by summing the sizing functions of its child nodes (eq. 5). Figure 3(d) shows the sizing function resulting from adding the sizing function of Figure 3(a) and (b). In case of an *x-cut*, the inverse of the sizing function of $r1$ is the sum of the inverse sizing function of $r2$ and the inverse sizing function of $r3$ (eq. 6). This is illustrated in Figure 3(e).

$$s_{r1} = s_{r2} + s_{r3} \quad \text{for t-cut} \tag{5}$$

$$s_{r1}^{-1} = s_{r2}^{-1} + s_{r3}^{-1} \quad \text{for x-cut} \tag{6}$$

Note, that in the algorithmic implementation, the sizing function of each task is stored only as its list of variants sorted by x. In the same way the sizing function of any room is stored as a variant list which represents the pareto points of the continuous sizing function. When two sizing functions have to be added as result of a *t-cut*, both lists a processed by a linear scan in x direction and combined to new variants of the parent room. The same is done in t direction in case of an *x-cut* respectively.

In the 3-dimensional case, where the variants are triples, the inverse of a sizing function is not well defined. In case of an *t-cut* we still obtain the sizing function of a room by summing up the sizing functions of the child nodes ($s_{r1}(x,y) = s_{r2}(x,y) + s_{r3}(x,y)$). In case of an *x-cut*,

the sizing functions of the child nodes $s_{r2}(x,y)$ and $s_{r2}(x,y)$ are flipped along the $x = t$ plain, summed up and the result is again flipped along the $x = t$ plain to obtain $s_{r1}(x,y)$. The same is done with the $y = t$ plain for an *y-cut* respectively.

In our algorithmic implementation we store each sizing function as a list of triples $(t_{i,k}, x_{i,k}, y_{i,k})$. Since no total order is defined on the variants in the 3-dimensional case, we combine all variants of the first child with all of the second child. In a second step, we remove all non pareto optimal variants.

The first part of the sizing phase is finished, when the sizing function of the root node of the bipartitioning tree has been computed.

Sizing unoriented placement topologies. When the orientation of the cuts is left undefined, the placement topology describes a larger set of actual placements then in the oriented case. When combining the variants of two child rooms to the variants of the parent room, we have to consider all possible cut orientations. Therefore, we compute the sizing functions of a room r for a t-cut,x-cut and y-cut as described for the oriented placement topologies. Then, the sizing function of r for the unoriented case $s_r^{unor.-cut}$ is defined as the minimum over its sizing functions for every possible cut orientation (eq. 7). Figure 3(e) shows an example of a sizing function for an unoriented cut in the 2-dimensional case.

$$s_r^{unor.-cut}(x,y) = \min\left(s_r^{t-cut}(x,y), s_r^{x-cut}(x,y), s_r^{y-cut}(x,y)\right) \quad (7)$$

In the algorithmic implementation, we compute the minimum by joining the three variant lists for the different cut orientations of r and remove the non pareto variants. The unoriented placement topology clearly leads to better variants of a room and therefore improves the placement quality. This comes at the cost of computational complexity, since the variant lists for the rooms contain more elements.

3.3 Deriving Selection and Placement Function

When the possible variants of the container have been computed, that is the sizing function of the root node, we select the variant that minimizes our cost function, e.g. the variant with smallest $t \times x$, or $t \times x \times y$ in the 3-dimensional case respectively. This immediately determines the selection of the variants of the two child nodes *child*1 and *child*2, since we store which variants of *child*1 and *child*2 are responsible for which variant of *root* during the recursive sizing process described above. Therefore, when we reach the leafs of the tree in a top down fashion, we obtain the selection function $f(T_i)$ for every task.

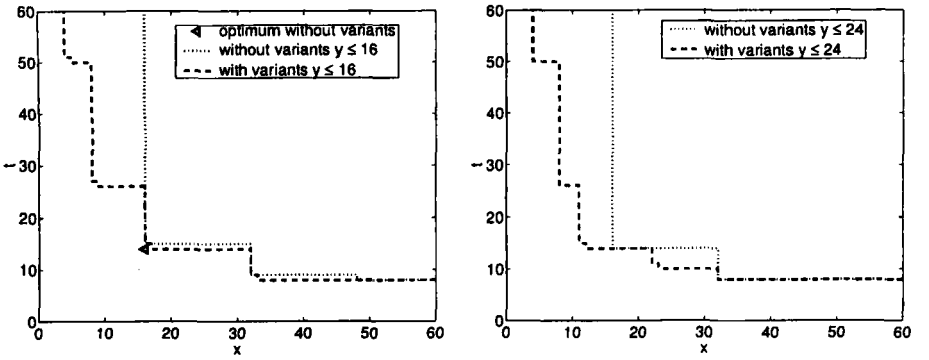

Figure 4. 2-dim. projection of the computed sizing function for the DE benchmark with and without additional task variants.

To complete the solution, the placement function $p(T_i)$ has to be defined. Therefore, we define the position of each room (each node of the tree) in a top down fashion, until the tasks positions are defined. To be able to do so, the bipartitioning tree has to be oriented and ordered. As mentioned before, the cut orientation is defined either during the partitioning phase or in the sizing phase. In our current model, the cut order does not influence the quality of the placement and is chosen randomly.

The position of the container (root node) is initialized to $p_t(root) = 0, p_x(root) = 0, p_y(root) = 0$. The position of each child node with order *bottom, left* or *front* is always equal to the position of its parent node and independent of the cut orientation. The position of each *top, right* or *back* child node is equal to the position of the parent node, except in the cut dimension: In cut dimension the size of the neighbor node is added to the position of the parent node. E.g. the position of room d of Figure 2 would be $p_t(d) = p_t(c)$ and $p_x(d) = p_x(c) + x(f(T_4))$, since room c is sliced by an x-cut.

4. EXPERIMENTAL RESULTS

As a first test, we applied our method to the DE benchmark application used in [3]. The application is an task graph with 11 nodes consisting of multiply and ALU-operations, which numerically solves a differential equation. In [3], every multiplier was modeled by an 16 times 16 rectangle of FPGA cells using 2 clock cycles ($x_i = 16, y_i = 16, t_i = 2$) and the ALU operations where assumed to occupy $x_k = 16 \times y_k = 1$ cells taking $t_i = 1$ clock cycle. Using the unmodified task graph, our algorithm computed a sizing function consisting of 15 different variants. After introducing 14 different variants for the multiplier tasks as well as for the adder task, our method computed 118 different variants for the overall design. Figure 4 shows both sizing functions as two-dimensional

projections, where y is given as parameter for the cases $y \leq 16$ and $y \leq 24$. It can be seen, that the sizing function of the task set with variants introduces more and better design alternatives. The left figure shows also an optimal placement $t = 14, x = y = 16$ taken from [3], which our method came close to, but missed it by one unit in either x or t dimension. We plan to evaluate the quality of the method on a larger set of benchmark applications in future work.

5. CONCLUSION

In contrast to previous work in the field of RHD task placement, we presented a model that allows several alternative implementations for each task and therefore enables *better* solutions for the placement problem . These variants can be archived by synthesizing the tasks with several footprint constraints and by providing different register transfer level descriptions (e.g. parallel multiplier, bit serial multiplier) for each task. To attack the problem, we adopted algorithms used in floorplanning of integrated circuits, which where extended to the three-dimensional case to consider RHD resource sharing over time. The advantages of our method are, that it 1) considers an enlarged design space, 2) can be applied to task graphs of large size and 3) that not only one solution, but a set of pareto optimal solutions is computed at once. Our first experimental results have shown the applicability of our approach. Improvements and further evaluation on larger benchmarks will be subject of our further work.

REFERENCES

[1] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[2] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[3] S. P. Fekete, E. Köhler, and J. Teich. Optimale FPGA module placement with temporal precedence constraints. In *Proc. DATE 2001, Design, Automation and Test in Europe*, pages 658–665, Munich, Germany, March 13-16 2001. IEEE Computer Society Press.

[4] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, pages 68–83, March 2000.

[5] Christophe Bobda. *Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement.* Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, 2003. Euro 35,-, ISBN 3-935433-37-9.

[6] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley, 1990.