Offline and Online Aspects of Defragmenting the Module Layout of a Partially Reconfigurable Device

Sándor P. Fekete, Jan C. van der Veen, Ali Ahmadinia, Diana Göhringer, Mateusz Majer, and Jürgen Teich, *Senior Member, IEEE*

***Abstract*—Modern generations of field-programmable gate arrays (FPGAs) allow for partial reconfiguration. In an online context, where the sequence of modules to be loaded on the FPGA is unknown beforehand, repeated insertion and deletion of modules leads to progressive fragmentation of the available space, making defragmentation an important issue. We address this problem by proposing an online and an offline component for the defragmentation of the available space. We consider defragmenting the module layout on a reconfigurable device. This corresponds to solving a 2-D strip packing problem. Problems of this type are NP-hard in the strong sense, and previous algorithmic results are rather limited. Based on a graph-theoretic characterization of feasible packings, we develop a method that can solve 2-D defragmentation instances of practical size to optimality. Our approach is validated for a set of benchmark instances. We also discuss a simple strategy for dealing with online scenarios, called "least-interference fit" (LIF); we give a number of analytic results that allow a comparison of LIF with the best offline solution, and demonstrate that it works well on benchmark instances of moderate size.**

***Index Terms*—Defragmentation, exact algorithms, NP-hard problems, partial reconfiguration, reconfigurable computing, 2-D packing.**

I. Introduction

ONE OF THE cutting-edge aspects of modern reconfigurable computing is the possibility of *partial* reconfiguration of a device. Ideally, a new module can be placed on a reconfigurable chip without interfering with the processing of other running tasks. (See the end of this subsection for some practical restrictions in current generations of field-programmable gate arrays (FPGAs).) Clearly, this approach has many advantages over a full reconfiguration of the whole chip. Predominantly, it lessens the bottleneck of reconfigurable computing: reconfiguration time.

On the other hand, partial reconfiguration introduces a new complexity: management of the free space on the FPGA. In the 2-D model this is an NP-hard optimization problem. There has been a considerable amount of work to solve this problem computationally. However, due to its computational complexity most recent work has focused on the online setting or on the 1-D area model (see [2] for a recent survey).

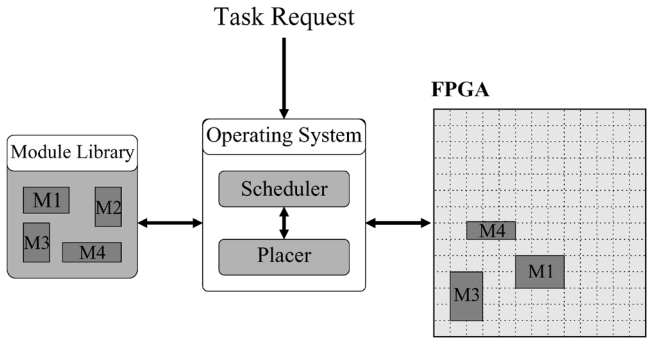


Fig. 1. Schematic overview of an operating system for reconfigurable computers. Relocatable, presynthesized modules that are constrained to a rectangular layout are stored in a module library. As requests for tasks arrive, a module capable of running the task is selected, scheduled, and eventually placed on the FPGA.

Management of free space and scheduling of arriving tasks are the core components of an operating system for reconfigurable platforms (see Fig. 1). In all previous work, these components use simple online strategies for the placement problem. The use of these strategies leads to fragmentation of the free space, as modules are placed on and removed from the chip area. This leads to situations where a new module has to be rejected by the placer because there is no free rectangle that could accommodate the new module, even though the total free space available would be more than sufficient (see [3] for further discussion).

In this paper, we propose a different placer module. Instead of just relying on online strategies our placer has an additional offline component: the *defragmenter*. Consider the following scenario. A car is equipped with a multimedia device that contains a partially reconfigurable FPGA. Let this multimedia device be responsible for audio, video, telephony, and WLAN. While the car is in use, the device is busy and tasks must be scheduled and modules must be placed as they arrive. However, the recurring idle times of the car (e.g., over night) can be utilized to optimally defragment the FPGA chip area.

This optimal defragmentation has two goals: one is to maximize the available contiguous free space. and the other comes from the FPGA device we use. The current XILINX Virtex-II series does not admit full 2-D partial reconfiguration [4]. Instead, configuration can only be performed columnwise. While

Manuscript received May 10, 2006; revised June 28, 2007; accepted August 31, 2007. First published July 25, 2008; last published August 20, 2008 (projected). This research was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the project "ReCoNodes," under Grant FE407/8-1, Grant FE407/8-2, Grant TE163/11-1, and Grant TE163/11-2. A subset of the results of this paper appeared as a Distinguished Paper in the Proceedings of ERSA'05 [1].

S. P. Fekete and J. C. van der Veen are with the Algorithms Group, Department of Computer Science, Braunschweig University of Technology, D-38106 Braunschweig, Germany (e-mail: s.fekete@tu-bs.de; j.van-der-veen@tu-bs.de).

A. Ahmadinia, D. Göhringer, M. Majer, and J. Teich are with the Department of Computer Science 12, University of Erlangen-Nuremberg, D-91058 Erlangen, Germany (e-mail: ahmadinia@cs.fau.de; goehringer@cs.fau.de; majer@cs.fau.de; teich@cs.fau.de).

Digital Object Identifier 10.1109/TVLSI.2008.2000677

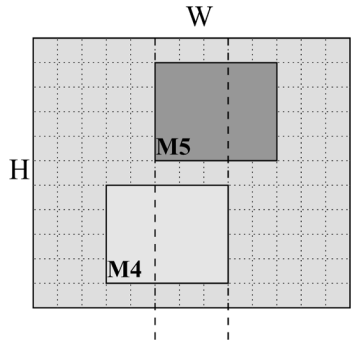


Fig. 2. FPGA of width $W = 13$ and height $H = 11$. Assume that module M4 of width $w_4 = 5$ and height $h_4 = 4$ is located at position (3, 1). If module M5 of same width and height is placed at position (5, 6) the resulting overlap is three columns as indicated by the dashed lines. Consequently, M4 is interrupted for $3c$ time units.

a column is reconfigured, all other modules that use this column have to be stopped, because reconfiguration interferes with the running tasks in a nontrivial way; see Fig. 2. So the other goal of the offline defragmenter is to free as many columns as possible. This way the next modules placed by an online placer will not interfere with other modules.

The rest of this paper is organized as follows. In Section II, we describe our FPGA model and conclude that the offline optimization problem that has to be solved is the 2-D strip packing problem. In Sections III and IV, we describe our algorithm for solving this problem to optimality, followed by a discussion of online scenarios in Section VI. Then we will report on computational results. In our conclusion, we hint at possible extensions of our model.

II. Column-Oriented Cost Function

Due to its widespread use, our device model closely resembles that of a Xilinx Virtex-II FPGA. In our model, the FPGA consists of a certain number of reconfigurable units called *configurable logic blocks* (CLBs). These CLBs are organized in W columns and H rows. There is no way to reconfigure CLBs individually: Reconfiguration takes place on the column level. We assume that it takes c units of time to configure one column of CLBs.

On this FPGA, we execute a certain set of tasks $T = \{t_1, t_2, \ldots\}$. In an offline setting one could assume that for each task an arrival time is known in advance. When focusing on the cost of reconfiguration, we are only interested in the order in which modules arrive, so we may simply assume that a given set of modules arrives in some permutation. As we will see later (in Section VI), the permutation itself does not matter when considering the optimal offline solution.

Tasks can be executed in hardware or in software. We assume that for each task there is at least one hardware or software module. A hardware module is a relocatable presynthesized digital circuit that has been constrained to a rectangular area. In the following, the dimensions of a module are given by (w_j, h_j), where w_j and h_j denote the width and the height of the jth module. As a consequence, placing module j on the FPGA takes time cw_j. A software module is a precompiled executable that can be executed, e.g., on a soft-core intellectual property (IP) such as the MicroBlaze soft-cores for Xilinx devices. For ease of notation, we assume that a software module j requires the width and height of its processor IP core. The set of all modules is given by $M = \{m_1, m_2, \ldots\}$, including possible processor cores.

Currently, communication between modules is still an issue. However, as chip size and complexity increases, circuit as well as packet-based on-chip communication networks, such as DyNoC [5] become more and more realistic. Here, we assume the availability of a fine-grained underlying communication infrastructure supporting intermodule communication requests.

In an offline setting we simultaneously seek for the following.

- *A feasible schedule for the tasks.* In other words, each task i is assigned a starting time s_i.
- *An assignment $m : T \to M$ of tasks to modules.* By $m(t_i)$ we denote the module task on which t_i will be executed.
- *A configuration schedule for the modules.* Each module j is assigned a reconfiguration time c_j. Of course configuration and starting time are related through $s_i \geq c_{m(t_i)} + cw_{m(t_i)}$.
- *A feasible placement of the modules on the FPGA.* For each module j its location $x_i \in [0, W - w_i)$ and $y_i \in [0, H - h_i)$ has to be determined.

Among all feasible solutions we select one that minimizes the *makespan*, i.e., the completion time of the last task. This alone is an NP-hard optimization problem, as it contains 2-D packing as a subproblem. At the same time, this problem is closely related to scheduling problems. (See [6] for an overview of classical "1-D" scheduling problems.)

In the 2-D placement model, column-wise reconfiguration has the drawback that reconfiguring a column of the FPGA affects all modules using this column in a nontrivial way. In our model, we assume that the reconfiguration of one column interrupts all modules using this column for the reconfiguration time c. Therefore, a task running on a module j is interrupted for $c|[x_j, x_j + w_j) \cap [x, x + w_i)|$ time units, if module i is placed starting at column x. Some of the modules might get harmed by the reconfiguration process (i.e., modules containing shift registers). These then need to be restarted, leading to an additional increase in total running time.

Experimental evidence suggests that a placement strategy should take this interference into account. This suggests a simple heuristic called *least interference fit* (LIF): new modules are placed in consecutive columns that are used by as few other modules as possible. Just like other heuristics, LIF can be used in both offline and online scenarios. As we showed in [7], LIF seems to perform better than other simple heuristics based on bin packing. In Section V, we compare LIF with optimal solutions.

At the same time, there are the following two other issues that any strategy faces.

1) *Free Space Fragmentation:* Even though the free space available on the FPGA would allow executing a task on a hardware module (resulting in better quality and/or faster execution), the largest free space fragment available may not be able to accommodate the respective module.
2) *Interference:* Even though respecting the number of interrupted modules, LIF still has to interrupt modules in the long run.

In this paper, we propose a strategy that can increase the long-term quality of any online strategy. As described previously, our scenario gives rise to times where the system is rather busy. On the other hand, there also are times when the system is more or less offline or unused. These are times when the FPGA could be defragmented. By defragmentation we mean giving up modules that have a low usage count and then relocating all other modules so that a maximal number of columns is unused. This increases the effectiveness of online strategies like LIF.

Defragmentation as described in the previous paragraph can be regarded as the 2-D strip packing problem. In Section III, we will take a closer look at this classic NP-complete optimization problem. As it turns out, for currently relevant numbers of modules, optimal placements can still be computed, using a cutting-edge algorithm for higher-dimensional packing.

III. 2-D Strip Packing

Packing rectangles into a container arises in many industries, whenever steel, glass, wood, or textile materials are to be cut, but it also occurs in less obvious contexts, such as machine scheduling or optimizing the layout of advertisements in newspapers. The 3-D problem is important for practical applications such as container loading or scheduling with partitionable resources. For many of these problems, objects must be positioned with a fixed orientation; this requirement also arises when configuring modules on a chip area.

Different types of objective functions for multidimensional packing problems have been considered. The *strip packing problem* (SPP) is to minimize the width W of a strip of fixed height H such that all rectangles fit into a rectangle of size $W \times H$. The *orthogonal knapsack problem* (OKP) requires selecting a most valuable subset S from a given set of rectangles, such that S can be packed into the large rectangle. The *orthogonal bin packing problem* (OBPP) considers the scenario in which a supply of containers of a given size is given and the objective is to minimize the number of containers that are needed for packing a set of boxes.

Crucial for all those optimization problems is the corresponding decision problem. The *orthogonal packing problem* (OPP) is to decide whether a given set of rectangles can be placed within a given rectangle of size $W \times H$. As all of the previous problems can be generalized to arbitrary dimensions, we denote by SPP-d, OKP-d, OBPP-d, and OPP-d the strip-packing problem, the orthogonal knapsack problem, the orthogonal bin packing problem, and the orthogonal packing problem, respectively, in d-dimensions (e.g., when considering scheduling problems on an FPGA implies considering two space and one time dimension, yielding $d = 3$). Being a generalization of the 1-D problem 3-Partition, the OKP-d is NP-complete in the strict sense, and so the corresponding optimization problems are NP-hard [8].

Dealing with an NP-hard problem (often dubbed "intractable") does not mean that it is impossible to find provably optimal solutions. While the time for this task may be quite long in the worst case, a good understanding of the underlying mathematical structure may allow it to find an optimal solution (and prove its optimality) in reasonable time for a large number of instances. A good example of this type can be found in [9], where the exact solution of a 120-city instance of the Traveling Salesman Problem is described. In the meantime, benchmark instances of size up to 13 509 and 15 112 cities have been solved to optimality [10], showing that the right mathematical tools and sufficient computing power may combine to explore search spaces of tremendous size. In this sense, "intractable" problems may turn out to be quite tractable.

Higher-dimensional packing problems have been considered by a great number of authors, but only few of them have dealt with the exact solution of general 2-D problems. See [11] and [12] for an overview. It should be stressed that unlike 1-D packing problems, higher-dimensional packing problems allow no straightforward formulation as integer programs. After placing one box in a container, the remaining feasible space will in general not be convex. Moreover, checking whether a given set of boxes fits into a particular container is trivial in 1-D space, but NP-hard in higher dimensions.

Nevertheless, attempts have been made to use standard approaches of mathematical programming. Beasley [13] and Hadjiconstantinou and Christofides [14] have used a discretization of the available positions to an underlying grid to get a 0–1 program with a pseudopolynomial number of variables and constraints. Not surprisingly, this approach becomes impractical beyond instances of rather moderate size.

To our knowledge there is only one work that tries to solve SPP to optimality. In [15], Martello *et al.* derive improved lower and upper bounds for the 2-D strip-packing problem. These bounds are based on a continuous relaxation of the 1-D contiguous bin-packing problem (1CBP). These bounds are used in a branch-and-bound type algorithm to solve 27 benchmark instances from the literature.

In [11], [12], and [16]–[18], a different approach to characterizing feasible packings and constructing optimal solutions is described. A graph-theoretic characterization of the relative position of the boxes in a feasible packing (by so-called *packing classes*) is used, representing d-dimensional packings by a d-tuple of interval graphs (called *component graphs*) that satisfy two extra conditions. This factors out a great deal of symmetries between different feasible packings, it allows to make use of a number of elegant graph-theoretic tools, and it reduces the geometric problem to a purely combinatorial one without using brute-force methods like introducing an underlying coordinate grid. Combined with good heuristics for dismissing infeasible sets of boxes [19], [20], a tree search for constructing feasible packings was developed. This exact algorithm has been implemented; it outperforms previous methods by a clear margin. This approach has been extended to strip-packing problems in the presence of order constraints; see [21]. (Note that in [21], the emphasis is on the mathematical aspects of dealing with order constraints, not on solving pure strip-packing instances efficiently, as is the case in this paper.)

For the benefit of the reader, a concise description of this approach is contained in Section IV.

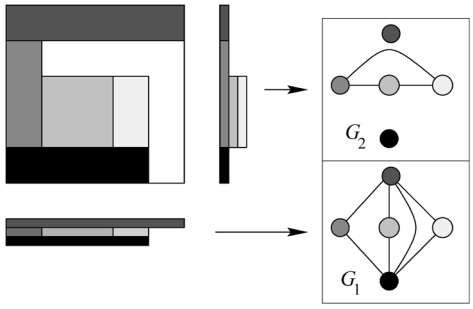


Fig. 3. Projections of the boxes onto the coordinate axes define interval graphs (here in 2-D: $\boldsymbol{G_1}$ and $\boldsymbol{G_2}$).

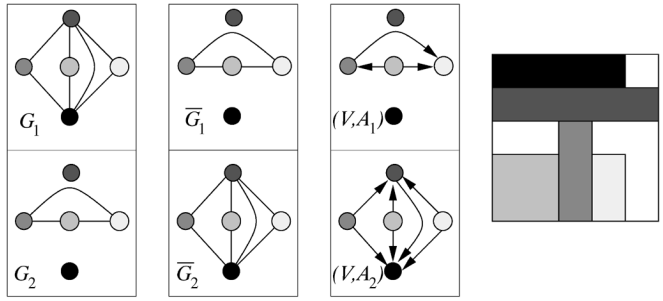


Fig. 4. (a) 2-D packing class. (b) The corresponding comparability graphs. (c) The transitive orientations. (d) A feasible packing corresponding to the orientation.

IV. Solving Unconstrained Orthogonal Packing Problems

A. General Framework

If we have an efficient method for solving OPPs, we can also solve SPPs by using a binary search. However, deciding the existence of a feasible packing is a hard problem in higher dimensions, and proposed methods suggested by other authors [13], [14] have been of limited success.

Our framework uses a combination of different approaches to overcome the following problems:

1) try to disprove the existence of a packing by classes of lower bounds on the necessary size;
2) in case of failure, try to find a feasible packing by using fast heuristics;
3) if the existence of a packing is still unsettled, start an enumeration scheme in form of a branch-and-bound tree search.

By developing good new bounds for the first stage, we have been able to achieve a considerable reduction of the number of cases where a tree search needs to be performed. (Mathematical details for this step are described in [16], [19], and [20].) However, it is clear that the efficiency of the third stage is crucial for the overall running time when considering difficult problems. Using a purely geometric enumeration scheme for this step by trying to build a partial arrangement of boxes is easily seen to be immensely time-consuming. In the following, we describe a purely combinatorial characterization of feasible packings that allows to perform this step more efficiently.

B. Packing Classes

Consider a feasible packing in d-dimensional space, and project the boxes onto the d coordinate axes. This converts the one d-dimensional arrangement into d 1-D ones (see Fig. 3 for an example in $d = 2$). By disregarding the exact coordinates of the resulting intervals in direction i and only considering their intersection properties, we get the *component graph* $G_i = (V, E_i)$. Two boxes u and v are connected by an edge in G_i, iff their projected intervals in direction x_i have a non-empty intersection. By definition, these graphs are *interval graphs*. This class of graphs has been studied intensively in graph theory (see [22] and [23]), and it has a number of very useful algorithmic properties.

Considering sets of d component graphs G_i instead of complicated geometric arrangements has some clear advantages (algorithmic implications for our specific purposes are discussed further down). It is not hard to check that the following three conditions must be satisfied by all d-tuples of graphs G_i that are constructed from a feasible packing.

C1: G_i is an interval graph, $\forall i \in \{1, \ldots, d\}$.

C2: Any independent set S of G_i is i-admissible, $\forall i \in \{1, \ldots, d\}$, i.e., $w_i(S) = \sum_{v \in S} w_i(v) \leq h_i$, because all boxes in S must fit into the container in the ith dimension.

C3: $\cap_{i=1}^{d} E_i = \emptyset$. In other words, there must be at least one dimension in which the corresponding boxes do not overlap.

A d-tuple of component graphs satisfying these necessary conditions is called a *packing class*. The remarkable property (proven in [12] and [24]) is that these three conditions are also sufficient for the existence of a feasible packing.

Theorem 4.1 (Fekete, Schepers): A set of boxes allows a feasible packing, iff there is a packing class, i.e., a d-tuple of graphs $G_i = (V, E_i)$ that satisfies the conditions *C1*, *C2*, and *C3*.

This result allows us to consider only packing classes in order to decide the existence of a feasible packing, and to disregard most of the geometric information. See Fig. 4 to see how a packing class gives rise to a feasible packing; note that this packing is not identical to the one in Fig. 3. (In fact, there are many possible packings for a packing class, see Section IV-C and Fig. 4.)

C. Solving OPPs

Our search procedure works on packing classes, i.e., d-tuples of component graphs with the properties C1, C2, and C3. Because each packing class represents not only a single packing but a whole family of equivalent packings, we are effectively dealing with more than one possible candidate for an optimal packing at a time. (The reader may check for the example in Fig. 3 that there are 36 different feasible packings that correspond to the same packing class; see Fig. 5.)

For finding an optimal packing, we use a branch-and-bound approach. The search tree is traversed by depth first search, see [17] and [24] for details. Branching is done by fixing an edge $\{b, c\} \in E_i$ or $\{b, c\} \notin E_i$. After each branching step, it is checked whether one of the three conditions C1, C2, and C3 is violated; furthermore, it is checked, whether a violation can only be avoided by fixing further edges. Testing for two of the conditions C1–C3 is easy: enforcing C3 is obvious; property C2 is hereditary, so adding edges to E_i later will keep it satisfied.

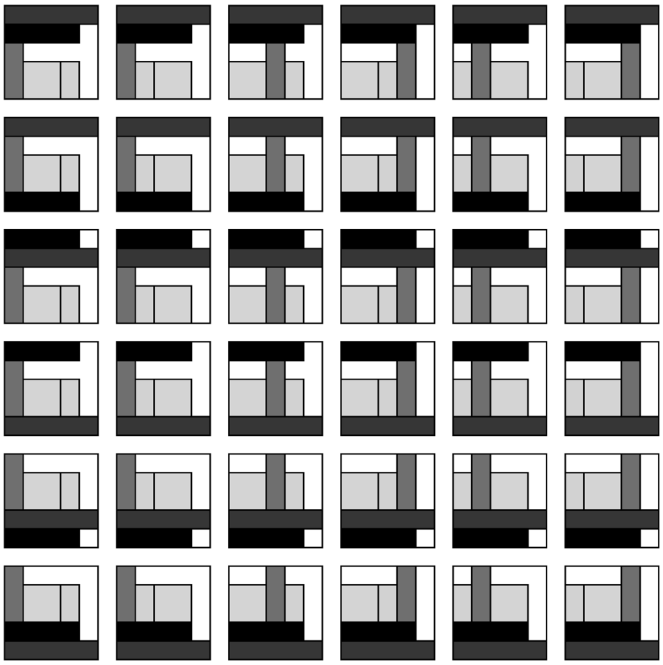


Fig. 5. All shown 36 packings correspond to the component graphs G_1 and G_2 that are shown in Fig. 3.

(Note that computing maximum weighted cliques on comparability graphs can be done efficiently, see [22].) In order to ensure that property C1 is not violated, we use some graph-theoretic characterizations of interval graphs and comparability graphs. These characterizations are based on two forbidden substructures (again, see [22] for details; the first condition is based on the classical characterizations by [25] and [26]: a graph is an interval graph *iff* its complement has a transitive orientation, and it does not contain any induced chordless cycle of length 4.) In particular, the following configurations have to be avoided:

G1: induced chordless cycles of length 4 in E_i;

G2: so-called 2-chordless odd cycles in the set $\overline{E_i}$ of edges excluded from E_i (see [17] and [22] for details);

G3: infeasible stable sets in E_i.

Each time we detect such a fixed subgraph, we can abandon the search on this node. Furthermore, if we detect a fixed subgraph, except for one unfixed edge, we can fix this edge, such that the forbidden subgraph is avoided.

Our experience shows that in the considered examples these conditions are already useful when only small subsets of edges have been fixed, because by excluding small sub-configurations, like induced chordless cycles of length 4, each branching step triggers a cascade of more fixed edges.

V. Defragmentation Approach and Computational Results

We have used our implementation for the OPP (as described in the previous section) as a building block for our defragmentation algorithm shown in Fig. 6 and described in the following. In order to allow our optimization algorithm to make use of computing devices such as a MicroBlaze core, we have used very simple lower and upper bounds to restrict the search. Suppose I denotes the set of indices of the modules currently present on the FPGA; then a lower bound for the number of columns W_L used by all modules after defragmentation is given by

$$W_L = \left\lceil \frac{\sum_{i \in I} w_i h_i}{H} \right\rceil.$$

The upper bound is computed as the minimum of the three shelf-packing heuristics next-fit-decreasing, first-fit-decreasing and best-fit-decreasing [27]. These heuristics partition the strip into shelves. A new shelf of height h_j is created if there is no shelf in which the module j can be placed. If the module can be placed in more than one shelf the shelf is picked according to the next-fit, first-fit, or best-fit strategy, respectively.

Based on these bounds the defragmentation algorithm performs a binary search until an optimal solution is found. The algorithm is outlined in Fig. 6.

We have benchmarked our code against a set of ten instances. See Figs. 7 and 8 for an illustration of Scenario A, and Figs. 9

```
DefragmentModuleLayout()
1  LB ← CalculateLowerBound()
2  UB ← CalculateUpperBound()
3  while LB ≠ UB do
4        W ← LB + ⌊(LB+UB)/2⌋
5        if SolveOPP(W) then
6              LB ← W
7        else
8              UB ← W
```

Fig. 6. Binary search algorithm for determining an optimal module layout. In this algorithm, the OPP as described in Section IV is solved repeatedly to determine if all modules fit in a strip of width W. This search is iterated until an optimal solution is found.

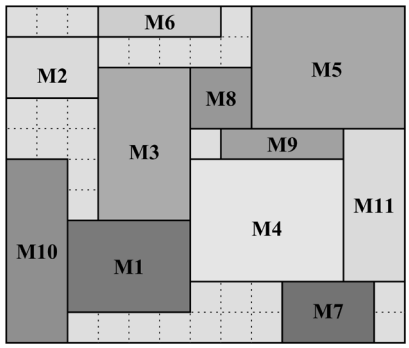


Fig. 7. FPGA before defragmentation. Even though the remaining free space is 30 reconfigurable units (CLBs), the maximal free rectangle of dimension 7 × 1 has only 7 CLBs. Note that there is no free column.

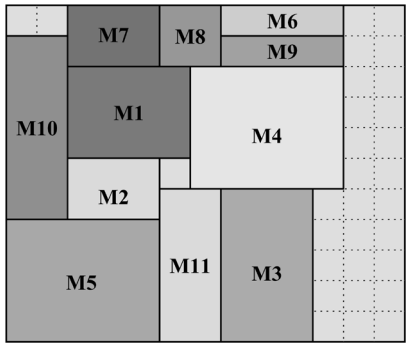


Fig. 8. Same FPGA as in Fig. 7 after defragmentation. The remaining free space is 30 reconfigurable units (CLBs). Now the maximal free rectangle is of dimension 2 × 11 has 22 CLBs. The number of free columns is 2.

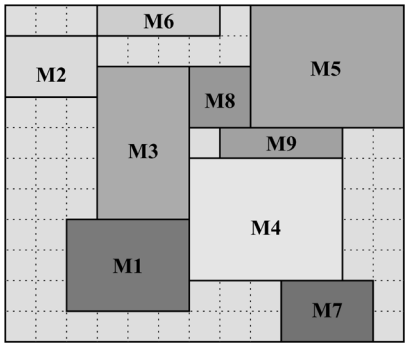


Fig. 9. Same FPGA as in Fig. 7. Modules M10 and M11 have been removed in this instance, e.g., because of a low usage count. The remaining free space is now 52 CLBs. The largest free rectangle has dimension 2×8 and 16 CLBs. There still is no free column.

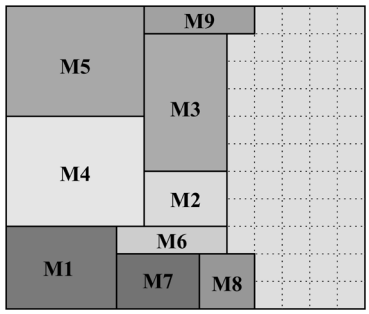


Fig. 10. Same FPGA as in Fig. 8 after successful defragmentation. The free space of 52 CLBs is the same as before. The largest free rectangle has grown to dimension 4×11 and contains 44 CLBs. Now there are four free columns.

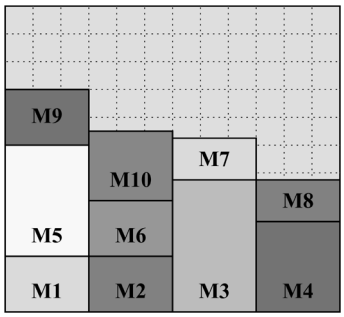


Fig. 11. Illustration for the proof of 1-competitiveness of LIF for modules of constant width: in this example, $k = 4$, $n = 10$, and $k' = 2$; the numbers on the modules correspond to the order in which they arrived.

and 10 for an illustration of Scenario B. Considering our multimedia scenario, we have constrained different IP cores like MPEG2 decoders, MP3 decoders, MicroBlaze core, interface modules like CAN, CardBus, etc., to rectangular shapes. The online placement strategy we used was LIF. For these instances, we report the maximal free rectangle and the number of free columns before and after defragmentation. On an Intel Pentium IV clocked at 3 GHz, the running time for computing the optimum for these instances of an NP-hard problem was less than 0.5 s for each scenario.

As shown in Table I defragmentation increases the area of the maximal free rectangle and the number of free columns in all of the ten scenarios. The smallest increase in area can be seen in scenarios E and J. Here a factor of 1.4 is obtained. In scenarios A and C, an increase of area of the maximal rectangle reaches its maximum with a factor of 3.1. On average, the area of the maximal free rectangles is increased by a factor of 2.2. The number of free columns grows by at least two and by at most six. The average increase of free columns is 4.2.

VI. Online Packing

A. Online Problems

A key motivation for considering partial reconfiguration is the necessity to develop fast and effective strategies for dealing with new and changing demands. In our car scenario, such a situation may arise from a user that requires additional applications for a system with limited resources, a changing traffic environment, or the failure of some system components. In any of those cases, decisions have to be made that are based on incomplete information, as the system does not know what further requests will arise in the future; therefore, we are dealing with *online* scenarios, as opposed to the *offline* scenarios described in Section VI.

In an online scenario, we have to specify a general strategy, as well as a cost function. An evaluation of the strategy is based on a comparison with a best offline solution, i.e., one that could have been achieved with full information; note that computing the latter may require solving an NP-hard problem by itself. In the context of dynamic partial reconfiguration, we can again describe module placement as a 2-D packing problem. In an online setting, rectangular modules arrive one at a time and need to be placed on the chip area.

1-D online packing is well studied; for example the classical bin packing strategies "First Fit," "Best Fit," etc., can be used as online strategies. (See [28] for a relatively recent description of the state of the art, including the performance of more sophisticated methods.)

B. Online Challenges for Partial Reconfiguration

Much less than about 1-D online packing is known about 2-D online packing, and hardly anything for cost functions arising from column-wise reconfiguration as on Xilinx Virtex-II chips, where the reconfiguration cost arises by the interruption of existing modules that use one of the same columns as the newly placed modules. Most immediate are the following two important questions.

1) What is a reasonable strategy for dealing with an online scenario?
2) How can one compute an optimal offline solution in order to allow comparisons?

In the context of this paper, a third question follows.

3) How can one use defragmentation in order to reduce online reconfiguration cost?

C. LIF Versus the Optimum

From the previous discussion, it is not surprising that an answer to question 1) is LIF. Before we argue that it is indeed a good strategy, let us note the following.

Theorem 6.1: Consider a sequence of modules that is placed in some arbitrary permutation, resulting in an arrangement of

TABLE I
RESULTS FOR TEN DIFFERENT SCENARIOS, BASED ON FIG. 7. THE SOLUTION FOR SCENARIO A IS SHOWN IN FIG. 8. SCENARIO B IS SHOWN IN FIG. 9, ITS SOLUTION IN FIG. 10. THE NEXT COLUMNS SHOW THE NUMBER OF PLACED MODULES, THE TOTAL FREE SPACE, THE MAXIMAL FREE RECTANGLE, AND THE NUMBER OF FREE COLUMNS BEFORE DEFRAGMENTATION. THE FINAL COLUMNS SHOW RESULTS AFTER DEFRAGMENTATION

			Before defragmentation		After defragmentation	
Scenario	$\|I\|$	Free space	Max. rectangle	Free columns	Max. rectangle	Free columns
A	11	30	7×1	0	2×11	2
B	9	52	2×8	0	4×11	4
C	9	70	3×7	0	6×11	6
D	9	42	4×4	0	3×11	3
E	6	83	6×8	0	6×11	6
F	6	54	8×2	0	4×11	4
G	5	76	6×4	2	6×11	6
H	6	53	3×11	3	4×11	7
I	5	87	9×6	1	7×11	7
J	6	42	3×8	0	3×11	3

modules. Then the total number of module interruptions is independent of the order in which the modules were placed.

Proof: Let T be the total number of interruptions occurring in some permutations and consider two arbitrary modules. If they are placed in a way that makes them overlap in some column, then the second module to be placed must have interrupted the first, corresponding to one interruption, regardless of which of the two was first and which was second; on the other hand, if the modules do not share some column, there was no interruption of one by the other. This means that there is a one-to-one correspondence between interruptions and edges in the graph G_1, i.e., the interval graph obtained by vertical projection of the placed modules. As the number of edges in G_1 is only determined by the final arrangement of modules, so is T. ■

This means that we should aim at minimizing the total number of edges of G_1, which is precisely what LIF is trying to do in a greedy fashion. As the following shows, this works particularly well for slot-oriented architectures such as the Erlangen Slot Machine (ESM, see [29].)

Theorem 6.2: Consider a set of modules, all having the same width $w_i = w$, that are to be placed on a chip area of sufficient height. Then LIF is a 1-competitive online algorithm, i.e., optimal at each step: regardless of when it is stopped, it has produced a solution with the minimum possible number of interruptions for the set of modules it has faced.

Proof: Let W be the width of the chip; this means that the chip can be considered to be subdivided into $k = \lfloor W/w \rfloor$ slots, accommodating k modules without overlap. If the number of modules is n, LIF places them in $\lfloor n/k \rfloor$ full layers and an extra layer of $k' = n - k\lfloor n/k \rfloor$ modules. This results in k' slots with $\lceil n/k \rceil$ modules each, and $k - k'$ slots with $\lfloor n/k \rfloor$ modules. Note that this is the only way of allocating n modules to k slots with no two slots having a number of modules that differs by more than one.

Now consider an optimal allocation of the n modules to the k slots; let n_i be the number of modules in slot i. This means that the total number of interruptions between modules in slot i is $n_i(n_i-1)/2$; as there is no overlap between modules in different slots, the total number of interruptions is $\sum_{i=1}^{k} n_i(n_i-1)/2$. Suppose there are two slots j and ℓ, such that $n_j \geq n_\ell + 2$: then moving one module from slot j to slot ℓ reduces the number of interruptions in slot j by $n_j(n_j-1)/2 - (n_j-1)(n_j-2)/2 = n_j - 1$, while increasing the number of interruptions in slot ℓ by $n_\ell(n_\ell+1)/2 - n_\ell(n_\ell-1)/2 = n_\ell < n_j - 1$; thus, we get a better allocation of modules to slots, contradicting our assumption of considering an optimal allocation. This shows that no optimal allocation can ever have two slots whose number of modules differs by more than one; as discussed before, this is precisely what LIF produces.

TABLE II
RESULTS FOR TEN DIFFERENT ONLINE SCENARIOS. * WITHOUT M11; ** INCLUDING M11

Scenario	$\|I\|$	LIF Edges	OPT Edges
A	11	12*	11* or 13**
B	9	9	9
C	9	6	6
D	9	10	9
E	6	4	3
F	6	4	3
G	5	1	1
H	6	3	2
I	5	1	1
J	6	4	3

Moreover, our approach allows it to compute an arrangement of modules that is optimal with respect to reconfiguration cost, thus making it possible to perform ex-post comparison of online strategies like LIF with an optimal solution:

Theorem 6.3: The set of module placements that incurs minimal total configuration cost T corresponds precisely to those feasible packings that have smallest possible number of edges in the interval graph G_1.

Proof: This result is an immediate consequence of the proof of Theorem 6.1.

As our OPP algorithm is already based on a tree search that makes use of the interval graphs, minimizing this edge number only requires a simple extension to our tree search. ■

D. Experimental Results

Using the same benchmark instances as before, we compared LIF with an optimal placement, as computed by the defragmentation algorithm in Fig. 6 with additional minimization of the edge number of G_1. See Table II for an overview and Figs. 12 and 13 for a particular comparison. It is clear to see that LIF is near-optimal for most instances.

An important issue in the context of online placement strategies is illustrated by applying LIF to instance A, as shown in

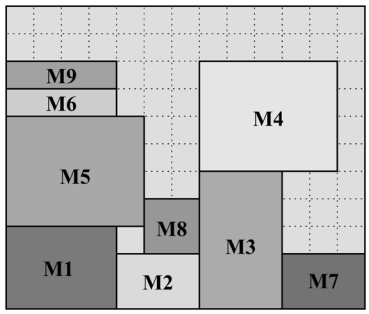


Fig. 12. Placement for scenario B, obtained by iteratively placing modules M1, M2,. . .,M9 by LIF. This results in a total reconfiguration cost of ten overlaps, i.e., ten interruptions.

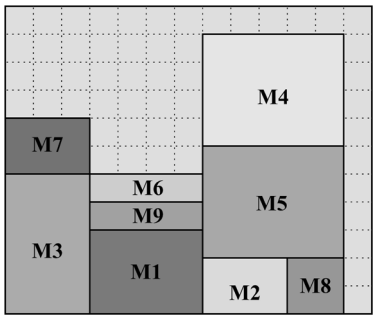


Fig. 13. Optimal placement for scenario B, resulting in a total reconfiguration cost of nine overlaps, i.e., nine interruptions.

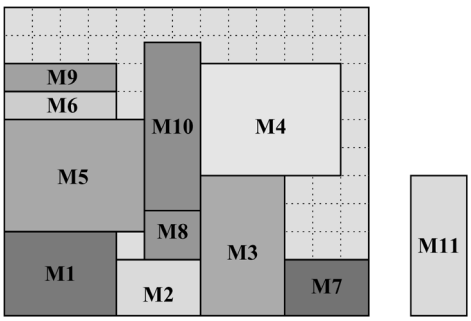


Fig. 14. Applying LIF to scenario A results in insufficient space for the last module.

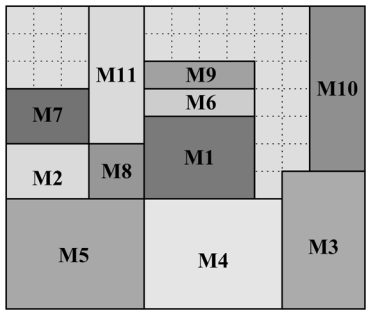


Fig. 15. Optimal solution for scenario A: a feasible placement for all modules with minimum total overlap.

Fig. 14: after placing modules M1,. . .,M10, there is no sufficient free space for placing M11. On the other hand, Fig. 15 shows an optimal placement for this instance.

Obviously, such occurrences are to be expected, as the existence of a feasible placement is the NP-hard OPP. This highlights the importance of defragmentation at appropriate times.

VII. Conclusion

We have shown that mixing online and offline strategies can improve the overall reconfiguration process in partial reconfiguration. Especially for FPGAs with partial reconfiguration restricted to column-wise reconfiguration, a defragmentation strategy as proposed in this paper helps to reduce the interference with other modules.

There are many possible extensions to our approach. We list the following two explicitly.

1) *Malleable Modules:* Tools for automatic synthesis normally do not create modules with rectangular shape. Instead, width and height of the modules can be chosen freely within certain technical bounds. This gives more room for the optimization in the defragmentation process. In a mathematical context this model would be called a *class strip packing problem*. Given a set of modules that has to be placed on a chip as to minimize the total number of columns used, choose for each module from a certain set of module realizations and try to find a placement.
   If the width and height of the modules can be chosen freely this problem is known as *strip packing with modifiable boxes*. In an offline setting this problem can be trivially solved by applying once the volume lower bound as described above and then setting the height of each box to this value. In [30], Imreh gives a four-competitive online algorithm for the problem and gives a class of instances for which no online algorithm can achieve a better competitive ratio than 1.73.
2) *Fixed Modules:* In most FPGA designs, pins of the FPGA are hard-wired. In this setting, it may be unavoidable or preferable to fix a placement of the respective modules in close proximity to I/O pins. When this is the case, the defragmentation problem is no longer a strip-packing problem. Freeing as many columns as possible can be achieved by placing other modules above or below the interface modules and not just as far as possible to the left.

We are optimistic that our general approach will allow some progress on these problem classes.

Our current research focuses on defragmentation algorithms for a different scenario. Here, we no longer require the system to have longer idle times as in our car scenario. Instead modules are rearranged one by one and the defragmentation process is more or less continuous.

Acknowledgment

The authors would like to thank J. Schepers for letting them continue the work with the packing code that he started as part of his thesis [24].

References

[1] J. van der Veen, S. Fekete, M. Majer, A. Ahmadinia, C. Bobda, F. Hannig, and J. Teich, "Defragmenting the module layout of a partially reconfigurable devices," in *Proc. Int. Conf. Eng. Reconfigurable Syst. Algorithms*, T. P. Plaks, Ed., 2005, pp. 92–101 [Online]. Available: http://arxiv.org/abs/cs.AR/0505005

[2] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1393–1407, Nov. 2004.

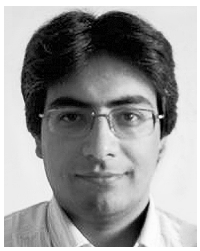
[3] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic scheduling of tasks on partially reconfigurable FPGAs," *IEE Proc.—Computers Digital Techniques*, vol. 147, no. 3, pp. 181–188, May 2000.
[4] Xilinx Inc., San Jose, CA, "Virtex-II Platform FPGAs: Complete data sheet," Jun. 2004.
[5] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich, "A dynamic NoC approach for communication in reconfigurable devices," in *Proc. Int. Conf. Field-Program. Logic Appl. (FPL)*. Antwerp, Belgium: Springer, Aug. 2004, vol. 3203, pp. 1032–1036.
[6] E. L. Lawler, J. K. Lenstra, A. H. G. Rinooy Kan, and D. B. Shmoys, "Sequencing and scheduling: Algorithms and complexity," in *Logistics of Production and Inventory*, ser. Handbooks in Operations Research and Management, S. C. Graves, A. H. G. Rinnooy Kan, and P. H. Zipkin, Eds. Amsterdam: North-Holland, 1993, vol. 4, pp. 445–522.
[7] A. Ahmadinia and J. Teich, "Speeding up online placement for XILINX FPGAs by reducing configuration overhead," in *Proc. IFIP Int. Conf. VLSI-SOC*, Darmstadt, Germany, Dec. 2003, pp. 118–122.
[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.
[9] M. Grötschel, "On the symmetric travelling salesman problem: Solution of a 120-city problem," *Mathematical Programming Study*, vol. 12, pp. 61–77, 1980.
[10] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "On the solution of traveling salesman problems," *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung*, vol. ICM III, pp. 645–656, 1998.
[11] S. P. Fekete and J. Schepers, "A new exact algorithm for general orthogonal d-dimensional knapsack problems," in *Algorithms—ESA*. New York: Springer, 1997, vol. 1284, Lecture Notes in Computer Science, pp. 144–156.
[12] S. P. Fekete and J. Schepers, "A combinatorial characterization of higher-dimensional orthogonal packing," *Math. Operations Res.*, vol. 29, pp. 353–368, 2004.
[13] J. E. Beasley, "An exact two-dimensional non-guillotine cutting tree search procedure," *Operations Res.*, vol. 33, pp. 49–64, 1985.
[14] E. Hadjiconstantinou and N. Christofides, "An exact algorithm for general, orthogonal, two-dimensional knapsack problems," *Eur. J. Operations Res.*, vol. 83, pp. 39–56, 1995.
[15] S. Martello, M. Monaci, and D. Vigo, "An exact approach to the strip-packing problem," *INFORMS J. Comput.*, vol. 15, no. 3, pp. 310–319, 2003.
[16] S. P. Fekete and J. Schepers, "A general framework for bounds for higher-dimensional orthogonal packing problems," *Math. Methods Operations Res.*, vol. 60, pp. 311–329, 2004.
[17] S. P. Fekete, J. Schepers, and J. van der Veen, "An exact algorithm for higher-dimensional orthogonal packing," *Operations Res.*, vol. 55, pp. 569–587, 2007.
[18] J. Teich, S. P. Fekete, and J. Schepers, "Optimal hardware reconfiguration techniques," *J. Supercomput.*, vol. 19, pp. 57–75, 2001.
[19] S. P. Fekete and J. Schepers, "New classes of lower bounds for bin packing problems," in *Proc. Integer Program. Combinatorial Optimization (IPCO'98)*, 1998, vol. 1412, pp. 257–270.
[20] S. P. Fekete and J. Schepers, "New classes of lower bounds for the bin packing problem," *Math. Program.*, vol. 91, pp. 11–31, 2001.
[21] S. P. Fekete, E. Köhler, and J. Teich, "Multi-dimensional packing with order constraints," in *Proceedings 7th International Workshop on Algorithms and Data Structures*. New York: , "," ser. Lecture Notes in Computer Science, vol. 2125, Springer-Verlag, 2001, pp. 300–312.
[22] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*. New York: Academic Press, 1980.
[23] R. H. Möhring, "Algorithmic aspects of comparability graphs and interval graphs," in *Graphs and Order*, I. Rival, Ed. Dordrecht, Germany: Reidel Publishing Company, 1985, pp. 41–101.
[24] J. Schepers, Exakte Algorithmen für orthogonale Packungsprobleme Universität Köln, Tech. Rep. 97-302, 1997.
[25] A. Ghouilà-Houri, "Caractérization des graphes non orientés dont on peut orienter les arrêtes de manière à obtenir le graphe d'une relation d'ordre," *C.R. Acad. Sci. Paris*, vol. 254, pp. 1370–1371, 1962.
[26] P. C. Gilmore and A. J. Hoffmann, "A characterization of comparability graphs and of interval graphs," *Canadian J. Math.*, vol. 16, pp. 539–548, 1964.
[27] B. S. Baker and J. S. Schwarz, "Shelf algorithms for two-dimensional packing problems," *SIAM J. Comput.*, vol. 12, no. 3, pp. 508–525, 1983.
[28] S. S. Seiden, "On the online bin packing problem," *J. ACM*, vol. 49, no. 5, pp. 640–671, 2002.
[29] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, J. Teich, and J. van der Veen, "The erlangen slot machine: A highly flexible fpga-based reconfigurable platform," in *Proc. FCCM*, 2005, pp. 319–320.
[30] C. Imreh, "Online strip packing with modifiable boxes," *Oper. Res. Lett.*, vol. 29, no. 2, pp. 79–85, 2001.

Sándor P. Fekete received the master's degree (Diploma) in mathematics from the University of Cologne, Cologne, Germany, in 1989, and the Ph.D. degree in combinatorics and optimization from the University of Waterloo, Waterloo, ON, Canada, in 1992, both as a Fellow of the German National Merit Foundation (GNMF).

He holds the chair for Algorithmics in the Department of Computer Science, Braunschweig University of Technology, Braunschweig, Germany. His main interests lie in combinatorial optimization, computational geometry, graph algorithms, distributed algorithms, and various applications in computer science, engineering, and economics, such as sensor networks, robot navigation, or traffic control. Following work as a Postdoctoral Research Fellow in computational geometry (SUNY Stony Brook, 1992–1993), he joined the Center for Parallel Computing, the University of Cologne (1993–1999), receiving his habilitation in 1998. After working as an Associate Professor with the Mathematical Optimization Group, TU Berlin, Berlin, Germany (1999–2001), he was a Professor for Mathematical Optimization, Braunschweig (2001–2007), until being appointed to the newly created chair for Algorithmics. He has published over 100 peer-reviewed papers with over 100 coauthors, in publications such as the *Journal of the ACM*, the *SIAM Journal on Computing*, *Algorithmica*, *Mathematical Programming*, *Discrete and Computational Geometry*, the *Journal of Algorithms*, and *Operations Research*; he has also served on program committees for numerous conferences, including the top-ranking algorithmic meetings SODA, ESA, and SCG, and as a reviewer for over 40 journals.

Jan C. van der Veen received the master's degree (Diploma) in mathematics from the Braunschweig University of Technology, Braunschweig, Germany, in 2002.

In 2002, he joined the Department of Optimization, Braunschweig University of Technology, as a Research Associate. His main interests lie in algorithms for packing problems and application of optimization in reconfigurable computing.



Ali Ahmadinia received the B.Sc. degree in Computer Engineering from Tehran Polytechnics University, Tehran, Iran, in 2000, and the M.Sc. degree from Sharif University of Technology, Tehran, Iran, in 2002.

Since 2006, he has been working as a Postdoctoral Research Fellow with the School of Engineering and Electronics, University of Edinburgh, Edinburgh, U.K. In 2003, he joined the chair of Hardware/Software codesign, University of Erlangen-Nuremberg, Erlangen, Germany, as a research assistant; in 2004, he became a member of the electronic imaging group of the Fraunhofer Institute for Integrated Circuits (IIS), Erlangen, where he also finished his Ph.D. dissertation on "Optimization algorithms for dynamically reconfigurable embedded systems". His main research interests include system-on-chip architectures, reconfigurable computing, and DSP applications on embedded systems.

Diana Göhringer received the master's degree (Dipl.-Ing.) from the University of Karlsruhe, Karlsruhe, Germany, in 2006.

In the same year, she joined the chair of Hardware/Software Codesign, University of Erlangen-Nuremberg, Erlangen, Germany, as a Research Assistant.

Mateusz Majer (M'08) received the master's degree (Dipl.-Ing.) from the University of Darmstadt, Darmstadt, Germany, in 2003.

Since 2003, he has been a Research Assistant with the chair of Hardware/Software Codesign, University of Erlangen-Nuremberg, Erlangen, Germany.

Jürgen Teich (M'93–SM'07) received the master's degree (with honors) (Dipl.-Ing.) from the University of Kaiserslautern, Kaiserslautern, Germany, in 1989, and the Ph.D. degree (summa cum laude) from the University of Saarland, Saarbrücken, Germany, in 1993.

Since 2003, he has been an appointed Full Professor with the Computer Science Institute, Friedrich-Alexander University Erlangen-Nuremberg, Erlangen, Germany, holding a chair in Hardware-Software-Co-Design. In 1994, he joined the DSP Design Group of Prof. E. A. Lee and D.G. Messerschmitt in the Department of Electrical Engineering and Computer Sciences (EECS), UC Berkeley, Berkeley, CA, where he was working in the Ptolemy Project (PostDoc). From 1995 to 1998, he held a position at the Institute of Computer Engineering and Communications Networks Laboratory (TIK), ETH Zurich, Zurich, Switzerland, finishing his habilitation entitled "Synthesis and optimization of digital hardware/software systems" in 1996. From 1998 to 2002, he was a Full Professor with the Electrical Engineering and Information Technology Department, University of Paderborn, holding a chair in Computer Engineering. Dr. Teich has been a member of multiple program committees of well-known conferences and workshops, including CODES+ISSS 2007. In 2004, he was elected reviewer for the German Science Foundation (DFG) for the area of Computer Architecture and Embedded Systems. He is involved in many interdisciplinary national basic research projects, as well as industrial projects. Currently, he is supervising more than 20 Ph.D. students.