Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 608312, 12 pages
doi:10.1155/2012/608312

Research Article

Placing Multimode Streaming Applications on Dynamically Partially Reconfigurable Architectures

S. Wildermann, J. Angermeier, E. Sibirko, and J. Teich

Hardware/Software Co-Design, University of Erlangen-Nuremberg, 91058 Erlangen, Germany

Correspondence should be addressed to S. Wildermann, stefan.wildermann@cs.fau.de

Received 29 April 2011; Revised 21 October 2011; Accepted 25 October 2011

Academic Editor: Claudia Feregrino

By means of partial reconfiguration, parts of the hardware can be dynamically exchanged at runtime. This allows that streaming application running in different modes of the systems can share resources. In this paper, we discuss the architectural issues to design such reconfigurable systems. For being able to reduce reconfiguration time, this paper furthermore proposes a novel algorithm to aggregate several streaming applications into a single representation, called merge graph. The paper also proposes an algorithm to place streaming application at runtime which not only considers the placement and communication constraints, but also allows to place merge tasks. In a case study, we implement the proposed algorithm as runtime support on an FPGA-based system on chip. Furthermore, experiments show that reconfiguration time can be considerably reduced by applying our approach.

1. Introduction

Embedded systems used to have fixed functionality dedicated to a specific task. A trend of recent years is, however, to build embedded systems which are regarded as "smart." In this context, the functionality can be adapted, reorganized, and controlled by external requests from human operators or even internal self-managing, self-adapting mechanisms. This means that an embedded system's functionality is not fixed to a single configuration anymore, but several application are running. Still, not all applications are running all the time, but depend on the *operational mode* of the system. Here, we are speaking of *multimode systems*, which can, for example, be found in a typical "smart phone" scenario where the operational mode depends, for example, on the radio link quality or the user's input. Other examples are "smart cameras" which have internal adaptation mechanism that switch between appropriate signal processing algorithms. In this paper, we consider multimode streaming applications which can also be found in above-mentioned examples, in the form of signal and image processing flows, but also Data Stream Management Systems, and so forth.

In a multimode system, the system resources can be utilized better. This is due to the fact that, during system design, only those applications which run concurrently are considered. This makes it possible to perform resource sharing between processes which run mutually exclusive. When speaking of resource sharing of hardware resources, one precondition is the reconfigurability of the computing platform. In this context, field-programmable gate arrays (FPGAs) offer the capability of partial reconfiguration. This means that hardware modules can be dynamically loaded and partly exchanged during runtime. Applying this technology in the design of multimode streaming applications, however, requires (a) system architectures which support partial reconfiguration and (b) proper reconfiguration managers which provide algorithms for scheduling and placing applications at runtime.

This paper deals with modeling and dynamically placing streaming applications on FPGA-based platforms. The focus lies on systems where it is not predictable at design time which applications actually run and in which order they are executed. This is especially the case when implementing autonomic SoCs. For example, [1] introduces a methodology for self-organizing smart cameras which switch between configurations depending on runtime decisions. In this self-managing context, placement needs to be flexible since it is not known beforehand when and what kind of reconfiguration is requested.

One advantage of running streaming applications in hardware is to exploit parallelism by means of pipelining. Therefore, the complete data flow is placed and modules are connected via circuit switching techniques. Particularly in the domain of streaming applications, data flows may have strong similarities. For example, in image processing, convolution filters are commonly applied. Now, while one streaming application uses a Canny filter for edge detection, another application might use a Sobel filter. In this case, it is not necessary to replace the filters by means of reconfiguration when switching between both streaming applications, since the logic remains the same. A better strategy would be to use the same filter module for both applications, and to keep the filter parameters in registers. So, replacing the Canny filter module by the Sobel filter can be achieved by replacing the filter parameters in the registers. It is possible to reduce reconfiguration time when applying this kind of resource sharing.

In this work, we provide a novel approach to merge the data flow graphs of multiple applications with strong similarities into a single data flow graph. We review techniques to build proper reconfigurable systems to run these multimode streaming application. Furthermore, an algorithm is present for placing such streaming applications. The approach considers the heterogeneity of common FPGAs, such as Block-RAM columns, as well as the routing restrictions of on-chip streaming bars.

The paper is organized as follows. Section 2 presents related work. Section 3 provides the preliminaries of partial reconfiguration. Section 4 describes the model for streaming applications as well as the algorithm to generate the merge graph. The problem of placing streaming applications is then formally defined in Section 5. Section 6 provides the placing algorithm. An implementation of the algorithm on an FPGA platform is described in Section 7 which is used in the experiments provided in Section 8. Section 9 concludes this paper.

2. Related Work

Partial run-time reconfiguration of FPGAs has been investigated since several years. But still, reconfiguration harbors many problems. There are of course architectural issues concerning how to support partial reconfiguration and how to establish systemwide communication. Solutions can be classified into three basic principles: *on-chip buses* are the most common way of linking together communicating modules. For example, [2] and [3] present on-chip buses for flexible module placement. Hard macros implement the communication infrastructure within the FPGA fabric. *Circuit switching* is a technique where physically wired links are established between two or more modules as, for example, presented in [4] and [5]. The latter has a low overhead since it implements the wires directly within the routing fabric of the FPGA. We consider this kind of technique for streaming data in reconfigurable systems. *Packet switching* techniques, for example, networks on chip [6], are also possible but impose a certain overhead when implemented on FPGAs.

Hardware reuse is basically investigated on the logic and the system level. Reference [7] provides a mechanism for hardware reuse on the logic level. Frames in common with the previously placed configuration are determined which will then be reused. A similar approach is followed in [8] which aims at increasing the number of common frames by properly ordering the LUT inputs to keep their content the same over several configurations. Such approaches are highly dependent on the architecture and are computationally intensive.

System level approaches are often performed via graph matching techniques. For example, [9] presents an approach where common components between successive configurations are determined to minimize reconfiguration time. Reference [10] presents an approach which is based on the *reconfiguration state graph* (RSG) which models the configurations and the reconfiguration between them. These approaches are applied during synthesis and require knowledge of the switching order between configurations. The result is the synthesized system. Hence, these methods are not applicable when extending a running system without having to perform a complete redesign, also affecting flexibility as well as maintainability and limiting the use of self-managing mechanisms.

Reference [11] introduces an algorithm to place streaming applications with similarities. In order to express these similarities, the authors propose to model the applications by means of extended data flow graphs. Here, so-called OR nodes are used to describe differing parts of applications while keeping their similarities. A First-Fit heuristic is then introduced to place such applications. In the work at hand, we replace the model from [11] by merge graphs. In a merge graph, several data flow graphs are merged into a single representation. However, no OR nodes are required. Each node of the merge graph is annotated with the identifier of the applications it is part of. An algorithm to automatically generate merge graphs is proposed. The placing algorithm from [11] is then extended to work with this novel model.

3. Partially Reconfigurable Architectures

Reconfigurable architectures are typically divided into static and dynamic parts. Logic residing in the static part is needed throughout system operation, for example, the connection logic for the communication interfaces.

On the other side, the dynamic part is designed for loading and exchanging functionality at runtime. Functionality which is only required in some operational system modes may be implemented as *partially reconfigurable modules* (PR modules) and, thus, can be dynamically loaded and exchanged during runtime. In this way, logic and other resources can be shared by system components which only run mutually exclusive.

In the following, we concentrate on hardware reconfiguration on FPGAs. We present possibilities to organize the reconfigurable fabrics to support partial reconfiguration, and give an overview of sophisticated communication techniques, before presenting a concrete example.

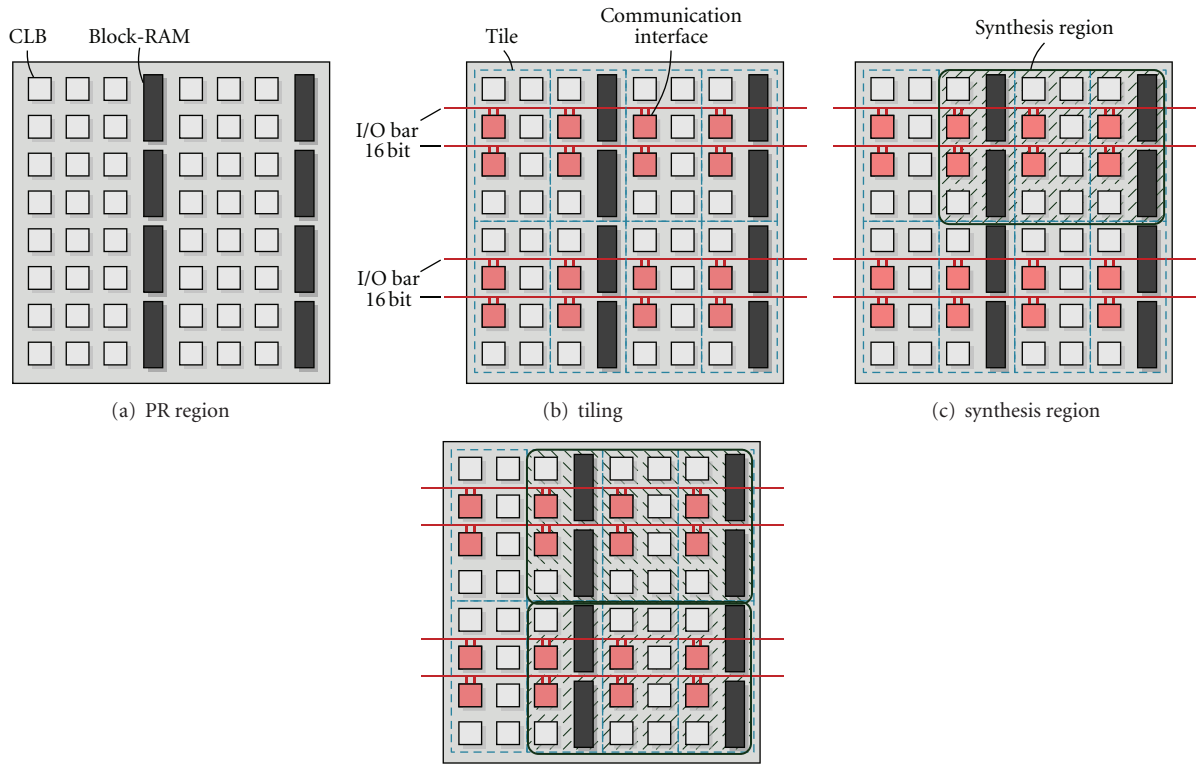

FIGURE 1: Example of a PR region and a possible tiling.

3.1. Enabling Partial Reconfiguration. PR modules can be loaded into dedicated areas on the reconfigurable systems. These areas are denoted as partially reconfigurable regions (PR regions). The possible shapes and positions for loading PR modules into these regions strongly depend on the PR region layout. Here, the problems are that, first, modern reconfigurable fabrics are an array consisting of cells of different types, such as logic blocks (CLBs), Block-RAMs, or multiply-accumulate units (DSPs), as illustrated in Figure 1(a). This increases the inhomogeneity of the underlying structure. Second, low-overhead infrastructures have to be provided that enable systemwide communication and support partial reconfiguration.

One common solution is to build so-called *tiled partially reconfigurable systems*, as proposed, for example, by [2]. Here, the PR regions are divided into *reconfigurable tiles* which are the smallest atomic units for partial reconfiguration. An example for a possible tiling is shown in Figure 1(b). The approach allows to place multiple PR modules with various sizes in the same PR region. At the same time, placement restrictions are decreased and usability is enhanced. Still, the tiles consist of several different cell types according to the underlying cell structure.

Sophisticated communication techniques for reconfigurable systems are commonly provided by *communication macros* which can be placed in the PR region to interconnect the reconfigurable tiles. Since each reconfigurable tile offers a *communication interface*, the tiling of the PR region depends on the chosen communication technique. Koch et al. give a rule in [12] how to calculate the size of tiles. It depends on the communication overhead in terms of required cells as well as the set of modules which are candidates for being placed as PR modules.

The synthesis flow for PR modules requires additional steps compared to the flow for static system designs (see, e.g., [2, 5]). First, an area on the reconfigurable system is selected which provides sufficient cell types as required for implementing the module. This area is denoted as *synthesis region*. Although this does not necessarily have to be the case, most synthesis flows propose the use of rectangular synthesis regions. Then, the communication macro is placed. Finally, the module can be synthesized in the selected region. This flow now allows to place the module at any location with the same arrangement of cell types as the synthesis region. An example of a synthesis region for a PR module requiring 10 CLBs and 4 Block-RAMs is given in Figure 1(c). The two *feasible positions* for placing this PR module are depicted in Figure 1(d).

As pointed out in this section, the chosen communication technique strongly influences the layout of the partially reconfigurable part of the system. In the following, the communication techniques targeted in this work are presented.

3.2. Communication Techniques. Section 2 describes approaches to provide communication interfaces for dynamic placement of hardware modules. *On-chip buses* are suitable for dynamically integrating hardware modules into an FPGA

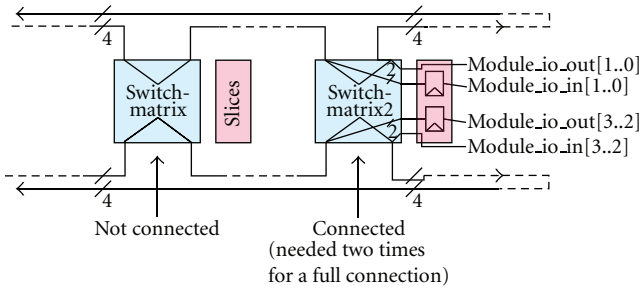

Figure 2: Streaming bar implemented through the FPGA routing fabric.

by partial reconfiguration. Sophisticated techniques [2, 3] are provided as FPGA macros which allow partial reconfiguration at runtime, even during bus transactions. Such buses permit connections between modules, including the communication to and from components of the static part of the system.

Circuit switching techniques are used for streaming data and establishing point-to-point connections between hardware modules. Sophisticated approaches, for example, *I/O bars* [5] allow to implement this behavior with a low overhead within the FGPA routing fabric. The basic principle is depicted in Figure 2. A module is able to read the incoming data, modify it if necessary, and pass it further to the next module. By properly configuring the switch matrix, modules can be connected to these signal by accessing the data via the flip-flops of the Configurable Logic Block (CLB) slices. Hence, modules occupy signals of the streaming bar to send data to successive modules.

An example for building a system that uses an I/O bar for streaming data through a PR region is shown in Figure 3. In this example, four communication macros are placed which are connected via multiplexers in the static part of the system. Furthermore, three modules are depicted which can subsequently access data streamed via the I/O bar.

4. Streaming Applications with Similarities

In many areas of application, it is necessary to process data streams. This is commonly the case in signal and image processing where a set of filters is applied on the data stream, or in Data Stream Management Systems for querying of data in data streams. We denote these kind of applications as *streaming applications*.

4.1. Application Model. A streaming application can be formally defined by a data flow graph (DFG), which is denoted as $G(V, E)$. Vertices $V = T \cup C$ represent both tasks T and communication nodes C. Edges E represent data dependencies between them. Examples of data flow graphs are given in Figure 4.

Each task $t \in T$ represents a functional IP core, which can be implemented and dynamically loaded as PR module. As described in Section 3.1, PR modules can only be placed at locations which have the same arrangements as their synthesis regions. Therefore, the set $M(t)$ contains all PR modules available for implementing a task $t \in T$. Each PR module $m_i \in M(t)$ is described by a rectangular bounding box denoted by the parameters height(m_i) and width(m_i). The arrangement of reconfigurable tiles within this bounding box is represented as a string, denoted by string(m_i). This is a common approach in reconfigurable computing since string matching techniques can be applied to determine the feasible positions for PR modules during online placement. In addition, the parameter type(m_i) $\in$ {Master, Slave} states whether the module serves as a slave module or a master module, where the latter is able to initialize bus requests. This specification is necessary since communication macros only provide a limited amount of signals for connecting master modules to the arbiter and, thus, impose constraints for the online placement.

Our graph merging algorithm, described in the next section, requires that each data flow graph has an initial *no-operation* (NOP) node NOP_0 which is connected to all processes with no predecessor. Furthermore, a final *no-operation* node NOP_n is required to which all processes with no successor are connected.

Each communication node $c \in C$ has exactly one predecessor module (the sender), but may have several successor modules (the receivers). This paper concentrates on communication of streaming applications established via circuit switching techniques, such as I/O bars. Nonetheless, if on-chip buses are also included in the design, this may influence the placement: since buses allow a mutually exclusive medium access, they have to be arbitrated before usage. Each macro provides a limited number of dedicated signals for connecting master modules with the arbiter. Thus, only a restricted number of master modules can be placed per macro.

As explained in Section 3.2, circuit switching techniques provide a limited number of signals which can be accessed by modules. Since the signals are streamed over the architecture, this communication affects the placement. The parameter com(c) denotes requirements of communication bandwidth c in terms of the number of signals that are required to establish the communication between the sending module pred(c) and the receiving modules succ(c). This number of signals is occupied until they are consumed by the last receiving module.

4.2. Merging Data Flows with Similarities. We propose a heuristic based on [13] for merging two DFGs represented by graphs G_i and G_j which contain some or several processes implemented by the same IP cores. The approach is presented in Algorithm 1. It works by extracting all paths of the data flow graphs. Each path represents a sequence of processes and communication nodes. Then, paths of the two graphs are subsequently merged. The basic functionality is to identify common nodes in the paths, and then, merge paths such that common nodes are only included once. Common nodes are defined in the following manner.

Definition 1. We denote processes of two paths as common processes if they may be implemented by the same IP core.

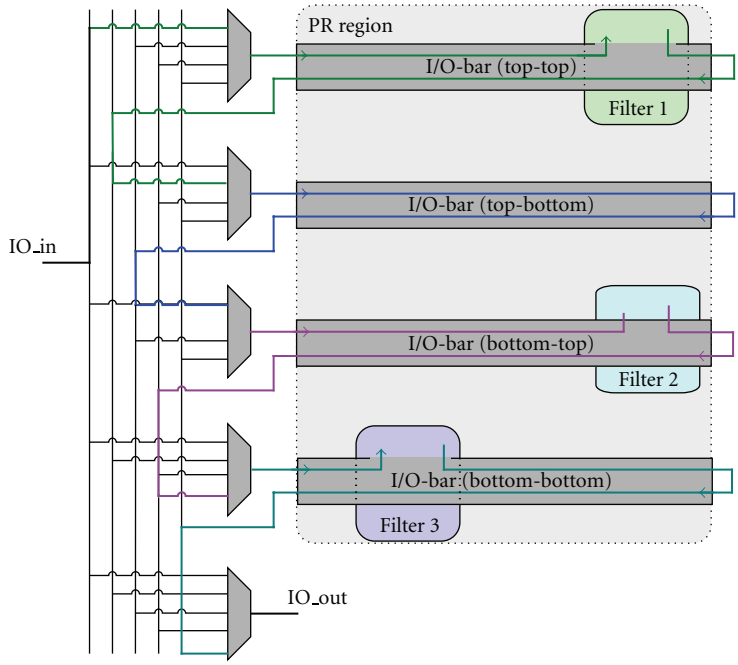

Figure 3: I/O bar used for streaming data through PR region.

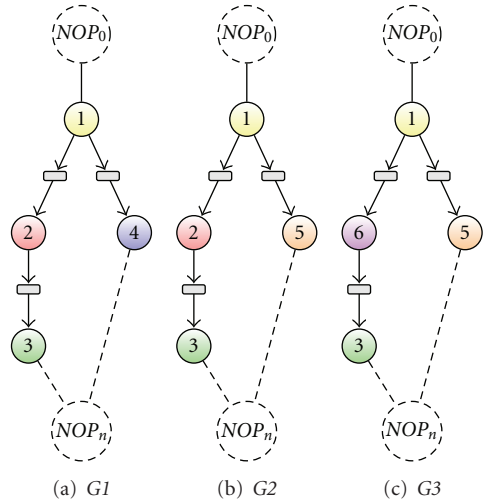

Figure 4: Example of data flow graphs with strong similarities that can be merged.

Moreover, communication nodes of two paths are considered as common communication nodes if they connect senders which are common processes and receivers which are common processes within the paths.

The overall objective is to reduce the reconfiguration time by exploiting similarities. The decision of which paths to merge is therefore based on the idea of [13]. Here, for maximizing the reconfiguration time reduction, the Maximum Area Common Subsequence (MACSeq) and Substring (MACStr) is applied. Each sequence of string consists of a number of components. In the case of [13], not the longest common sequence or string of components in two paths is determined by MACSeq and MACStr, respectively. Rather, the sequence and string determine which components have the highest accumulated area.

In our case, the area is replaced by the reconfiguration time of the PR modules represented by the processes so that sequences and strings with high reconfiguration times are preferred. (Reconfiguration time may depend on the area. In the case of framewise reconfiguration, however, reconfiguration time depends on the width of the PR module.) In the algorithm (lines 4–7), the MACSeq and MACStr $S_{i,j}$ of all pairs of paths are calculated. The pair containing the sequence/string $S_{i,j}^{\max}$ with the maximal accumulated reconfiguration time is selected and merged according to

(1): **initialize** merged graph G_M;
(2): **extract** all paths $P_{i/j}$ of $G_{i/j}$;
(3): **While** at least one pair of paths with non-empty MACSeq or MACStr exists **do**.
(4): **calculate** MACSeq/MACStr $S_{i,j}$ for each pair $(p_i, p_j) \in P_i \times P_j$;
(5): **find** pair (p_i, p_j) with $S_{i,j}^{\max}$ having maximum reconfiguration time;
(6): **merge** paths p_i and p_j according to $S_{i,j}^{\max}$ and add them to G_M;
// see Algorithm 2
(7): $P_{i/j} = P_{i/j} \setminus p_{i/j}$;
(8): **end while**
(9): **add** all remaining paths $p \in P_i$ and $p \in P_j$ as or-paths to G_M;
// see Algorithm 3

Algorithm 1: Merging two data flow graphs G_i and G_j.

(1): **ensure** G_M contains the NOP nodes NOP_0 and NOP_n;
(2): $v_0 = NOP_0$
(3): **for** $k = 0$ **to** $|S_{i,j}|$ **do**
(4): $(v_i^{(k)}, v_j^{(k)}) = S_{i,j}[k]$; // the k-th pair in the sequence
(5): **determine** or-paths $or_{i/j}$, that is, all vertices in $p_{i/j}$ between $v_{i/j}^{(k-1)}$ and $v_{i/j}^{(k)}$
(6): **add** $or_{i/j}$ to G_M; // see Algorithm 3
(7): **make** new node n_M;
(8): **link** $v_M[i]$ to $v_i^{(k)}$ and $n_M[j]$ to $v_j^{(k)}$;
(9): **add** v_M to G_M;
(10): **add** edge to G_M connecting the last vertices added from p_i and p_j to v_M;
(11): $v_0 = v_M$;
(12): **end for**

Algorithm 2: Merging two paths p_i and p_i according to common sequence $S_{i,j}$ and adding them to the merged graph G_M.

Algorithm 2. When no further paths with similarities exist, the remaining paths are added to the merge graph (line 9).

Path merging works as shown in Algorithm 2. Here, $(v_i^{(k)}, v_j^{(k)}) = S_{i,j}[k]$ denotes the kth pair of common nodes in the sequence (line 4). All common nodes are successively added to the merged graph G_M. The nodes of the paths which are not part of the sequence/string and lie between common nodes are denoted as *or-paths* (line 5) and added separately according to Algorithm 3 (line 6). When merging a pair of common nodes, a new node is added to the merged graph which represents the common node. It contains references to the common nodes and is connected with the last vertices from the paths added to the merged graphs (lines 7–10).

Algorithm 3 shows how to add an or-path to the merged graph. Here, the elements in the or-paths are iteratively added. If the current element $v_i^{(k)}$ is not already part of the merged graph G_M, a new node is added to G_M with a reference to $v_i^{(k)}$ (lines 3–6). Then, the last element of the or-path added to G_M is connected to this new node in G_M (line 10).

The proposed heuristic can now be used to subsequently merge several data flow graphs. In all cases, it is required that all graphs have a initial NOP node NOP_0 and a final NOP node NOP_n. As an example, Figure 5 shows the data flow graph resulting when merging the three graphs from Figure 4.

5. Problem Formulation

Placing tasks on a reconfigurable device is a complex problem, in which many resource constraints must be respected. In our scenario, one important limited resource consists in the communication infrastructure, for example, the signals available for circuit switching. Typically, the amount of used connection bandwidth at each position and instant in time must not exceed a certain maximum number. Therefore, it may be necessary to delay the placement of one module until some more communication bandwidth is released. Furthermore, the data dependencies must be respected, that is, all the temporary results on which one tasks depends must have been already computed before it can be executed. In addition, each module may require Block-RAM resources at specific relative positions; thus the Block-RAM resource limitations must also be respected for the task placement. In a more formal manner, the placement problem can be specified as follows.

Given a task graph $G_T(V_T, E_T)$ and a tiled reconfigurable architecture where A represents the set of reconfigurable tiles,

find for each process $t \in T$ with outgoing communication nodes $C_t^{\text{out}} = \text{succ}(t)$ and incoming communication nodes $C_t^{\text{in}} = \text{pred}(t)$ a feasible location $F \subseteq A$

```
(1): for k = 0 to |or_i| do
(2):     v_i^(k) = or_i[k];
(3):     if v_i^(k) is not already part of G_M then
(4):         make new node v_M;
(5):         link v_M[i] to v_i^(k);
(6):         add  v_M to G_M;
(7):     else
(8):         get node v_M in G_M where v_M[i] = v_i^(k);
(9):     end if
(10): add edge (v_0, v_M) to G_M;
(11):    v_0 = v_M
(12): end for
```

Algorithm 3: Adds an or-path or_i to the merged graph G_M. The or-path is part of the path p_i. The last vertex of the path p_i added to G_M is denoted by v_0. If no such vertex is given, $v_0 = NOP_0$ is used as default, that is, the NOP node of G_M.

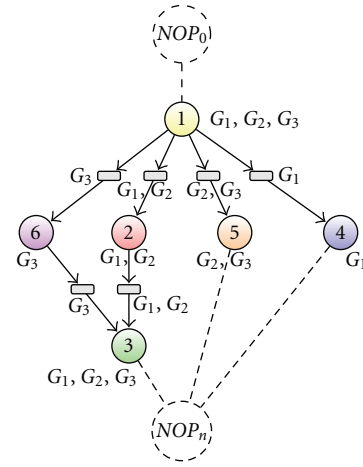

Figure 5: Example of the merge graph of the data flow graphs G_1, G_2, and G_3 from Figure 4. Each node of the merge graph holds a reference to the data flow graph it is a part of.

such that there exists an implementation of t as PR module $m_i \in M(t)$

subject to the following conditions:

(i) height(F) = height(m_i) and width(F) = width(m_i): the module and the selected region have the same bounding box;

(ii) string(F) = string(m_i): the module and the selected region have the same underlying arrangement;

(iii) for all $c \in C_t^{in}$: all predecessors pred(c) have already been placed, and there exists a valid route between their location and F;

(iv) Master(row_y) < MaxMaster Nbr(row_y), where row_y denotes the communication macro accessed via location F. Master(row_y) counts the number of masters already placed at macro row_y, and MaxMasterNbr(row_y) gives the number of links provided by the macro to connect to the arbiter. This constraints take into account the limited number of master modules allowed per macro.

(v) The number of required output lines $\text{com}_{\text{out},t}$ with

$$\text{com}_{\text{out},t} = \sum_{c \in C_t^{\text{out}}} \text{com}(c) \tag{1}$$

may not exceed the number of freely available communication bandwidth, such that there exists sufficient signals to route the communication.

6. Algorithm

A placement algorithm decides where and at what moment in time a module is loaded and executed. In the online case, information about all tasks to be placed is not known from the beginning, but tasks arrive over time. Our proposed online placement is given as Algorithm 4. In each iteration, a module candidate is determined out of the list of modules to be placed. Therefore, it is checked for each module if all its predecessors are already placed. In the case that this is fulfilled, the communication bandwidth requirements, for example, streaming bar signals, are checked. Either, there must be enough free bandwidth available, or bandwidth claimed by some predecessor can be freed, because it is no longer necessary as the data end points have already been placed. If there are multiple modules which all fulfill these requirements, then a priority-based approach is used to determine the selection of the module. Usually modules with more descendants in their data flow graph receive a higher priority. Furthermore, the priorities can also depend on user-defined objectives or the underlying architecture. The priorities are dynamic, meaning that they are determined new in each interation.

An important step of the algorithm is to determine the possible feasible positions. Hereby, the candidate rectangles are dependent on the underlying architecture graph. As described in Section 3, heterogeneities on the reconfigurable device partition the reconfigurable area. Based on that, an initial list of possible rectangles is created. Hereby, rectangles can have different heights, measured in resource slots, and are sorted in the order of their appearance in the architecture graph. A rectangle is determined in the lists by the *First-Fit* rule, that is, the first rectangle found in which the module can be placed. Thus, modules are tried to be placed near the origin at coordinate $(0, 0)$ to reduce fragmentation. Data dependencies are respected by first checking for each module if all predecessing modules are already placed. Special care must be taken for modules with Block-RAM requirements which are represented with a string and are checked by a string matching approach.

The complexity of the algorithm is an important issue, because it is not performed offline, but during the runtime. The overhead of performing the algorithm contributes to the overall execution time. Therefore, we kept an eye on

Initialize list of rectangles, communication bandwidth usage;
for $i = 1$ to $|M|$ **do**
  Determine set of modules for which data-dependencies are met, and still unplaced, named K_i;
  Remove modules from K_i, for which currently not enough communication bandwidth is available;
  Remove modules from K_i, for which no suitable feasible position can be found in G_A with a first-fit heuristic.
  If more than one module is still included in K_i, then select one according to criteria such as
  number of descendants, module size, Block-RAM usage, and so forth.
  Place K_i in the determined feasible position and update resource usage informations;
end for.

Algorithm 4: Placement Algorithm.

the performance of the algorithm and followed a list-based technique. The algorithm consists of a loop with various checks for suitability over all still available tasks. More precisely, it has a running time of $O(|M|^2)$. Thus, it can be executed efficiently also at runtime. Other offline approaches, like [14], might achieve better solutions, but are to complex too be employed in an online setting, where the tasks to be considered might also change over runtime, and downtime of the system must be minimized. Thus, new modules can be developed and be integrated in the running reconfigurable system, because the placement algorithm is fast and flexible enough to adapt at runtime to the new situation.

However, when placing a merge graph, special care must be taken. In this case, we start placing the modules from the left top-most position towards the right bottom-most position. Furthermore, we keep pointers for all streaming applications that are aggregated into the merge graph. Each pointer indicates the position from which the First-Fist heuristic starts searching. Each time a module is selected according to Algorimth 4, the heuristic starts searching from that pointer that is most advanced and belongs to an application the module is part of. After the module is placed, all affected pointers are advanced onto the first free position after the bounding box of this module. By successively performing this step all modules are placed, and amodule running mutually exclusive may use the same resources.

7. Case Study

Figure 6 illustrates the FPGA-based system on chip from [15] which is implemented on a Xilinx Virtex-II XUP Pro board and serves as a case study. The two dark-red areas on the right top and bottom compose the dynamic part of the system. Reconfiguration is only possible in the dynamic part which contains a reconfigurable on-chip bus and streaming bars as communication primitives. Both primitives are generated by the framework ReCoBus-Builder [5]. Note that the partial modules possess different sizes and that modules can be placed in a two-dimensional manner since four macros are placed in the partial part.

The proposed placement algorithm was implemented for the described SoC. The procedure of runtime reconfiguration is illustrated in Figure 7, highlighting the steps required to dynamically load PR modules. Reconfiguration is basically performed by keeping an image of the current FPGA configuration in the memory. This is used as a virtual representation of the FPGA. When a PR module is loaded, its bitfile is loaded from the memory and combined with the virtual representation (see steps 1 to 4 in Figure 7). The module is then physically loaded by writing the modified frames of this bitfile to the FPGA via the ICAP interface (see steps 5 to 8 in Figure 7). The use of the virtual representation is necessary since reconfiguration happens in a framewise manner, and thus, also logic from the static part has to be rewritten when placing a new PR module. Details of this procedure are described in [15].

In addition to the static bitfile and the bitfiles of the PR modules, the CF card also contains the application descriptions provided in XML format. This XML format allows to described the application by specifying the nodes and the edges of the data flow graphs or merge graphs, respectively. Each node is associated with the required parameters, such as the name of the bitfile of the PR module and the parameters specified in Section 4. When loading a new application, the XML description is first loaded to initialize the placement algorithm. Then, candidate modules are selected and positions determined as described in the last section.

8. Experiments

We have performed several experiments in order to evaluate the proposed approach. The architecture from the case study was chosen to perform the experiments. Furthermore, 6 data flow graphs were generated, consisting of 5 to 15 nodes. Processes were chosen from 5 different IP block types, where each type is associated with a PR module, specified by its bounding box and arrangement as well as a reconfiguration time which is proportional to the PR module width. These data flow graphs are chosen according to image processing applications already implemented on the architecture described in Section 7. Details of this kind of filters and data flows are presented in [15, 16]. As shown there, the input image can be sent pixel by pixel via the I/O bar. The pixels can then be accessed by image filters which are loaded as PR modules. The width of the I/O bar chosen in our experiments is 32 bit. The test case applications are assumed to work on grayscale input images. Therefore, the required output lines per communication node are set to 8 bit.

Now, the presented placement approach allows to place such applications by respecting the placement constraints

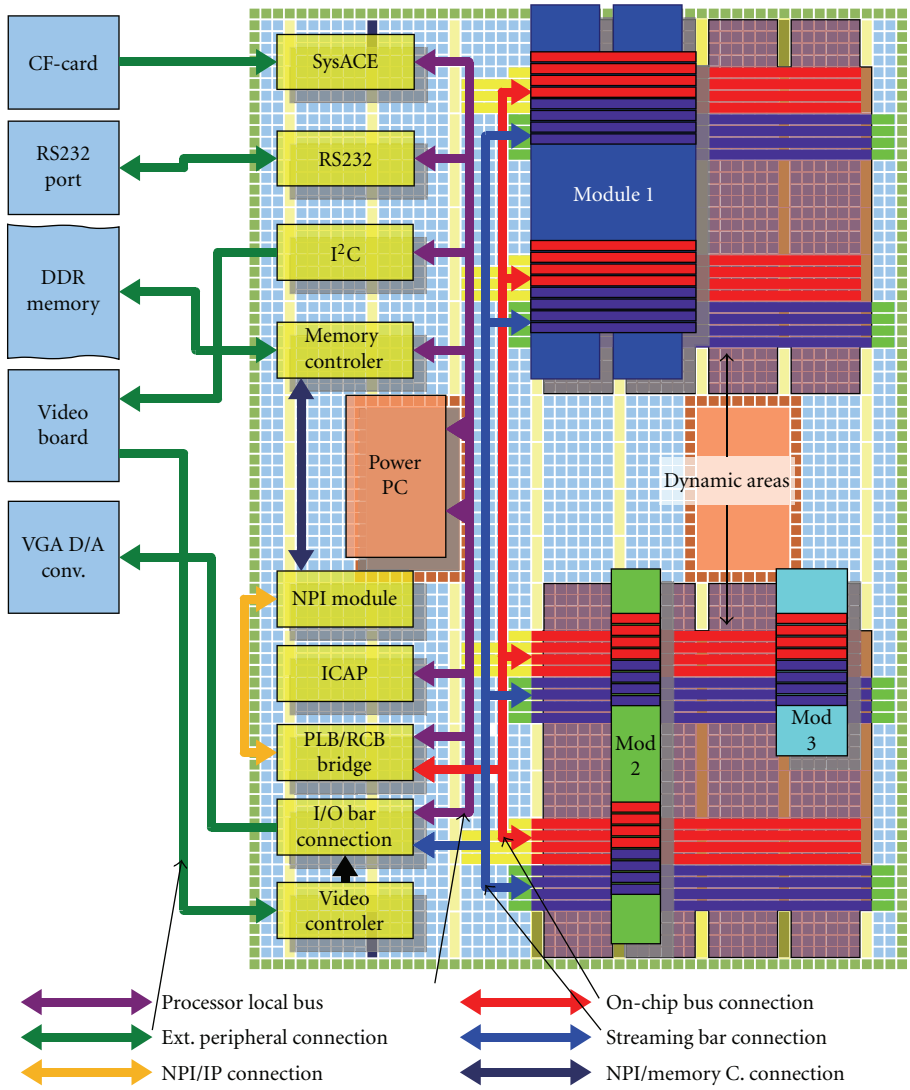

Figure 6: System overview of one possible partitioning of a heterogeneous FPGA-based SoC platform consisting of CPU subsystem and reconfigurable area from [15]. Reconfigurable modules can vary in size and be freely placed, allowing a very good exploitation of the FPGA space. The on-chip ReCoBus (red) and the streaming bar (blue) are routed through the dynamic part, allowing a systemwide communication between hardware and software modules.

as well as the data dependencies expressed by the data flow graph, so that pixels are accessed and processed in the correct order. Furthermore, when switching between image processing applications, our merging approach allows to reduce the required reconfiguration time. This is analyzed in the following experiments.

We have chosen the following metrics to compare our test runs. Let $\mathcal{G}$ denote the set of data flow graphs which are merged into graph $G_M(V_M, E_M)$ according to Algorithm 1. Then, the *similarity* s_0 is measured according to

$$s_1 = \frac{\sum_{G(V,E)\in\mathcal{G}} |V| - |V_M|}{\sum_{G(V,E)\in\mathcal{G}} |V|},$$
$$s_0 = \frac{s_1}{(|\mathcal{G}| - 1)/|\mathcal{G}|}, \tag{2}$$

where $0 \leq s\_1 \leq (|\mathcal{G}| - 1)/(|\mathcal{G}|)$ measures how much we can reduce the number of nodes by merging the graphs $G(V, E) \in \mathcal{G}$. In the best case, all graphs would be exactly the same and consist of vertex set V. Here, s_1 would have the value $(|V| \cdot |\mathcal{G}| - |V|)/(|V| \cdot |\mathcal{G}|)$. In the worst case, no similarities do exist and s_1 would be 0. So, the similarity $s_0 \in [0, 1]$ measures how far we are away from the best case.

The area rate indicates how much the area required for placing the merged graph differs from the area required when not merging the graphs. In the second case, resource sharing is enabled so that graphs are placed separately and may also occupy the resource used by the other graphs. Each graph $G \in \mathcal{G}$ is placed separately according to the proposed algorithm, and the area requirements $area_G$ stemming from this placement is calculated. Then, the maximal area $\text{area}_{|\mathcal{G}|}^{\max} = \max_{G\in\mathcal{G}}\{\text{area}_G\}$ is determined. We have

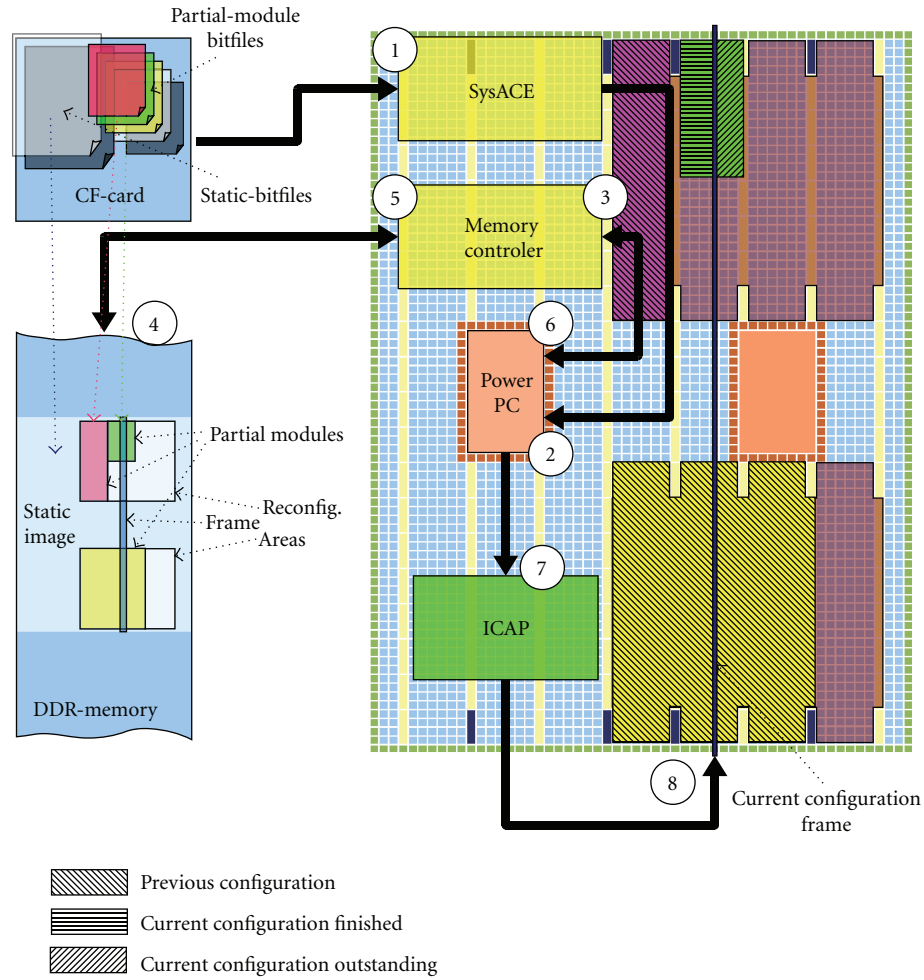

Figure 7: Reconfiguration procedure.

furthermore calculated the area required for placing the merged graph according to the described algorithm. This area is denoted as area_{G_M}. The *area rate* is given as

$$\frac{\text{area}_{G_M}}{\text{area}^{\max}_{|\mathcal{G}|}}. \tag{3}$$

The *reconfiguration time rate* states how much we can reduce the reconfiguration time when merging the graphs. For each pair $G_1, G_2 \in \mathcal{G}$ with $G_1 \neq G_2$, we calculate the reconfiguration time required for switching from G_1 to G_2 (a) by completely placing G_2 without exploiting similarities of the data flow graphs and (b) by using the merged graph and only loading the differing PR modules. By dividing the maximal values, we obtain the reconfiguration time rate.

In the first set of experiments, we have generated all $\binom{6}{2} = 15$ pairs for the data flow graphs. Each pair was merged according to the proposed algorithm and evaluated according to the metrics described above. The results are shown in Table 1 and indicate a correlation between the similarity of graphs and the potential to reduce the reconfiguration time. Test-case A2 has the biggest similarity and lowest reconfiguration time rate, thus reducing the reconfiguration time by 76% when applying the merged graph. However, the similarity metric does not consider the reconfiguration time of the merged processes. So, it is not possible to directly relate this metric with the reconfiguration time rate. This can, for example, be seen in test-cases A7 and A8. A7 has the minimal similarity but using the merged graph reduces the reconfiguration time by 17%. In contrast, A8 has a higher similarity than A7. Nonetheless, using merge graphs only reduces the reconfiguration time by 8%. Still, the experiments show the benefits of the proposed approach. Reconfiguration times can be reduced in all cases, ranging from 6% to 76% for the test cases.

The second observation is that, in most cases, there is no difference between separate placement and using the merged graph regarding the required area. In some cases, the area requirement can even be reduced. Then, in one case, it is larger than placing the graphs separately. This difference stems from the fact that the candidate list used in Algorithm 4 differs in the approaches. This affects the order in which PR modules are placed and may consequently result in different area requirements.

In the second set of experiments, we have generated all 20 combinations when choosing 3 out of the 6 data flow graphs. The graphs were merged and again evaluated

Table 1: Overview of the results of the first set of experiments with two graphs merged.

test case	No. of graphs	similarity	area rate	reconf. rate
A1	2	0.29	1.0	**0.94**
A2	2	**0.89**	1.0	**0.24**
A3	2	0.80	0.86	0.47
A4	2	0.67	1.00	0.75
A5	2	0.67	1.00	0.43
A6	2	0.25	**0.62**	0.76
A7	2	**0.22**	0.86	0.83
A8	2	0.40	1.00	0.92
A9	2	0.25	1.00	0.70
A10	2	0.73	0.86	0.40
A11	2	0.57	1.00	0.81
A12	2	0.40	1.00	0.48
A13	2	0.50	**1.16**	0.87
A14	2	0.55	1.00	0.57
A15	2	0.29	1.00	0.87

Table 2: Overview of the results of the second set of experiments with three graphs merged.

Test case	No. of graphs	similarity	Area rate	Reconf. rate
B1	3	0.63	**0.62**	0.95
B2	3	0.58	1.00	**0.97**
B3	3	0.50	1.00	0.94
B4	3	0.50	1.00	0.70
B5	3	0.80	0.86	0.57
B6	3	**0.82**	1.00	0.81
B7	3	0.75	1.00	**0.43**
B8	3	0.75	1.00	0.87
B9	3	0.70	0.73	0.57
B10	3	0.68	1.00	0.96
B11	3	0.64	**1.16**	0.83
B12	3	0.45	1.00	0.81
B13	3	0.46	1.00	0.70
B14	3	**0.41**	0.86	0.87
B15	3	0.43	1.00	0.83
B16	3	0.45	1.00	0.87
B17	3	0.69	1.00	0.87
B18	3	0.66	1.00	0.57
B19	3	0.63	1.00	0.96
B20	3	0.58	**1.16**	0.87

according to the chosen metrics. The results are shown in Table 2. We see again that there is no direct proportionality of reconfiguration time rate and similarity, since the latter only measures the similarity of the graph topology. In all test cases, we are able to reduce the reconfiguration time. In the best case, this are 57% for B7, and in the worst case, 3% for B2. Again, the area requirements stay the same for most test cases. However, in B11 and B20, the requirements grow by 16%. Then again, the requirements can be reduced by 38% in the case of B1.

All in all, the experiments show the benefits regarding the reconfiguration time when using our approach. In the best case, we could reduce the time by 76% for the first set of experiments, and by 57% for the second set. The experiments have also shown, that the area requirements are only affected in some cases. This is due to the fact, that the candidate lists used by the proposed First-Fit heuristic differ when placing the graphs separately, or as a merged graph. Here again, the replacement of the proposed First-Fit algorithm by a more sophisticated search would probably lead to a convergence towards 1.00.

9. Conclusion

Partial reconfiguration of FPGAs allows to dynamically change the configuration of a hardware system at runtime. By applying this technique, it is possible to perform resource sharing between streaming applications which are part of different operational mode. While this allows to build smaller systems, the time and power needed to perform hardware reconfiguration increases. In this case, it is wise to keep similar modules and only replace the differing parts.

This paper deals with merging streaming applications which into a single representation, denoted as merge graph. Besides an algorithm to perform this step, architectural issues of partial reconfiguration and on-chip communication are discussed, and an architecture is described which contains techniques to deal with this. In addition, the paper presents a placement algorithm for streaming applications and merge tasks. In order to be applied in real world, the placement approach takes into account the heterogeneities of the reconfigurable device and respects the bandwidth constraints of the communicating processes. The complexity of the proposed algorithm allows the execution to be performed efficiently at runtime. Of course, offline approaches [14] might find better solutions, but are to complex to be employed in an online scenario.

The experiments show that the proposed approach allows fast switching between streaming applications. In all cases, we were able to reduce the reconfiguration time. In the best-case, the reconfiguration time could even reduce by 76%. The experiments furthermore show that the area requirements do not degenerate by placing the merge graphs approach for a majority of the test cases. In some cases, it even gets better than placing the streaming applications separately. Here, however, the results demonstrate that, due to varying candidate list resulting from separate or merged placement, the proposed First-Fit heuristic is very sensitive on changes in the placement order of PR modules. In future work, the heuristic will therefore be replaced by a more stable placement technique.

References

[1] S. Wildermann, A. Oetken, J. Teich, and Z. Salcic, "Self-organizing computer vision for robust object tracking in smart

cameras," in *Proceedings of the 7th International Conference on Autonomic and Trusted Computing (ATC '10)*, vol. 6407 of *Lecture Notes in Computer Science*, pp. 1–16, Springer, 2010.

[2] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, "Design of homogeneous communication infrastructures for partially reconfigurable FPGAs," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '07)*, pp. 238–247, Las Vegas, Nev, USA, June 2007.

[3] D. Koch, C. Haubelt, and J. Teich, "Efficient reconfigurable on-chip buses for fpgas," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 287–290, April 2008.

[4] H. ElGindy, H. Schroder, A. Spray, A. K. Somani, and H. Schmeck, "RMB—a reconfigurable multiple bus network," in *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA '96)*, pp. 108–117, February 1996.

[5] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder—a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 119–124, September 2008.

[6] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "DyNoC: a dynamic infrastructure for communication in dynamically reconfigurable devices," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 153–158, August 2005.

[7] U. Malik and O. Diessel, "On the placement and granularity of FPGA configurations," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 161–168, December 2004.

[8] K. P. Raghuraman, H. Wang, and S. Tragoudas, "A novel approach to minimizing reconfiguration cost for LUT-based FPGAs," in *Proceedings of the 18th International Conference on VLSI Design: Power Aware Design of VLSI Systems*, pp. 673–676, January 2005.

[9] N. Shirazi, W. Luk, and P. Y. K. Cheung, "Automating production of run-time reconfigurable designs," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pp. 147–156, IEEE Computer Society, Napa Valley, Calif, USA, 1998.

[10] M. Rullmann and R. Merker, "Design methods and tools for improved partial dynamic reconfiguration," in *Dynamically Reconfigurable Systems—Architectures, Design Methods and Applications*, pp. 147–163, Springer, Berlin, Germany, March 2010.

[11] J. Angermeier, S. Wildermann, E. Sibirko, and J. Teich, "Placing streaming applications with similarities on dynamically partially reconfigurable architectures," *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 91–96, 2010.

[12] D. Koch, C. Beckhoff, and J. Teich, "A communication architecture for complex runtime systems and its implementation on spartan-3 FPGAs," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 253–256, February 2009.

[13] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *Proceedings of the 41st Design Automation Conference (DAC '04)*, pp. 395–400, June 2004.

[14] S. Wildermann, F. Reimann, D. Ziener, and J. Teich, "Symbolic design space exploration for multi-mode reconfigurable systems," in *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '11)*, pp. 129–138, 2011.

[15] A. Oetken, S. Wildermann, J. Teich, and D. Koch, "A bus-based SoC architecture for flexible module placement on reconfigurable FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 234–239, Milano, Italy, August 2010.

[16] D. Ziener, S. Wildermann, A. Oetken, A. Weichslgartner, and J. Teich, "A flexible smart camera system based on a partially reconfigurable dynamic FPGA-SoC," in *Proceedings of the Workshop on Computer Vision on Low-Power Reconfigurable Architectures at the FPL*, Chania, Greece, September 2011.