

# Chapter 8

## Design Methods and Tools for Improved Partial Dynamic Reconfiguration

Markus Rullmann and Renate Merker

*We dedicate this chapter to Prof. Wunsch  
on the occasion of his 85th anniversary.*

**Abstract** In current FPGAs the overhead associated with partial dynamic reconfiguration limits the application of this method in system design. We review the origins of this overhead and present a novel approach to solve this problem. We introduce the reconfiguration state graph which is used to describe dynamic reconfiguration for individual resources and to assess reconfiguration cost. We present new method to map reconfigurable modules to resources such that the reconfiguration cost are small. The method can be applied to both digital circuits and dataflow graphs. We demonstrate that we can exploit the trade-off between resource requirements and reconfiguration cost by a unique high-level synthesis tool. We further discuss how our methodology can be integrated into a design flow for efficient runtime reconfigurable systems.

### 8.1 Introduction

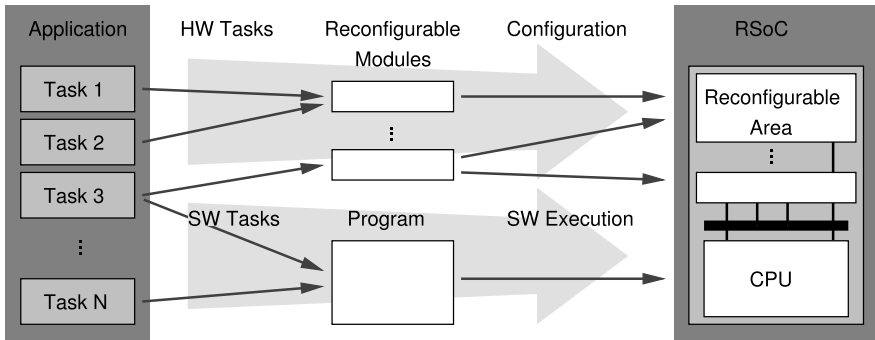
Reconfigurable computing architectures provide a combination of high data processing throughput, similar to ASICs, and the flexibility of a software processor. In such architectures an array of functional units provides the resources to perform many operations in parallel, thus enabling high throughput. The function of the resources and the data transfer between resources are programmable, hence functionality is customized during deployment, after device fabrication. Whereas in reconfigurable computing systems with one static configuration any anticipated functionality must be provided statically, in systems with partial dynamic reconfiguration

---

Markus Rullmann · Renate Merker  
Technische Universität Dresden, Dresden, Germany, e-mails: [markus.rullmann@gmx.de](mailto:markus.rullmann@gmx.de),  
[renate.merker@tu-dresden.de](mailto:renate.merker@tu-dresden.de)

more efficient realizations are possible because the functionality can be adapted at runtime to the requirements.

The design of reconfigurable systems-on-a-chip (RSoC) often follows the principles shown in Fig. 8.1. The application consists of a number of tasks. The tasks are partitioned into hardware tasks (HW tasks) and software tasks (SW tasks). In the final system implementation, the software tasks constitute the software program which is run on the RSoC's CPU. The hardware tasks are implemented as reconfigurable modules, which are loaded into the reconfigurable areas of the RSoC during runtime. This method is called partial dynamic reconfiguration. The partitioning of the tasks is crucial for the system performance and the requirements of reconfigurable resources. The reconfiguration between different reconfigurable modules induces a high runtime overhead. In order to prevent frequent dynamic reconfiguration we introduced multimode reconfigurable modules. A multimode module can perform the computation for different tasks without reconfiguration.



**Fig. 8.1** Application partitioning into HW tasks and SW tasks. The tasks are run on the RSoCs reconfigurable area and the CPU.

We investigate the problems of reconfiguration overhead from the device architecture point of view. Consistently with current methodologies we assume that the design functionality is partitioned into modules, but we make two important extensions: (1) modules are not reconfigured completely but based on individual resources. We call this *fine grain reconfiguration*. (2) one module can provide several functions for an application, implemented in a *multimode circuit*.

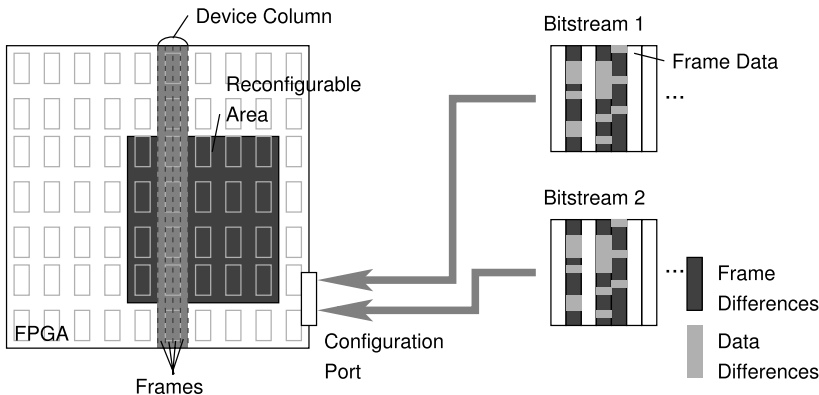
In this chapter we describe new models, methods and tools that reduce the overhead associated with the dynamic reconfiguration of reconfigurable modules. The methods use a new high-level synthesis (HLS) approach to generate reconfigurable modules that execute the HW tasks. At first we motivate our approach by a short description of current limitations of partial reconfiguration in Sect. 8.2. In Sect. 8.3 we explain general reconfigurable module architecture that is produced by our HLS tool. The models used in the reconfiguration cost assessment are introduced in Sects. 8.4 and 8.5. In Sect. 8.6 we describe our HLS approach. Experimental results obtained with our HLS tool are provided in Sect. 8.7. Finally in Sect. 8.8, we

present our work in a broader context of general system design issues. Here we describe how our methods should be integrated into an FPGA system design flow and give further references to related work.

## 8.2 Motivation

Partial dynamic reconfiguration in FPGAs is usually associated with a module based design approach, see Fig. 8.1. At first, the designer defines a *reconfigurable area* on the device. Second, he implements the reconfigurable tasks as modules that can be loaded on the reconfigurable area. At runtime the resources in the reconfigurable area are reconfigured to enable different modules.

Using standard methodology, the reconfiguration cost of the implementation depends on the size of the reconfigurable area, cf. Fig. 8.2. Each reconfiguration is performed by loading a partial *bitstream* into the device. The configuration bitstream itself is composed of *configuration frames* that contain any data needed to configure the entire reconfigurable area. A configuration frame is the smallest reconfigurable unit in a device; the size of a frame and configurable logic that is associated with each frame depends on the FPGA device. Because the standard bitstreams contain all data for a reconfigurable area, the size of these bitstreams is large, typically hundreds of kilobyte. The *configuration port* of the device has only a limited bandwidth. Together, this leads to configuration times in the order of some hundred microseconds. As a conclusion, configuration data becomes often too large for on-chip storage and frequent reconfiguration leads to considerable runtime overhead.



**Fig. 8.2** Illustration of module-based partial reconfiguration.

If the properties of reconfiguration data are analyzed in detail, it can be observed that the data does not differ completely between reconfigurable modules: (1) some of the reconfiguration frames are equal in two designs and (2) the data in two frames

that configure the same part of the device frequently exhibit only a few different bytes (see frame/data differences in Fig. 8.2). This has implications on the device reconfiguration at runtime. After the initial device configuration, the reconfigurable area is always in a known configuration state. When a new configuration must be established on this area, a new bitstream is used to program the associated device resources. In the ideal case, the reconfiguration programs only the device resources that need a different configuration. Currently the granularity of the reconfiguration is limited by the configuration frame size. For an efficient partial reconfiguration, only configuration frames that contain new configuration data are used to program the reconfigurable area. This is only possible if the current configuration and the frame-based differences are known at runtime.

In the latest FPGA generations (Virtex4, Virtex5) the size of configuration frames has been reduced considerable, which enables a more fine-granular reconfiguration. Also, the bandwidth of the configuration port has been increased, which enables faster reconfiguration. Nevertheless the drawback of existing design flows persists: reconfiguration overhead depends solely on the reconfigurable area, but not on the contents of the reconfigurable modules.

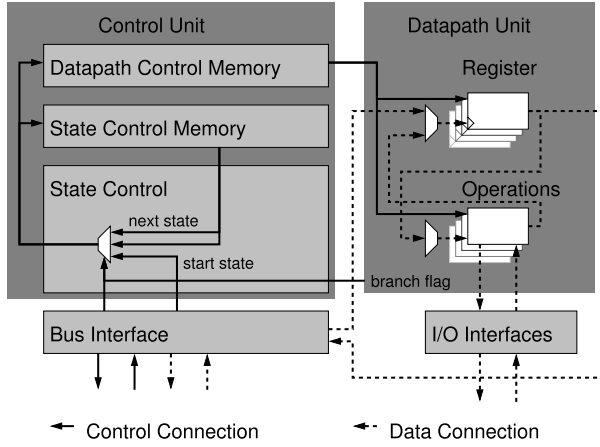
The configuration data themselves are the result of the circuit design and the place and route tools. The configuration data of two modules can become very similar if the initial design exhibits a similar circuit structure and the tools place and route the circuits similarly. In this chapter we describe how the similarity between reconfigurable modules can be increased in order to reduce the differences in configuration data and therefore reduce reconfiguration cost.

### 8.3 Reconfigurable Module Architecture and Partitioning

The HW tasks are implemented in the reconfigurable modules. In these modules, the tasks are executed in parallel to the CPU and other modules in the RSoC. Therefore, each module needs its own local execution control that runs the task within the module. The HW task execution is started by the software program running on the CPU. The software also receives data and status information from the HW task. The HW task functionality is realized by using computational resources and a state machine that is implemented in the module. The state machine creates a sequence of control signals for the resources in order to execute a task.

Here we describe the logical architecture of a reconfigurable module. Our high-level synthesis tool automatically generates modules with such an architecture. The modules consist of a datapath unit, a control unit, control memories, a bus interface and additional I/O interfaces. The general structure is shown in Fig. 8.3.

The control unit operates as a state machine that is split into a state control memory, which holds the sequence of states and a datapath control memory which stores the control sequence for the datapath. The datapath unit contains registers as storage elements for variables, operations to process data, and multiplexers to control the dataflow on the datapath connections. The control signals of these units are driven



**Fig. 8.3** Reconfigurable module architecture used by the HLS tool.

from the datapath control memory. The operations can realize either math and logic functionality or they are used as an interface to external I/O. External I/O can realize bus master accesses and access to FIFOs, memories and other periphery.

The amount of reconfiguration can be chosen if the module is partitioned into static and dynamic sub-modules accordingly. With existing methods, the whole module would be implemented as one monolithic reconfigurable module. Instead we propose to keep the bus and I/O interfaces static and to reconfigure the contents of the control memory and the resources of the datapath independently. Thus reconfiguration can be used for a subset of resources, depending on the configuration differences between modules.

## 8.4 Reconfiguration State Graph

The current configuration and the partial reconfiguration of an FPGA must be managed at runtime. A reconfiguration state graph (RSG) [6] is used to model the partial reconfiguration. The RSG defines the configurations and the reconfiguration formally. The RSG describes the different configurations available and for each reconfiguration it can be decided what resources must be reconfigured. It provides a framework for reconfiguration management and reconfiguration overhead assessment.

The RSG is defined as a digraph  $G(\mathcal{N}_T, \mathcal{E}_T)$  where the set  $\mathcal{N}_T$  of nodes  $i$  represents the reconfigurable modules and the set  $\mathcal{E}_T$  of edges  $e = (i, j)$  represents the reconfiguration from reconfigurable module  $i$  to reconfigurable module  $j$ . With  $\mathbf{d} : \mathcal{N}_T \mapsto \mathcal{D}^m$  for each reconfigurable module  $i$  a configuration  $\mathbf{d}(i) = (d(i)_1, \dots, d(i)_m)$  is given. The set  $\mathcal{D}$  denotes possible configurations of a resource. The elements  $d(i)_k, k = \{1, \dots, m\}$  describe the configurations of the

smallest independent reconfigurable resources  $k$  in a device. The reconfiguration  $\mathbf{r} : \mathcal{E}_T \mapsto \{0, 1\}^m$  describes for each edge  $e = (i, j) \in \mathcal{E}_T$  which resources  $k$  in a device must be reconfigured in order to change a current configuration  $\mathbf{d}(i)$  to a new configuration  $\mathbf{d}(j)$ . Hence, if  $\mathbf{d}(i)_k \neq \mathbf{d}(j)_k$  (i.e. reconfiguration of resource  $k$  is necessary)  $\mathbf{r}(e)_k = \mathbf{r}((i, j))_k = 1$  and if  $\mathbf{d}(i)_k = \mathbf{d}(j)_k$  (i.e. reconfiguration of resource  $k$  is not necessary)  $\mathbf{r}(e)_k = \mathbf{r}((i, j))_k = 0$ .

The reconfiguration overhead can now be computed on the basis of the RSG. We assume that each reconfiguration is performed once. The average number of resources that are reloaded if reconfiguration occurs is given by:

$$c_{rc} = \frac{1}{|\mathcal{E}_T|} \sum_{e \in \mathcal{E}_T} \sum_{k=1}^m w_1(k) \mathbf{r}(e)_k, \quad (8.1)$$

where the function  $w_1(k)$  yields the cost for the reconfiguration of element  $k$ . We assume that the reconfiguration time is proportional to the weighted sum of reconfigured resources and therefore  $c_{rc}$  is called average reconfiguration time.

The RSG model is further illustrated in Example 8.1.

## 8.5 Module Mapping and Virtual Architecture

The RSG model describes only how the reconfiguration overhead is affected by the configuration data in  $\mathbf{d}$ . For any module functionality there exist many possible realizations, where each yields a different configuration  $\mathbf{d}$ . We have developed a method to map the original HW tasks to the reconfigurable modules such that the differences between module configurations are minimized. As a result, the average reconfiguration time is reduced, too.

We observe that functionality of a module is given as a structural representation until the module is finally translated to binary configuration data. It is not possible to describe the structural representation directly as a configuration  $\mathbf{d}$ , because the functionality is not directly related to fixed resources. Here, we introduce a model that enables us to provide a configuration  $\mathbf{d}(i)$  for any structural representation of a module  $i$ . First, we define the structural representation formally as a digraph and then we map the digraph to a *virtual architecture* (VA) in order to derive the configuration  $\mathbf{d}(i)$ . For any such mapping we can therefore compute the reconfiguration overhead and thus can optimize the mapping accordingly, without creating bitstreams for different mappings.

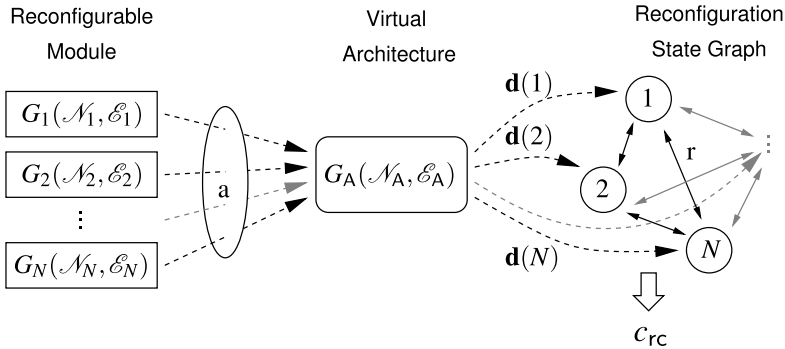
A module  $i \in \mathcal{N}_T$  is represented by a digraph  $G_i(\mathcal{N}_i, \mathcal{E}_i)$  where the set  $\mathcal{N}_i$  of nodes defines the functions used by a module and the set  $\mathcal{E}_i$  of edges defines the data transfer between the functions. The resource configuration required by a node is assigned by the function  $\mathbf{l} : \mathcal{N}_i \mapsto \mathcal{D}$ . Note that the digraph can describe structural representations at several levels in the design flow, e.g. it can describe dataflow graphs, synthesized digital circuits etc. The digraphs  $G_i, i \in \mathcal{N}_T$  are called *input graphs*.

The input graphs of all modules  $i$  are mapped to a VA. A VA is defined as a digraph  $G_A(\mathcal{N}_A, \mathcal{E}_A)$ . The nodes  $\mathcal{N}_A$  denote reconfigurable resources and the edges  $\mathcal{E}_A$  denote reconfigurable interconnect. Furthermore we define an allocation  $a : \mathcal{N}_i \mapsto \mathcal{N}_A$ , which maps the node of any input graph to a resource in the virtual architecture. As a by-product the edges from the input graphs are mapped to the VA, too. The most important feature of the allocation is that the different input graphs are mapped to a common context, and hence, their configuration can be compared to each other.

Now we specify for a module  $i$  the configuration of the VA, i.e. the configuration of the resources  $n \in \mathcal{N}_A$  and of the interconnects  $e \in \mathcal{E}_A$  that realizes the module on the VA. The configuration of these  $m = |\mathcal{N}_A| + |\mathcal{E}_A|$  elements is given for module  $i$  by  $\mathbf{d}(i) = (d(i)_1, \dots, d(i)_m)$  as follows:

- The configuration of resource  $n_k \in \mathcal{N}_A, k \in \{1, \dots, |\mathcal{N}_A|\}$  is specified by  $d(i)_k = l(n_i)$  if a node  $n_i \in \mathcal{N}_i$  exists with  $a(n_i) = n_k$ , otherwise  $d(i)_k = 0$ .
- The configuration of interconnect  $e_k \in \mathcal{E}_A, k \in \{|\mathcal{N}_A| + 1, \dots, m\}$  is given with  $d(i)_k = 1$ , if an edge  $e_i \in \mathcal{E}_i$  is mapped to an edge  $e_k \in \mathcal{E}_A$ , otherwise  $d(i)_k = 0$ .

In summary, we provided a formal definition of the module's structural representation and an abstract reconfigurable architecture model. An allocation describes how the module is mapped to the architecture model. In addition, the input graphs, the allocation, and the VA define the configuration in the RSG model. Thus, we can calculate the reconfiguration cost. The model is illustrated in Fig. 8.4. In order to reduce reconfiguration cost, we are interested in an allocation that minimizes reconfiguration cost.



**Fig. 8.4** Mapping of the structural representations of modules to a VA and its relationship to the RSG model.

*Example 8.1.* Consider three modules 1, 2, and 3 represented by the input graphs  $G_1, G_2, G_3$  shown in Fig. 8.5. The nodes  $n_i \in \mathcal{N}_i$  of the input graphs are labeled with their respective configuration  $l(n_i) = f_i$ , e.g. node  $n_1$  requires a configuration  $l(n_1) = f_1$  of the resource in order to realize the required functionality.

The VA graph is depicted in Fig. 8.5, too. The VA  $G_A(\mathcal{N}_A, \mathcal{E}_A)$  is given by the elements  $\mathcal{N}_A = \{n'_1, n'_2, n'_3, n'_4\}$  and  $\mathcal{E}_A = \{e_5, e_6, e_7\}$ .

For each node  $n_i$  in the input graphs  $G_1, G_2, G_3$  the allocation to a node  $n'_k$  of the VA is shown, i.e.  $a(n_i) = n'_k$ . The allocation of an edge  $e_i = (n'_{i_1}, n'_{i_2}) \in \mathcal{E}_i$  results directly from the allocation of the nodes  $n'_{i_1}, n'_{i_2}$ . For example, the edge  $(n_1, n_2) \in \mathcal{E}_1$  is allocated to edge  $e_6 = (a(n_1), a(n_2)) \in \mathcal{E}_A$  of the VA.

The configuration  $\mathbf{d}(i)$  of the VA that realizes the functionality required by module  $i$  depends on the allocation of nodes  $\mathcal{N}_1$  and edges  $\mathcal{E}_1$ . The configuration  $\mathbf{d}(i) = (d(i)_1, \dots, d(i)_7)$  describes the configuration of the resources and interconnects in the following order:  $n'_1, n'_2, n'_3, n'_4, e_5, e_6, e_7$ . For example the allocation of module 1 yields the configuration  $\mathbf{d}(1) = (l(n_1), l(n_2), l(n_3), 0, 0, 1, 1) = (f_1, f_2, f_3, 0, 0, 1, 1)$ .

The RSG model related to the input graphs  $G_1, G_2, G_3$  contains the modules  $\mathcal{N}_T = \{1, 2, 3\}$  and the reconfigurations between the modules  $\mathcal{E}_T = \{(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)\}$ , cf. Fig. 8.5. The reconfiguration  $\mathbf{r}((i, j))$  is derived from the configurations  $\mathbf{d}(i), \mathbf{d}(j)$  of the modules  $i, j$ . Consider the configurations  $\mathbf{d}(1) = (f_1, f_2, f_3, 0, 0, 1, 1)$  and  $\mathbf{d}(2) = (f_5, f_6, 0, f_4, 1, 1, 0)$ : the related reconfiguration yields  $\mathbf{r}((1, 2)) = (1, 1, 1, 1, 1, 0, 1)$  because the configuration of all elements  $k$  differs except for  $k = 6$ . The edge  $e_6$  is allocated by both configurations.

The reconfiguration for the full RSG is as follows:  $\mathbf{r}((1, 2)) = \mathbf{r}((2, 1)) = (1, 1, 1, 1, 1, 0, 1)$ ,  $\mathbf{r}((1, 3)) = \mathbf{r}((3, 1)) = (1, 1, 1, 1, 1, 0, 0)$ ,  $\mathbf{r}((2, 3)) = \mathbf{r}((3, 2)) = (1, 1, 1, 1, 0, 0, 1)$ . If we assume a unit weight for all reconfigurable elements, i.e.  $w_1(k) = 1, k \in \{1, \dots, 7\}$ , then the reconfiguration cost (Eq. (8.1)) evaluate to:

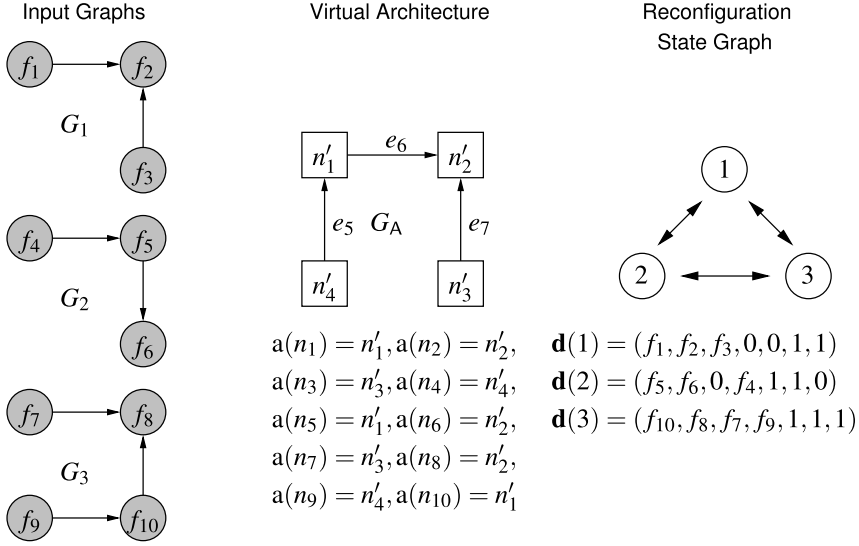
$$c_{rc} = \frac{1}{6}(6 + 6 + 5 + 5 + 5 + 5) = 5\frac{1}{3}.$$

## 8.6 High-Level Synthesis of Reconfigurable Modules

In this section we describe our methodology that enables us to compute allocations that lead to minimal reconfiguration cost. More specifically we implemented the methodology into a high-level synthesis (HLS) tool. The tool receives the functionality which must be implemented in the reconfigurable modules as ANSI-C like source code. The source code is compiled into control dataflow graphs (CDFG) first, one for each C-function. The CDFGs are considered as input graphs to our allocation problem. The tool performs four essential HLS steps: (1) for each node in the CDFG an appropriate resource type is chosen, (2) scheduling assigns an execution time to each node, (3) each node is allocated to a resource instance in the datapath which is described as a VA, and (4) architectural synthesis creates the submodules according to the module architecture, i.e. the control unit and the datapath unit.

HLS provides the ideal abstraction level to describe reconfigurable modules. From the designer's point of view, it is much easier to use C-like descriptions of the algorithm: they can be modified more easily and they can be integrated into system-





**Fig. 8.5** Input graphs  $G_1, G_2, G_3$  for Example 8.1. The input graphs are mapped to the VA  $G_A$  provides a common reference for the image graphs.

level simulations. It is also possible to adapt the hardware/software partitioning late in the design process because functions can be moved to the software processor or to hardware modules with little effort. On the tool side, the high-level descriptions provide a great deal of freedom to map the functionality to a datapath. In our work we exploit this freedom in order to generate reconfigurable modules with small reconfiguration overhead. More specifically, we investigated several methods to choose the resource types in HLS step 1, and we implemented a resource allocation method that takes advantage of our VA model to solve HLS step 3.

Both reconfiguration overhead and resource overhead in HLS is avoided by resource sharing. In intra-module resource sharing, several CDFG nodes of one module are mapped to the same resource instance in the VA, thus reducing resource overhead. In inter-module resource sharing, several CDFG nodes of multiple modules are mapped to the same resource instance in the VA. As a result, the resources are used in several modules and are not reconfigured between those modules.

In the following we describe the HLS steps 1 and 3 in more detail.

### 8.6.1 Resource Type Binding

In HLS step 1, Resource type binding can either enable or disable the reuse of VA resources in step 3. In extension to the input graphs defined previously, the CDFG nodes represent fixed operations or variables. An operation (or variable) can only

be executed (or stored) on selected resource types. During resource type binding, one resource type is chosen for each CDFG node. In the resource instance binding step, the allocation can only be chosen such that each node is allocated to a resource instance that is of the previously specified type. The resource type binding defines the potential reuse of nodes and edges in the VA because it enables or disables resource sharing possibilities between nodes. We investigated several different type binding strategies in our research in order to see how much the strategy affects the final results. We assume that many nodes can be bound to resource types of different complexity, which effects resource sharing: e.g. an addition could be bound to a simple adder-resource or to a complex ALU-resource. The different type binding strategies are stated below:

- (a) *Minimum Cost Resource Type.* Here, we choose a resource type for each node independently with the objective to select the least costly one, e.g. in terms of FPGA resources.
- (b) *Minimum Number of Resource Types.* The resource types are chosen such that the number of different resource types becomes minimal. As a result, it is possible that there are fewer resource instances in the VA because resources can be shared more often. For instance two nodes may realize two different functions, but if they are mapped to the same resource type, they may share a resource instance in the datapath. The number of resource types can be minimized either over each task independently or for all tasks at once.
- (c) *Minimum Number of Interconnect Types.* The data transfers, indicated by edges in the CDFG, are mapped to interconnects in the VA. Although the exact interconnect is not known during type binding, it is already possible to determine if two edges may share an interconnect or not. Two edges can share an interconnect if the they are mapped to the same interconnect type. The interconnect type is defined by the resource types were the source node and the drain node of an edge are mapped to. The minimization of interconnect types targets specifically the reuse of interconnect in the datapath. As above, the number of resource types can be minimized either over each task independently or for all tasks at once.

The effect of the different strategies for resource type binding will be discussed for the benchmarks provided in Sect. 8.7.

### 8.6.2 Resource Instance Binding

After a resource type has been selected for each operation and the operation has been determined in HLS step 2, the operations must be allocated to specific resource instances. In step 3, the HLS tool allocates the CDFG nodes to resource instances and derives the datapath interconnect to realize the data transfer between nodes. This step is based on the transformation of the input graphs to the VA presented in Sect. 8.5. In our tool, we employ a heuristic optimization method that is

based on simulated annealing [7] in order to find an allocation that minimizes a cost function  $c$ . The cost function is a weighted sum of several cost parameters of the datapath. Thus the datapath can be optimized to achieve minimal reconfiguration cost (for resources and interconnect), resource use, interconnect overhead, and dataflow multiplexers.

Dataflow multiplexers are introduced into the datapath in order to realize resource sharing. Assume that multiple nodes of one CDFG are allocated to the same resource. However, the data supplied to the resource originates from different resources. For each computation, a dataflow multiplexer connects the output of the resource which provide the data to the shared resource input. The dataflow multiplexers are controlled by the control unit. The node allocation defines the resource sharing and thus the interconnect structure and the dataflow multiplexers.

Our HLS tool can generate multimode circuits in order to reduce reconfiguration cost. For a set of tasks it can be chosen, which tasks are implemented in the same reconfigurable module. Hence we can take advantage of inter-module resource sharing within one configuration, which reduces resource overhead in multimode circuits, and between different configurations, which reduces overhead for dynamic reconfiguration.

The resource instance binding step works as follows: The scheduling algorithm provides information how many resource instances are required. This defines the number of resources in the VA. Further, the scheduling algorithm provides an initial solution which assigns each CDFG node to a node in the VA. The initial solution is modified iteratively by the simulated annealing algorithm in order to improve the initial solution.

In simulated annealing a current solution is modified iteratively. At first, the current solution is slightly modified to gain a new solution. Then, the cost function for the new solution is calculated. Depending on the state of the algorithm and the cost of the new solution, the new solution is either accepted and becomes the current solution or the new solution is discarded.

In our tool, the new solution is derived from the current solution by a random permutation of the allocation. Subsequently, the interconnect structure in the VA is derived as well as the dataflow multiplexers. The permutation must observe the constraints imposed by the scheduling: Any resource instance can only be allocated by one CDFG node at any cycle at runtime.

For the permutation, a node  $n \in \mathcal{N}_i$  is selected randomly and the allocation  $a(n) = r$  is changed to a randomly selected resource  $r'$ , i.e.  $a(n) = r'$ . Vice versa any node  $n' \in \mathcal{N}_i$  which is already allocated to the resource  $r'$  and which is in conflict with the new allocation of  $n$ , will be allocated to the previous allocation of  $n$ , i.e.  $a(n') = r$ . Thus, starting from a valid initial solution we permute the solutions iteratively such that each solution remains valid.

The cost function used in the simulated annealing algorithm is composed of several components. The basis for all components is the allocation to the VA and the RSG model. In the following we derive the relevant computations. The resource cost for a module  $i \in \mathcal{M}$  is calculated from the input graph  $G_i$  and the allocation  $a$  (cf. Sect. 8.5).

Here we assume that the resource instances  $n_k$  in the VA are either in use ( $d(i)_k = 1$ ) or unused ( $d(i)_k = 0$ ), similar to the interconnect configuration. Each resource instance and each interconnect is associated with a cost factor  $w_2$  that represents either the number of used FPGA slices for a resource instance or the bus width of an interconnect. Hence, the resource cost for a module  $i$  are given by:

$$c_{\text{res}}(i) = \sum_{k=1}^m w_2(k) d(i)_k. \quad (8.2)$$

The resource cost  $c_{\text{res}}(i)$  do not include the cost for the dataflow multiplexers. A single dataflow multiplexer switches between all edges running into the same input of node  $n \in \mathcal{N}_A$ . For a module  $i$ , the set  $\mathcal{E}_{A,n}$  of such edges are given by  $\mathcal{E}_{A,n} = \{e_k \in \mathcal{E}_A : e_k = (n', n), n' \in \mathcal{N}_A \wedge d(i)_k = 1\}$ . The resource cost caused by the dataflow multiplexers are now computed as:

$$c_{\text{mux}}(i) = \sum_{n \in \mathcal{N}_A} w_3(|\mathcal{E}_{A,n}|), \quad (8.3)$$

where  $w_3(x)$  yields the resource cost of an  $x$ -to-1 multiplexer.

The reconfiguration cost are already defined in Eq. (8.1). The overall cost function for the simulated annealing algorithm is given by:

$$c = \frac{1}{|\mathcal{N}_T|} \sum_{i \in \mathcal{N}_T} c_{\text{res}}(i) + \frac{1}{|\mathcal{N}_T|} \sum_{i \in \mathcal{N}_T} c_{\text{mux}}(i) + c_{\text{rc}}. \quad (8.4)$$

### 8.6.3 Control Generation

The scheduling of nodes from the input graphs is computed in HLS step 2. Together with the allocation of those nodes, the tool generates the contents of the datapath control memory and the state control memory. The resources in large datapaths are often not used in every control state, i.e. the datapath control memory can be underutilized. Therefore we combined our tool with the approach described in Chap. 15. We implemented a greedy algorithm that translates the contents of the datapath control memory to multi-context tables. In [18] we have shown that this method can reduce the storage overhead for datapath control information significantly. We further proposed two possible extensions to current FPGA architectures that enable a very efficient and yet flexible integration of multi-context tables into FPGAs.

In this section we have provided a brief overview of a HLS tool for improved partial dynamic reconfiguration. We have illustrated that there exists a large space for optimizations that increase the similarity of reconfigurable modules. The aim of these optimizations are—besides conventional optimization targets time and area—the reduction of reconfiguration overhead.

## 8.7 Experiments

In this section we discuss the efficiency of our HLS approach on a set of benchmarks. The benchmark set has been implemented with different combinations of type binding and instance binding methods presented in the previous section. The results obtained depend on the combination of the HLS steps 1 and 3. This allows us to analyze the quality of results for each combination as well as the costs in terms of tool runtime. First we describe the experimental setup and then we discuss the results and draw some conclusion concerning the design of reconfigurable modules.

### 8.7.1 Experimental Setup

In Sect. 8.6 we described several options to perform resource type binding. The resource instance binding can also be performed with different objectives: with the help of the weight functions  $w_1$ ,  $w_2$ , and  $w_3$  we are able to set up different cost functions that are used by the simulated annealing algorithm as a cost function. The different objectives are used to optimize the HW task implementations for different scenarios within a common framework. The implementation scenarios describe how the modules are implemented in the RSoC. Thus we can compare non-reconfigurable and reconfigurable solutions.

**Resource Type Binding Methods** The resource type binding methods discussed in Sect. 8.6.1 (a)–(c) are used in our experiments as follows:

1. Minimum cost resource type,
2. Minimum number of resource types for each module individually,
3. Minimum number of resource types and interconnect types for each module individually,
4. Minimum number of resource types over all modules,
5. Minimum number of resource types and interconnect types over all modules,

where the cost for interconnect types is not optimized independently but in combination with the cost for resource types.

**Resource Instance Binding Methods** Further we investigated several optimization targets during HLS step 3 that were combined with the different type binding methods. The different optimization targets for the instance binding are as follows:

1. Minimum average resource and interconnect cost of the tasks individually,
2. Minimum resource and interconnect cost for all tasks merged into one datapath,
3. Minimum average resource reconfiguration cost,
4. Minimum average resource and interconnect reconfiguration cost.

**Implementation Scenarios** In this work we compare the non-reconfigurable and reconfigurable implementation of HW tasks, both for existing methods and for our new methodology. The following implementation scenarios are considered:

- A: *static, parallel implementation*. Classic implementation, where each HW task is implemented as an individual module, which is placed statically on the device. The HW tasks can be executed concurrently.
- B: *static, sequential implementation*. Our method allows that several HW task are implemented in a multimode module. The tasks can share resources but can not operate in parallel.
- C: *reconfiguration without reuse of resources*. Classic module based reconfiguration. Each HW task is assigned to an individual module which is completely dynamically reconfigured.
- D: *reconfiguration with reuse of resources*. In this scenario our new reconfiguration model is applied. It is assumed that only those resources and interconnect within the datapath are reconfigured that are different between configurations.

With the data obtained from our benchmarks we want to investigate, which resource type and which resource instance binding strategies lead to the best solution for a scenario. Further, we will show that the scenarios B and D, which are available through our methodology, are superior to previous concepts A and C.

**Benchmark Characteristics** The chosen benchmarks consist of several task sets. Each task set contains tasks that might be used in a real reconfigurable system. The tasks within one set are assumed to be reconfigured against each other. Thus the tasks provide a good example on how our methodology can be employed in practice. This kind of tasks can be found in many similar work on HLS. Here with give a short summary of the tasks functionality and complexity by (number of tasks, total number of nodes for all tasks). The benchmark ADPCM (2 tasks, 280 nodes) contains an ADPCM encoder and decoder from the MediaBench suite [8]. EDGE (3 tasks, 422 nodes) contains three different Sobel edge detection filters: a combined horizontal and vertical filter, a horizontal only, and a vertical only filter. JPEG\_DCT (2 tasks, 613 nodes) consists of tasks that perform an integer based forward discrete cosine transform (DCT) and a task for the backward transform. Both tasks are also taken from MediaBench. The JPEG\_DCT represents the most complex task set in terms of operations per input graph. Finally the RGB\_YUV (2 tasks, 84 nodes) describes a color conversion from RGB color space to the YUV color space and vice versa, this function is used in many image and video coding applications.

## 8.7.2 Benchmark Results and Discussion

Our HLS tool has been used to implement the HW tasks of the benchmarks according to the different scenarios, by using different resource type and instance binding methods. Fig. 8.6 shows the results obtained for our benchmarks using the scenarios A–D. For each scenario we present the results for the best overall combination of resource type and resource instance binding method. For each benchmark, the results are labeled on the x-axis as follows: *scenario: resource type binding method, resource instance binding method*.

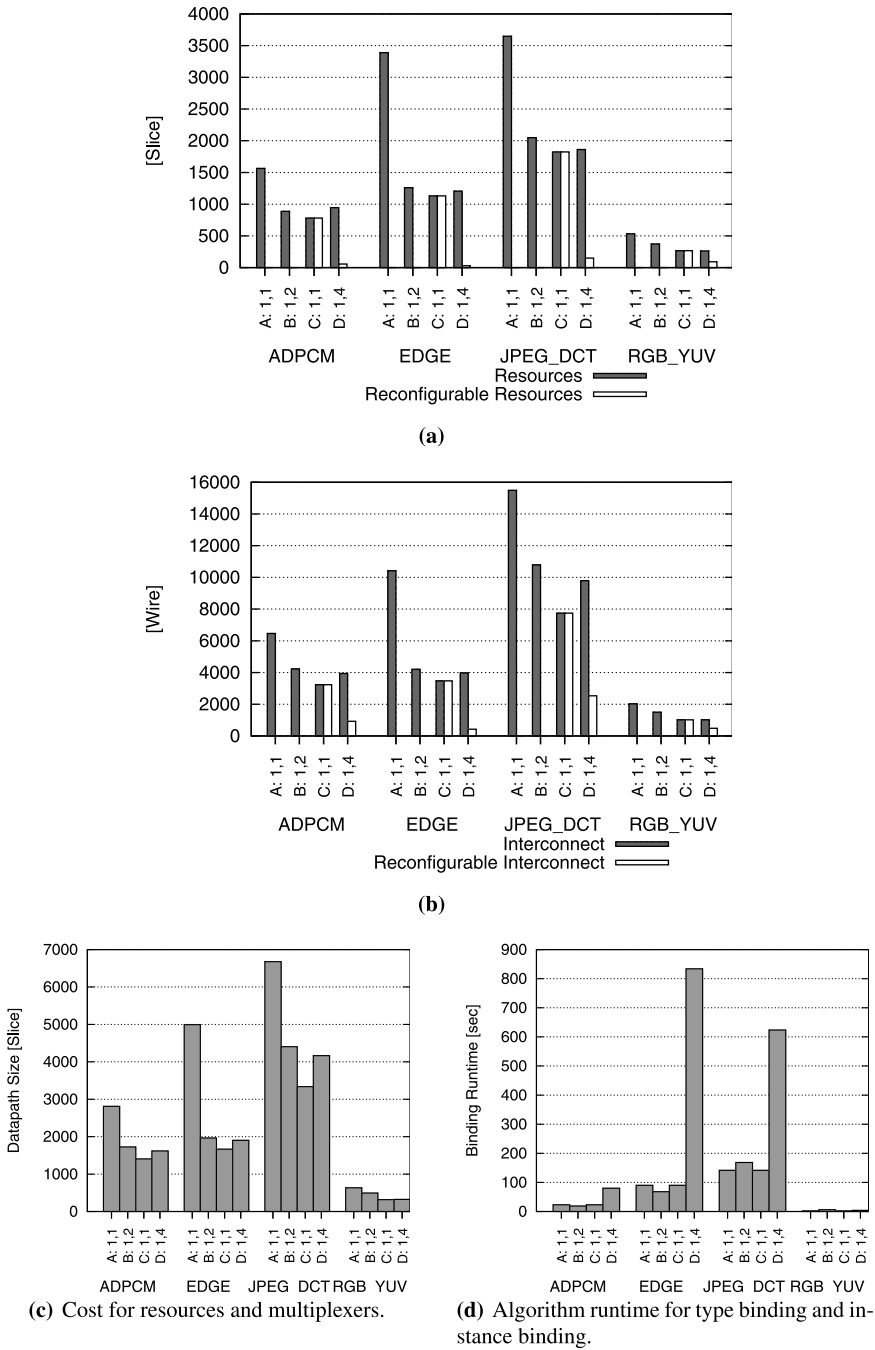


Fig. 8.6 Comparison of results obtained with selected binding methods for scenarios A–D.

In Fig. 8.6(a) the average amount of module resources used for operations and storage in the datapath of a module is shown. The bright bars show the average amount of reconfigurable resources in each module. Similarly, the average number of interconnect wires used in the datapath of a module are depicted in Fig. 8.6(b). Here, the bright bars show the average amount of reconfigurable interconnect. Thus, the average resource cost  $\frac{1}{|\mathcal{N}_T|} \sum_{i \in \mathcal{N}_T} c_{\text{res}}(i)$  and the average reconfiguration cost  $c_{\text{rc}}$  are shown separately for resources and interconnect in Fig. 8.6(a, b). The average datapath size in terms of resources, which includes both operation/storage and multiplexer cost is shown in Fig. 8.6(c). The tool runtime for the type and instance binding algorithms is depicted in Fig. 8.6(d).

In our experiments we found that the most straightforward resource type binding method (1) achieves the best overall results in the final implementation for all scenarios. While the type binding method has a considerable effect on the resource sharing possibilities, the differences are negligible in the final datapath implementation. We suspect that there are always good resource sharing possibilities, because the high-level description uses only a limited set of different operations. We found that the resource instance binding method has a much more severe effect. For the classic scenarios A, C we choose instance binding method (1) because in both scenarios, the modules are implemented independently. For scenario B we found that instance binding method (2) performs best, because only in this case the resource sharing between tasks that are implemented in one module is exploited. Finally for scenario D we could show that instance binding method (4) performs best in terms of reconfiguration cost, because both resources and interconnect reconfiguration are targeted by the optimization.

The scenarios A and C represent the conventional approaches to implement HW tasks on FPGAs. In scenario A, a static configuration contains all reconfigurable module, which may lead to a high resource and interconnect overhead but enables a parallel execution of tasks. As shown in our benchmarks, the resource and interconnect requirements are reduced drastically if the modules are dynamically reconfigured. At the same time, reconfiguration of all resources and interconnects used by a module causes a high overhead.

The newly introduced scenario B, which implements static, merged datapaths provides an attractive trade-off between the scenarios A and C. Scenario B requires much less resources than scenario A because resources are shared between HW tasks. Further scenario B causes no reconfiguration cost, but scenario C employs partial reconfiguration of the module. Nevertheless, scenario B requires more resources than scenario C, because the flexibility is gained with additional operations and dataflow multiplexers, which result in a datapath with more, temporarily unused resources. Furthermore, it is interesting to note that the differences in the resource allocation for scenarios B and C are small, cf. Fig. 8.6(c).

In scenario D, the implemented datapaths are optimized for maximum similarity and hence, for minimal reconfiguration cost in terms of resource and interconnect resources. Scenario D demonstrates how much the datapaths can be optimized, such that they differ in only few resources and interconnects. For resources, the differences are less than 10% and for interconnects, the differences are less than 26% in



most cases, which is a significant reduction compared to the full reconfiguration of the datapath in scenario C. However, in scenario D the resource and interconnect cost were not included in the optimization. Therefore, the datapaths are slightly more costly in terms of resources and interconnects. Actually, the scenarios C and D represent two extremes between area optimization (scenario C) and reconfiguration optimization (scenario D). With the flexible weights  $w_1, w_2, w_3$  in the cost function, we are able to generate intermediate solutions that meet the needs of the overall system.

The runtime of the binding algorithms shown in Fig. 8.6(d) is comparable for the scenarios A–C, but the runtime for scenario D is much higher. The resource type binding has been performed with method (1) in all scenarios. Obviously the resource instance binding requires a much higher optimization effort, when the result is optimized for resource and interconnect reconfiguration cost. From our experience we believe that this optimization is still much more efficient in terms of runtime and results, than a similarity extraction of the final netlist. E.g. benchmark JPEG\_DCT contains a total of 613 nodes in the input graphs for which quality binding must be found. However, at netlist level the similarity extraction must be performed for  $2 \times 1800$  slices, which is much more complex.

The performance of the datapath implementations is very similar. The task execution cycles are equal in all scenarios because the same scheduling is used. The maximum clock frequency differs slightly between the scenarios, but usually less than 10%. However, our optimization does not target the critical path delay directly, it reduces the complexity of dataflow multiplexers instead.

Although the examples presented here have only a limited number of tasks, we discuss the development for an increased number of tasks. As we merge more tasks into one module (scenario B) it is likely that the increase in operation resources is small. However, because the dataflow in the tasks is different, more flexibility in interconnect is needed and the overhead in interconnect and dataflow multiplexers increases. Likewise, if the datapath implementation is optimized for low reconfiguration cost we expect that for more reconfigurable modules the average reconfiguration cost increase, because an efficient inter-module resource/interconnect sharing can not be achieved for many tasks at the same time. Our predictions are supported by an analysis in [11] and the fact that FPGAs, which target maximum flexibility, contain highly reconfigurable resources and very flexible interconnect routing. As a general rule, in FPGAs about 90% silicon area are used for interconnect and only 10% for reconfigurable logic.

Here we suggest the following strategy for a balanced use of our new methodology. HW tasks that are frequently reconfigured against each other should be merged in one reconfigurable module or optimized for low reconfiguration cost. HW tasks or the reconfigurable modules that are not frequently reconfigured must not be optimized for low reconfiguration cost. Thus, the implementation depends on the overall execution behavior of the application. With our methodology it is possible to use the reconfigurable area more efficiently and to reduce the penalty of runtime reconfiguration for the most critical parts of the application.

## 8.8 System Design for Efficient Partial Dynamic Reconfiguration

In this chapter we have presented a concise model to describe reconfiguration on the level of individual resources and interconnect. The cost model is based on the reconfiguration state graph. We introduced the model of a virtual architecture that allows us to assess reconfiguration cost for any structural representation of hardware tasks. The benefits of the model have been demonstrated on several examples, which have been implemented by our high-level synthesis tool. Finally, we describe a possible design flow that takes advantage of our methodology.

The general idea of our method, the reduction of reconfiguration overhead by reducing the differences between reconfigurable modules, is reflected throughout our proposed design flow for reconfigurable systems. In this section we summarize the steps of our design flow and provide references to further work.

In the system design phase, there must be decided how the functionality of the tasks is partitioned into HW tasks and SW tasks. The characteristics of the application and of the reconfigurable HW tasks provide information on how many reconfigurable resources are required and what kind of runtime management should be used. Our tools can aid the decision which HW tasks could be integrated into the same reconfigurable module and provide information on the size of the reconfigurable area.

Tasks that depend on each other can not run concurrently in a system. Therefore they can be either integrated into the same reconfigurable module or in different reconfigurable modules that are configured successively. If those tasks are executed frequently they should be integrated into the same reconfigurable module because then dynamic reconfiguration is avoided. Of course, this is only possible if the multimode module fits on the reconfigurable area. More information on the partitioning problem is given in Chap. 9. An alternative approach is described in Chap. 4: In hyper-reconfigurable hardware it is assumed that a sequence of configurations is known. Next, the sequence is partitioned into hypercontexts which contain reconfigurable resources and resources that are static within the hypercontext. The concept can be interpreted in our HLS context as follows: The sequence of configurations is similar to the sequence of control data supplied to the datapath. For such a sequence it can be derived with the methods described in Chap. 4, which parts of the sequence should be grouped into a reconfigurable module in order to minimize reconfiguration cost.

The RSoC is usually managed at runtime by an operating system (OS), which has been adapted to support dynamic reconfiguration. Examples for such systems are described e.g. in [4, 20, 19, 10]. In [1] we have demonstrated a video-based, realtime region-of-interest-detection application. The application demonstrates the capabilities of our HLS tool and the integration of HW tasks and the ReconOS OS (cf. Chap. 13). The application contains a HW task that runs as an independent thread, parallel to the software application. The HW task sends continuously data to the software application via the ReconOS API. As a hardware platform we use the ESM, cf. Chap. 3.

For design entry, the two major methods are HLS and synthesis from register transfer level (RTL) code. During this design phase it is possible to increase the reconfigurable module similarity significantly by special design practices. A brief overview of our HLS method is presented in this chapter, more details can be found in [16, 15]. We also explored possibilities to increase similarity during RTL design. Here, the designer can describe digital circuits that have an intended similarity [17, 12]. Expert knowledge of the device architecture and synthesis is necessary to achieve good results. Other synthesis methods that are used to reduce reconfiguration cost are described e.g. in [3, 9, 2].

The similarity information, which is required later in the module implementation phase can be provided either by the synthesis tool or it can be derived after synthesis. As discussed before, the HLS tool directly generates the similarity information. In addition we have developed a tool that is able to extract the similarity from generated netlists after synthesis [13].

In an FPGA design flow, the synthesized netlists are mapped to device resources before place and route. During the mapping the netlist elements are assigned to device specific resources, e.g. logic blocks (Slices). In this mapping several netlist elements (logic and interconnect) can be assigned to the same device resource, which may destroy the module similarity or invalidate similarity information. We have described a mapping tool [14] that is able to take advantage of the similarity instead. The mapping tool treats all netlists of reconfigurable modules at the same time and thus can retain the similarity directly. The tool takes the reconfiguration cost model into account in order to improve the mapping result.

For our methodology, existing place and route tools must be extended in order to take advantage of the similarity information. The tools must observe the following constraints: nodes that are allocated to the same resources in the VA must be placed on the same device resource later on. Similarly, edges that are allocated to the same interconnect in the VA must be routed using the same switch box configuration. With existing tools this can only be realized to a limited extend, e.g. by using the guide mode in the Xilinx ISE tools. This method has been used in [17, 12].

Finally, the placed-and-routed reconfigurable modules are transcribed into bitstreams that contain the binary programming data for the device. Because we use partial reconfiguration, the bitstreams contain only data that is relevant to adapt the reconfigurable area to the new module. In the established EAPR design flow, the bitstreams for the module contain the full configuration of a predefined reconfigurable area. The authors in [5] describe a tool that can produce bitstreams which remove the frames from the bitstreams which are static in all modules. We have described another method to create partial bitstreams in [15]. There, the reconfiguration bitstreams are chosen such that minimal reconfiguration time or minimal storage of configuration data is ensured. The method is based on the RSG model described above.

**Acknowledgements** Supported by DFG grant ME 1625/3-3, project *Development of Methods and Tools for the Minimization of Reconfiguration Cost*, as part of the Priority Programme 1148, *Reconfigurable Computing Systems*. We would like to thank all members of the Priority Programme

1148 for the fruitful discussions at the project meetings and for the close collaboration within the programme.

## References

1. Angermeier, J., Majer, M., Teich, J., Braun, L., Schwalb, T., Graf, P., Hubner, M., Becker, J., Lubbers, E., Platzner, M., Claus, C., Stechele, W., Herkersdorf, A., Rullmann, M., Merker, R.: Spp1148 booth: Fine grain reconfigurable architectures. In: International Conference on Field Programmable Logic and Applications (FPL 2008), Heidelberg, Germany, p. 348 (2008)
2. Aravind, D., Sudarsanam, A.: High level—application analysis techniques & architectures—to explore design possibilities for reduced reconfiguration area overheads in FPGAs executing compute intensive applications. In: Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, pp. 158–158 (2005)
3. Boden, M., Fiebig, T., Reiband, M., Reichel, P., Rulke, S.: Gepard—a high-level generation flow for partially reconfigurable designs. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI'08), pp. 298–303 (2008)
4. Brebner, G.: A virtual hardware operating system for the xilinx 6200. In: Field-Programmable Logic, Smart Applications, New Paradigms and Compilers. LNCS, vol. 1142, pp. 327–336. Springer, Berlin (1996)
5. Claus, C., Müller, F.H., Zeppenfeld, J., Stechele, W.: A new framework to accelerate Virtex-II pro dynamic partial self-reconfiguration. In: IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007, pp. 1–7 (2007)
6. Heron, J., Woods, R., Sezer, S., Turner, R.: Development of a run-time reconfiguration system with low reconfiguration overhead. *J. VLSI Signal Process.* **28**(1–2), 97–113 (2001)
7. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. *Science* **4598**, 671–680 (1983)
8. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In: Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30), Research Triangle Park, USA, pp. 330–335 (1997)
9. Moreano, N., Borin, E., de Souza, C., Araujo, G.: Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **24**(7), 969–980 (2005)
10. Nollet, V., Coene, P., Verkest, D., Vernalde, S., Lauwereins, R.: Designing an operating system for a heterogeneous reconfigurable soc. In: Proceedings of the International Conference on Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France (2003)
11. Rullmann, M.: Models, design methods, and tools for improved partial dynamic reconfiguration. PhD thesis, Technische Universität Dresden (2009, to appear)
12. Rullmann, M., Merker, R.: Design and implementation of reconfigurable tasks with minimum reconfiguration overhead. In: Dynamically Reconfigurable Architectures Workshop at 19th International Conference Architecture of Computing Systems (ARCS 2006), Frankfurt/Main, Germany, pp. 132–141 (2006)
13. Rullmann, M., Merker, R.: Maximum edge matching for reconfigurable computing. In: Reconfigurable Architectures Workshop at 13th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006), Rhodes, Greece (2006)
14. Rullmann, M., Merker, R.: A reconfiguration aware circuit mapper for fpgas. In: IEEE International Parallel & Distributed Processing Symposium—IPDPS 2007, 14th Reconfigurable Architectures Workshop (2007)
15. Rullmann, M., Merker, R.: A cost model for partial dynamic reconfiguration. In: Najjar, W., Blume, H. (eds.) International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS), pp. 182–186 (2008)

16. Rullmann, M., Merker, R.: Synthesis of efficiently reconfigurable datapaths for reconfigurable computing. In: International Conference on Field-Programmable Technology 2008 (ICFPT'08) (2008)
17. Rullmann, M., Siegel, S., Merker, R.: Optimization of reconfiguration overhead by algorithmic transformations and hardware matching. In: Workshop RAW 2005 at the 19th IEEE International Parallel and Distributed Processing Symposium, pp. 151–156 (2005)
18. Rullmann, M., Merker, R., Hinkelmann, H., Zipf, P., Glesner, M.: An integrated tool flow to realize runtime-reconfigurable applications on a new class of partial multi-context fpgas. In: International Conference on Field Programmable Logic and Applications (FPL 2009, to appear), Prague, Czech Republic (2009)
19. Steiger, C., Walder, H., Platzner, M.: Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Trans. Comput.* **53**(11), 1393–1407 (2004)
20. Walder, H., Platzner, M.: Online scheduling for block-partitioned reconfigurable devices. In: Design, Automation and Test in Europe Conference and Exhibition, pp. 290–295 (2003)