Design Flow for Reconfiguration based on the Overlaying Concept

André Meisel · Alexander Dräger · Sven Schneider · Wolfram Hardt

Abstract Reconfigurable hardware combines the flexibility of software and the efficiency of hardware. Thus, Embedded Systems can benefit from reconfiguration techniques. Many special aspects of dynamic and partial reconfiguration have been already analyzed. On the one hand reconfiguration is mostly used like a hot-plug mechanism. On the other hand approaches similar to the overlaying technique, known from the Pascal runtime library, can be used. The overlaying algorithm schedules different functions to the same hardware resource during runtime.

In this paper, the overlaying concept is adapted to reconfiguration. The used reconfiguration model is presented and the costs are optimized and evaluated. The average reconfiguration time is minimized. These methods have been integrated into the design flow for reconfiguration. This approach is best suited for small FPGAs, which are crucial in embedded system design.

Keywords Configuration · FPGA · Opimaziation

André Meisel
Chemnitz University of Technology
Faculty of Computer Science
E-mail: andre.meisel@cs.tu-chemnitz.de

Alexander Dräger
Chemnitz University of Technology
Faculty of Computer Science
E-mail: andre.meisel@cs.tu-chemnitz.de

Sven Schneider
Chemnitz University of Technology
Faculty of Computer Science
E-mail: sves@cs.tu-chemnitz.de

Wolfram Hardt
Chemnitz University of Technology
Faculty of Computer Science
E-mail: hardt@cs.tu-chemnitz.de

1 Introduction

Embedded systems penetrate more and more of everyday life. The structure size of semiconductors decreases and as a result more features are implemented. Usually not all of these functions are used at the same time. Reconfiguration offers the possibility to provide functions only at the time they are actually needed.

Reconfiguration means that it is possible to change the behavior of a system after it has been implemented. In this paper the focus is on partial and run-time reconfiguration. With Xilinx Virtex II Pro FPGAs for example this kind of reconfiguration is possible.

Using reconfiguration it is possible to get an efficient chip area utilization with the same performance. This improvement is realized by using more memory, which is needed to store the non-active system behavior information. However, this memory is cheaper than logic. In addition to the chip area other resources can be optimized by reconfiguration, e.g. power consumption.

Lower design and manufacturing costs are another advantage. It is simple to add new functionality by loading a newly designed and synthesized configuration.

Besides the advantages of reconfiguration, different conditions are necessary for the implementation of hardware reconfiguration. Reconfiguration is only possible on configurable hardware like FPGAs. For our applications Xilinx FPGAs, especially Spartan 2e and Virtex II Pro, are used. A memory for inactive reconfigurable bitstreams is also necessary. The reconfiguration process also needs a control unit implemented in software or hardware.

Reconfiguration technology can be integrated in systems in mainly two different ways. The first approach is similar to the plug and play or hot-plug technology. A uniform socket must be defined for all reconfigurable modules. The bus structure in such system can vary.

Several fixed constraints for all modules are helpful to generate bitstreams with the Xilinx design flow.

The second approach is similar to the overlay technique known from the Pascal runtime library [2]. Programs written in Pascal could be partitioned into parts which are loaded only at the time they are actually needed. The overlay technique was used if there was not enough memory. If a system is too large for a given FPGA then partitioning the system and reconfiguration could solve the problem. This paper focuses on the second approach.

The overlay technique in memory management differs from a hardware overlay approach. Each program module can be loaded to almost any memory address. The only limitation is the memory size. Hardware modules can similarly be loaded into an FPGA. But there is the communication between modules which must be considered when designing a reconfigurable system. The Xilinx bus macros [8] are used to implement the inter-module communication. Bus macros are fixed connection points for modules. Overlayed modules are placed into the same partition of the FPGA. An approach to get an optimal set of partitions, which minimizes the average time for reconfiguration, is detailed in section 5. An important requirement for the optimization algorithms is the definition of different sets of modules. These sets are necessary for different configurations of the system. In section 4 the single steps to obtain these modules sets are explained. The design flow with these methods for overlaying is shown in section 3. In section 2 we describe existing methods and available tools for reconfiguration. The test results in section 6 illustrate some details of the algorithms before we conclude with section 7.

2 Existing Methods and Tools

Generally speaking, three steps are necessary to design an embedded system. Notable work has been done to define these three design steps [13, ?,?,?]. First, the system must be specified. Second, the specified system must be represented by a hardware description language. Third, this description must be transformed to bitstreams or masks. The approach presented in his paper focuses on the second step.

For partially reconfigurable systems there are approaches offering a design flow. For example, [11] begins with descriptions of the design's hardware in VHDL and a specially defined reconfiguration information file. The authors of [5] present a design flow, named Caronte FLOW. It is used for the step after the System Partitioning and Analysis Phase. We focus on the step that generates the input information for both design flows. The design flow PaDReH presented in [1] is similar to ours but does not describe an approach for module placement.

There are some methods and tools that handle the integration of IPs in an embedded system during the design flow. There are projects [7, ?] that deal with IP utilization. The IPQ Format and the IPQ Toolbox were introduced by these projects. Details about the results can be read in [14, ?,?,?]. As we want to use IPs in our reconfigurable system, our design flow uses these IP tools. The IPs will be the input for our presented design flow.

Another important aspect is the usage of graphs. In [4] graphs are used to describe scheduling and partitioning. A problem graph is defined for the design representation and a second graph, the architecture graph, describes the target architecture. Like our approach this is based on IPs. In contrast to this paper no automated generation of configurations is addressed.

Another approach is presented in [16]. The authors propose finding reconfiguration modules by defining configurations and the costs of switching between these configurations. As control data flow graphs are used as input it is very complex to design IP based systems with this work flow.

Ghiasi et al. describe in [12] an optimal algorithm for minimizing the reconfiguration time named min-RPR. The algorithm is based on the Least Imminently Used (LIU) algorithm. There are different restriction for this algorithm: the application must have a fixed schedule and there are no parallel operations allowed.

This paper focus on the automation of the generation of reconfigurable modules using some ideas of these existing works.

3 Design Flow

The design flow for reconfigurable embedded systems is divided into two main phases. The first phase takes the system specification as input and generates the constraints and the system description at register transfer level (RTL) for the synthesis. There are different approaches to this first phase as presented in section 2. Therefore, some details of the used design flow for reconfigurable embedded systems based on the overlaying concept are described in this section. The synthesis of partial bitstreams is the second phase wich is not subject of this paper. In section 2 the Xilinx design flow for partial bitstreams is mentioned.

Figure 1 depicts the design flow. The first phase of the design flow is detailed in the frame *Phase I* and the second phase is summarized in the process *Design Flow of Partial Bitstreams*.

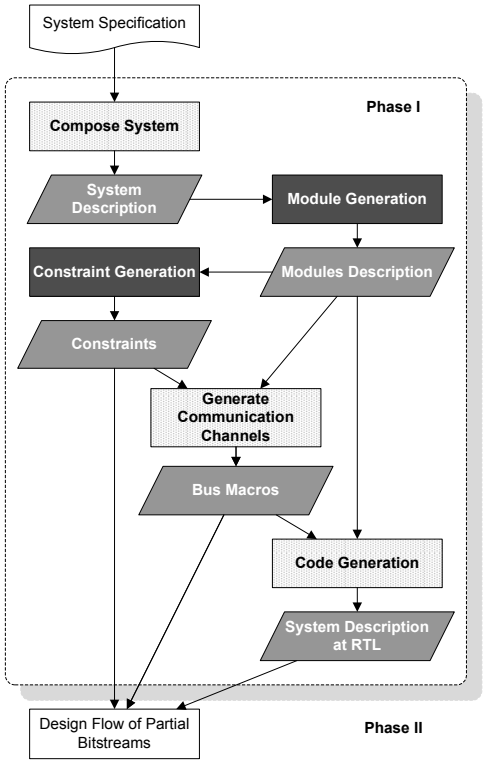

Fig. 1 Design Flow

The presented design flow in Figure 1 starts with a system specification. Functional validation of the system as e.g. in PaDReH is here not depicted. After the system is specified, the designer usually implements the functionality. The designer should use IPs (Intellectual Properties) that are available in order to reduce the design costs of a new product. A *System Description* is the result of the *Compose System* step. This is a set of chosen IPs and the system graph. The system graph is a representation of the connected IPs.

The clustering of IPs to reconfigurable modules is the task of the process *Module Generation*. Communication channels between modules are represented in a module graph. The *Module Description* is used to generate the top level RTL description in the process *Code Generation*. In in the process *Generate Communication Channels* the module graph is analyzed to automatically implement bus macros. Xilinx provides these bus macros for the placement and the exact routing of inter-module communication signals [8, **?**].

Constraint Generation needs the module description to place the module in FPGA partitions. Three properties of these partitions can be optimized. The number of partitions should be maximized so that the whole FPGA is used and number of reconfigurations is minimized. To maximize the partition number the size should be minimized. In order to minimize the bus

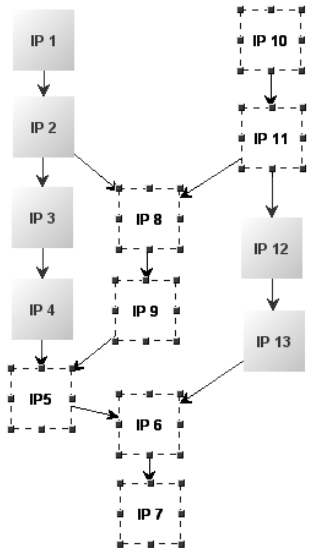

Fig. 2 System Graph

macro size the partitions placement should be analyzed. The constraints are needed to define bounding boxes in the *Design Flow of Partial Bitstreams* process. Bus macros can be placed in the FPGA after the partitions are defined. These bus macros are necessary communication channels between different modules as well as channels between modules and external interfaces. For the process *Code Generation* the identifier of generated bus macros needed.

In this paper the appropriate algorithms for *Module Generation* and *Constraint Generation* are described in the section 4 and 5.

The process *Compose System* is more detailed in [15] and the IPQ project as mentioned before. The process of reconfiguration during run-time requires a reconfiguration controller. This controller can be generated automatically from a template. These generation is not integrated in the presented design flow. More detail about reconfiguration controller are in [9, **?**].

4 Module Generation

As discussed in section 3, modules and the communication between them is essential for the design flow. In this section an approach for the algorithms of the step *Module Generation* is explained in detail.

The process *Module Generation* takes the system description as input. A system description consists of a set of IPs or VHDL components and the system graph. In Figure 2 a system graph is shown. Each vertex is associated with an IP. The edges represent the communication channels between these IPs. In the process

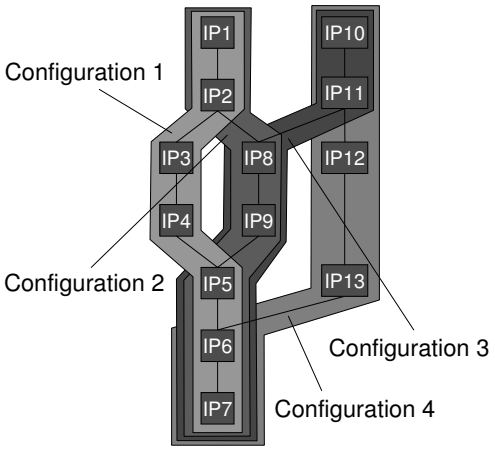

Fig. 3 System Configurations

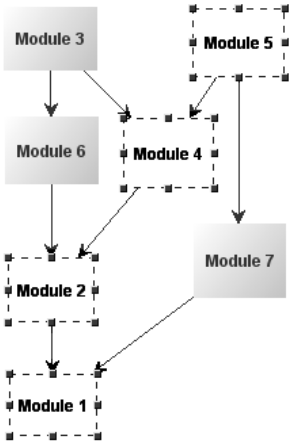

Fig. 4 Module Graph

Compose System (see Figure 1) the designer developed the system description.

The result of the *Module Generation* is a set of clustered IPs called modules. A module graph similar to the system graph will be generated.

4.1 Configurations

The first step of *Module Generation* is the definition of configurations. Configurations are needed to describe different system setups that are exchanged over time. These configurations are usually defined by the designer. The designer extracts system tasks from the specification and defines configurations for each task. Configurations are defined by grouping subsets of IPs. Each IP belongs to one or more configurations. This means that configurations can overlap. This step is similar to overlaying technique, known from the Pascal runtime library. In Pascal the programmer must partition the software.

Automated support for the designer during the matching process is possible but not detailed in this paper. Necessary conditions can be extracted from the system graph. For example, if an IP of one configuration expects data from another IP that is not in the same configuration then the designer can be warned.

It is also possible to detect these configurations automatically. This can be done by simulation and analysis of the whole system. During the simulation a monitoring of all IPs of the system is done. The simulation contains all possible tasks of the system. For each task the analysis shows which IPs are active and which are not. Monitoring can be implemented for every interface of each IP to detect the IP's state.

In Figure 2 one configuration is selected. Four configuration are defined in the example (see Figure 3).

4.2 Cluster Algorithm

Based on configurations an algorithm is proposed for automated generation of modules. Modules are defined because it is inefficient to reconfigure the whole chip. Further, it is also inefficient to reconfigure any single IP. Bus macros are necessary for communication between reconfigurable modules. If all IPs define their own modules, many bus macros are needed. The clustering of IPs reduces the number of these communication channels. Communication between IPs clustered to modules is routed automatically by standard tools. In contrast to this fact the bus macros between modules have to be implemented manually.

Another disadvantage of whole chip reconfiguration is the time needed for the reconfiguration process. Many reconfigurations accumulate to significant delays. Therefore, an optimal clustering is desirable. In this section an approach is provided to solve this problem.

Mainly the difference between two configurations should be reconfigured. This is the lower bound for module size. To optimize the reconfiguration process it is possible to combine several modules. Therefore, the communication between the modules is minimized and, in respect to time-critical processes, interruptions due to reconfiguration can be eliminated. The optimization can lead to parallely existing configurations on the chip. To change between such configurations no reconfiguration process is needed. This optimization step is detailed in section 5.

The following algorithm clusters IPs to minimal modules. Based on the possibility of overlapping configurations the algorithm determined all IPs which belong to the same configurations. The input for the algorithm is a set of configurations C and a set of IPs I. M is the set of modules.

for all $i \in I$ **do**
  $IP\_i\_is\_added \leftarrow$ false
  for all $m \in M$ **do**
    $j \leftarrow$ IP, where IP $\in m$
    $i\_in\_m \leftarrow$ true
    for all $c \in C$ **do**
      if $i \in c$ xor $j \in c$ **then**
        $i\_in\_m \leftarrow$ false
      end if
    end for
    if $i\_in\_m =$ true **then**
      $m \leftarrow m \cup \{\text{IP } i\}$
      $IP\_i\_is\_added \leftarrow$ true
    end if
  end for
  if $IP\_i\_is\_added =$ false **then**
    $M \leftarrow M \cup \{\text{new module } m\}$
    $m \leftarrow m \cup \{\text{IP } i\}$
  end if
end for
Output: M

For the example system in Figure 2 the algorithm defines seven modules. After defining the modules the connection between the modules will be analyzed and the module graph generated. In Figure 4 the module graph for the example system is shown. This optimal result of the algorithm leads to a minimal FPGA utilization and to reconfigurations that only change the differing modules. The configuration 3 is highlighted in both Figures 2 and 4. The module 5 for example consists of the IPs 10 and 11. Both IPs are in configuration 3 and 4 but not in configuration 1 or 2.

After the system is partitioned into modules, the mapping of modules to the hardware follows. This problem is detailed below.

5 Constraint Generation

The main goal of all considerations is a good partitioning of the hardware, here an FPGA. An optimal partitioning cannot be determined in an acceptable time, because this problem is NP-equivalent. A partition is called optimal, when the system with all configurations can be placed with minimal costs of reconfiguration time.

Searching for a good solution becomes easier by concentrating the algorithms on the most-used modules and most-occurring reconfigurations. The Markov chain [10] is just the right concept for this problem.

The input for optimization is a full system description (modules, configurations, reconfigurations), the target FPGA, a set of constraints and a set of estimation functions for reconfiguration time, for the CLB (configurable logic block) requirement of communication, and for buffer size.

5.1 Model

The configurations and reconfigurations are described as a Markov chain. A reconfiguration is a change from one configuration to another with specific probability. A reconfiguration is a tuple of three elements:

k_s source configuration
k_t target configuration
p probability

5.2 Optimize Function

The object of optimization is to minimize the average reconfiguration time while keeping all constraints. Both facts can comprised to the function (1). t_r is the function for reconfiguration time.

$$f(C) \cdot \sum_{r=(k_s,k_t,p)\in R} t_r(r) \cdot P(k_s) \cdot p \tag{1}$$

The sum is the average reconfiguration time. $f(C)$ is function, which describes, how many constraints are valid. For example: $\frac{n+1}{k+1}$ is a useful definition for $f(C)$, where n is the number of constraints, and k is the number of valid constraints.

5.3 Phases of Optimization

Figure 5 shows the three phases of the optimization. In the first phases, the start phase, the probabilities for all configurations and modules are computed and a minimal partitioning (a system with a minimal number of slots with minimal slot sizes) is implemented.

The actual optimization consists of different algorithms. The first step is a heuristic, which sorts all modules along size and probability. In this sequence the heuristic tries to assign a module in every configuration to one slot. The second step is a randomized optimization procedure called treshold accepting (TA)[3, 6]. TA varies the assignment from configurations to the slots. If TA does not find a better solution, the next phase is InsertSlot(IS). IS analyses the free space and inserts a new slot. If IS cannot inserts a new slot, ExpandSlot(ES) is trying to expand one of the slots. If ES cannot expand any slot, then all slots become the minimum size (ReduceSlots) and IS is trying.

The end phase determines the slot widths, the slot order, and both implicates the slot offsets. The output is a full described system (2), consisting of a set

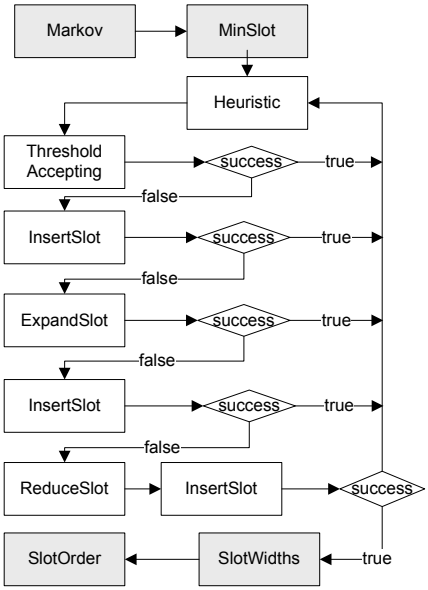

Fig. 5 The optimization phases

of modules, a set of configurations, of reconfigurations and of slots, and the FPGA. The configurations and the reconfigurations have now a function $\delta : M \longrightarrow S$.

$$S = (M, C, R, S, FPGA) \tag{2}$$

5.3.1 MinSlot Algorithm

The procedure that generates a minimal slot set with minimal sizes is an optimal algorithm.

Input: C, configuration set
$S \leftarrow \emptyset$
for all $c = (M, E) \in C$ **do**
  sort(M, size, descending)
  delete all marks $\forall s \in S$
  for all $m \in M$ **do**
    $b, l \leftarrow$ nil
    $b \leftarrow$ the smallest slot with b.size $\geq$ m.size and b is unmarked
    $l \leftarrow$ the greatest slot with l.size $\leq$ m.size and l is unmarked
    if b=nil and l=nil **then**
      $S \leftarrow S \cup \{newSlot(m.size)\}$
      mark the new slot
    else if b=nil **then**
      l.size $\leftarrow$ m.size
      mark l
    else
      mark b
    end if
  end for
end for

Output: S

The MinSlot algorithm leads to the minimal number of slots. The algorithm inserts only a new slot when all slots are marked. The configuration with the most modules determines the number of slots. Deleting a slot and this configuration doesn't fit in the slot set.

The algorithm expands one slot, when all slots, which greater or equal than this module, are marked. The modules are sorted from the smallest to the largest module, so that no slot is expand unnecessary. The size of one slot is $\max\{m_1.\text{size}, \ldots, m_k.\text{size}\}$. Decrementing the slot size by 1, the module with the biggest size doesn't fit into this slot and at least one configuration doesn't fit in the slot set.

5.4 Heuristic

The central heuristic of the optimization procedure is that the average reconfiguration time develops to a lower value if every module, especially the largest and the most frequented modules, is as often as possible assigned to the same slot.

This heuristic is the base for optimization phase *Heuristic* and for *Threshold Accepting*. The phase *Heuristic* sorts all modules according to the product of size and probability and optimizes every module to the slot with the best quality for fit in this module. In *Threshold Accepting* a non-static module and a non-static slot are determined randomized and the heuristic tries as often as possible to assign this module to this slot.

6 Example

In this section the optimization results of the example system given in section 4 are shown. The process *Module Generation* determined seven modules (see Figure 4). This is the optimal result for a minimal FPGA utilization which leads to 4 slots and maximal 490 CLBs (IP size 70 CLBs) on an FPGA. If there is unused FPGA area, an optimization is possible. The target FPGA has 32 rows and 20 columns of CLBs ($\sum$ 640 CLBs). All possible reconfigurations have the same probability. The required CLBs for connections between modules are not considered in this example. Table 1 shows the optimization results. If the IP size is 70 CLBs then the average number of reconfigured CLBs is 134.17 (20.96% of 640 CLBs) and the FPGA is partitioned into 5 slots. If the size of IPs decreases, the number of modules that can be placed simultaneously increases and the average number of reconfigured CLBs decreases. This can be seen in the last two columns. An additional effect

IP size in CLBs	70	60	55
System size in CLBs	910	780	715
change rate in CLBs	134.17	115.00	68.75
change rate in percent	20.96%	17.96%	10.74%
number of slots	5	5	6

Table 1 Test results for the example

of the optimization is that some modules are placed as static ones in the FPGA.

7 Conclusion

In this paper, a design flow for reconfigurable systems based on the overlaying concept, in contrast to the hotplug technology, is presented. The overlaying concept in the Pascal runtime library was used to handle large programs if there was not enough memory. Similar to this the reconfiguration is used in the context of embedded systems. The main problems in reconfigurable systems are the communication between reconfigurable modules and the reconfiguration overhead. The presented proposal facilitates moving from a definition of configurations to the automated creation of reconfigurable modules. After the designer has defined all reconfiguration parameters, our approach offers the automated generation of optimized reconfiguration partitions. The presented algorithms generate modules in a way that minimizes the number of internal interfaces (bus macros) and minimize the average time for reconfiguration.

The example illustrates the applicability of the proposed methods and highlights major advantages in design automation. A change in the system specification necessitates changes to the system implementation. Because of the encapsulation to configuration, the proposed design steps can run automatically without adaption. The process of generating an updated system implementation becomes simpler and more straightforward. This approach to the design process, then, facilitates the efficient and effective prototyping of IP based dynamic reconfigurable systems.

References

1. E. Carvalho, M. Calazans, E. Brio, and F. Moraes. PaDReH – A Framework for the Design and Implementation of Dynamically and Partially Reconfigurable Systems. In *Proc. of the 17th SBCCI*, Sept. 2004.
2. L. Dorfman and M. J. Neubauer. *Turbo Pascal Memory Management Techniques*. Windcrest Har Dsk edition, February 1993.
3. G. Dueck and T. Scheuer. Treshold accepting: A general purpose optimization appearing superior to simulated annealing. *Journal of Computational Physics*, 1990.
4. M. Eisenring and M. Platzner. An implementation framework for run-time reconfigurable systems, 2000.
5. F. Ferrandi, M. D. Santambrogio, and D. Sciuto. A Design Methodology for Dynamic Reconfiguration: The Caronte Architecture. In *Proc. of the 19th IPDPS*. IEEE Computer Society, Apr. 2005.
6. I. Gerdes, F. Klawonn, and R. Kruse. *Evolutionre Algorithmen*. Friedr. Vieweg & Sohn Verlag, 1 edition, 2004.
7. IPQ project partners. Projekt IPQ. https://www.ip-qualifikation.de, Jan. 2005.
8. D. Lim and M. Peattie. *Xilinx Application Note XAPP290: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*. Xilinx, Inc., May 2002.
9. A. Meisel, M. Visarius, W. Hardt, and S. Ihmor. Self-Reconfiguration of Communication Interfaces. In *Proc. of the 15th RSP*, Geneva, Switzerland, June 2004. IEEE Computer Society.
10. G. C. Pflug. *Stochastische Modelle in der Informatik*. Teubner, 1986.
11. I. Robertson and J. Irvine. A design flow for partially reconfigurable hardware. *Trans. on Embedded Computing Sys.*, 3(2):257–283, 2004.
12. M. S. Soheil Ghiasi, Ani Nahapetian. An optimal algorithm for minimizing runtime reconfiguration delay, 2003.
13. K. Ten Hagen. *Abstrakte Modellierung digitaler Schaltungen*. Springer, 1995.
14. M. Visarius, J. Lessmann, F. Kelso, and W. Hardt. Generic Integration Infrastructure for IP based Design Processes and Tools with a Unified XML Format. *Integration, the VLSI journal*, 37(4):289 – 321, Sept. 2004.
15. M. Visarius, A. Meisel, M. Scheithauer, and W. Hardt. Dynamic Reconfiguration of IP based Systems. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, pages 70 – 76, Montreal, Canada, June 2005. IEEE Computer Society.
16. X.-J. Zhang and K.-W. Ng. An effective high-level synthesis approach for dynamically reconfigurable systems. volume 01, page 343, Los Alamitos, CA, USA, 2000. IEEE Computer Society.