TECHNISCHE UNIVERSITÄT DRESDEN

Dissertation

Models, Design Methods and Tools for Improved Partial Dynamic Reconfiguration

Markus Rullmann

Supervisor
Prof. Dr.-Ing. habil. Renate Merker

FAKULTÄT **ELEKTROTECHNIK** UND **INFORMATIONSTECHNIK**

Technische Universität Dresden

Models, Design Methods and Tools for Improved Partial Dynamic Reconfiguration

Markus Rullmann

von der Fakultät Elektrotechnik und Informationstechnik
der Technischen Universität Dresden

zur Erlangung des akademischen Grades eines

Doktoringenieurs

(Dr.-Ing.)

genehmigte Dissertation

Vorsitzender:	Prof. Dr.-Ing. habil. H. Schreiber
Gutachter:	Prof. Dr.-Ing. habil. R. Merker
	Prof. Dr. rer. nat. F. J. Rammig
Tag der Einreichung:	07.09.2009
Tag der Verteidigung:	26.02.2010

Contents

List of Symbols

Notation

s	scalar value
$\mathbf{v}$	vector
$\mathcal{S}$	set
$\mathsf{f}(), \mathsf{f} : \mathcal{S} \mapsto \mathcal{S}$	function
$\mathbf{v}()$	vector valued function
$k = 1, \ldots, K$	range from 1 to K
$\lvert\mathcal{S}\rvert$	cardinality of the set $\mathcal{S}$

Chapter 3

$\mathbf{a}$	node allocation, p. 77
s	total configuration size, p. 53
$\overline{s}$	average configuration size, p. 53
t	total reconfiguration time, p. 52
$\overline{t}$	average reconfiguration time, p. 52
$t_{\mathcal{E}}$	total reconfiguration time for interconnect, p. 66
$\mathsf{a}()$	allocation, p. 59
$\mathsf{a}_\mathsf{s}()$	re-labelling of the source label, p. 59
$\mathsf{a}_\mathsf{d}()$	re-labelling of the drain label, p. 59
$\mathsf{a}_\mathsf{e}()$	edge allocation, p. 60
$\mathbf{d}()$	device configuration, p. 49
$\mathsf{l}_\mathsf{n}()$	node configuration, p. 57
$\mathsf{l}_\mathsf{s}()$	source label, p. 57
$\mathsf{l}_\mathsf{d}()$	drain label, p. 57
$\mathbf{r}()$	reconfiguration bitmap, p. 51
$\mathsf{u}()$	reuse function, p. 65
$\mathcal{A}$	stack of node allocations, p. 77
$\mathcal{E}_\mathsf{T}$	set of transitions between tasks, p. 50
$\mathcal{N}_\mathsf{T}$	set of tasks, p. 50
$G_i(\ldots)$	input graph of task i, p. 59

$G'_i(\ldots)$	image graph of task i, p. 59
$G_{\mathsf{A}}(\mathcal{N}_{\mathsf{A}}, \mathcal{E}_{\mathsf{A}}, \ldots)$	virtual architecture graph, p. 63
$G(\mathcal{N}_{\mathsf{T}}, \mathcal{E}_{\mathsf{T}})$	reconfiguration state graph (RSG), p. 50

Chapter 4

$\mathbf{a}$	node allocation, p. 107
$\mathbf{a}'$	possible LE allocation, p. 118
$A_{a'}$	binary variable to select an LE allocation, p. 119
B_{a_1,a_2}	binary variable to select a pair of LE allocations $\mathbf{a}'_1, \mathbf{a}'_2$, p. 120
$E_{\mathsf{L},i}$	number of local connections in task i, p. 115
$E_{\mathsf{M},i}$	number of merged connections in task i, p. 116
E_{U}	number of matching edges between all tasks, p. 120
S_v	binary variable to select an LE v, p. 114
$\mathsf{w}_{\mathsf{L}}()$	number of local connections in an LE, p. 117
$\mathsf{w}_{\mathsf{M}}()$	number of merged connections in an LE, p. 117
$\mathcal{A}$	set of node allocations, p. 107
$\mathcal{A}'$	set of possible LE allocations, p. 118
$\mathcal{A}''$	set of feasible LE allocations, p. 118
$\mathcal{L}_i$	relation between nodes and LEs for task i, p. 113
$\mathcal{R}_n$	set of LE resources for node n, p. 112
$\mathcal{V}_i$	set of LEs for task i, p. 112

Chapter 5

$C_{\mathsf{dp},m}$	implementation costs of a datapath of reconfigurable module m, p. 147
$C_{\mathsf{res},m}$	resource costs of reconfigurable module m, p. 147
$C_{\mathsf{mux},m}$	dataflow multiplexer costs of reconfigurable module m, p. 147
$C_{\mathsf{wire},m}$	interconnect costs of reconfigurable module m, p. 147
$\mathsf{a}()$	allocation, p. 59
$\mathsf{a}_{\mathsf{T}}()$	resource type allocation, p. 143
$\mathsf{b}()$	number of resource instances, p. 144
$\mathsf{b}_i()$	number of resource instances in task i, p. 157
$\mathsf{b}_m()$	number of resource instances in reconfigurable module m, p. 158
$\mathsf{b}_{\mathsf{VA}}()$	number of resource instances in the VA, p. 158
$\mathsf{c}()$	schedule, p. 150
$\mathsf{l}()$	latency, p. 142
$\mathsf{o}()$	offset, p. 142

$\mathbf{r}()$	reconfiguration bitmap for modules, p. 148
$\mathbf{w}_{\mathsf{LE}}()$	resource cost, p. 142
$\mathbf{w}_{\mathsf{S}}()$	control signal cost, p. 142
$\mathbf{w}_{\mathsf{W}}()$	wire cost, p. 148
$\mathbf{w}_{\mathsf{t}}()$	reconfiguration costs of a reconfigurable element, p. 149
$\mathsf{x}()$	number of inputs of a dataflow multiplexer, p. 146
$\mathcal{E}_{\mathsf{A},m}$	set of edges in the datapath of reconfigurable module m, p. 146
$\mathcal{E}_{\mathsf{C}}$	set of precedence constraints, p. 140
$\mathcal{E}_{\mathsf{D}}$	set of data dependencies, p. 140
$\mathcal{N}_{\mathsf{A},m}$	set of nodes in the datapath of reconfigurable module m, p. 146
$\mathcal{N}_{\mathsf{O}}$	set of operations, p. 140
$\mathcal{N}_{\mathsf{T},m}$	set of tasks realized in reconfigurable module m, p. 145
$\mathcal{N}_{\mathsf{V}}$	set of variables, p. 140
$\mathcal{M}$	set of reconfigurable modules, p. 148
$\mathcal{R}$	set of resource instances, p. 142
$\mathcal{R}_{\mathsf{T}}$	set of resource types, p. 142
$G_{\mathsf{C}}(\mathcal{N}, \mathcal{E}_{\mathsf{C}})$	conflict graph, p. 159
$G_{\mathsf{R}}(\mathcal{N}, \mathcal{R}_{\mathsf{T}}, \mathcal{E}_{\mathsf{R}})$	resource graph, p. 143

List of Abbreviations

ca.	approximately (*Latin:* circa)
cf.	compare (*Latin:* confer)
e.g.	for the sake of example (*Latin:* exempli gratia)
et al.	and others (*Latin:* et alii)
et seq.	and the following (*Latin:* et sequens)
etc.	and so on (*Latin:* etcetera)
i.e.	that is (*Latin:* id est)
p.	page
ACO	Ant Colony Optimization
ALU	Arithmetic Logical Unit
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Integrated Processor
CDFG	Control Dataflow Graph
CFG	Configuration
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DDR	Double Data Rate
DFG	Dataflow Graph
DSP	Digital Signal Processor
EAPR	Early Access Partial Reconfiguration
EDIF	Electronic Design Interchange Format
ESM	Erlangen Slot Machine
FPL	Field Programmable Logic
FPGA	Field Programmable Gate Array
FU	Functional Unit
GPIO	General Purpose Input Output
HDL	Hardware Description Language
HLS	High-Level Synthesis
ICAP	Internal Configuration Access Port
ILP	Integer Linear Program
LB	Logic Block

LCSG	Largest Common Subgraph
LE	Logic Element
LMG	Labelled Multidigraph
LUT	Look-up Table
MWCP	Maximum Weighted Clique Problem
OPB	On-chip Peripheral Bus
OS	Operating System
PLB	Processor Local Bus
PPC	PowerPC
RAM	Random Access Memory
RSG	Reconfiguration State Graph
RSOC	Reconfigurable System on a Chip
RTL	Register Transfer Level
RTR	Runtime Reconfiguration
SA	Simulated Annealing
SOC	System on a Chip
SRL	Shift Register Logic
UART	Universal Asynchronous Receiver Transmitter
VA	Virtual Architecture
XDL	Xilinx Description Language

Chapter 1

Introduction

Electronic devices today contain, apart from analog interfaces to the physical world, a wide range of digital circuitry for data processing and operation control. Digital circuits are part of automotive systems, mobile devices, home entertainment and multimedia systems, telecommunication networks, computers, and many more electronic devices in everyday live. As these devices become even more advanced, the requirements in terms of computing performance, energy consumption, reliability, and manufacturing cost increase, too.

Digital circuits are the common basis for a range of different digital computing architectures. All those architectures are dedicated to implement the data processing and control functionality in the most efficient way. The most common architecture is the instruction-set based processor that implements the Von-Neumann execution model. In Von-Neumann computing, the behaviour of the digital circuit is controlled by a sequential stream of instructions that defines the data processing and control operations. Von-Neumann computing is used in different computing architectures: general purpose processors (CPU), digital signal processors (DSP), and application-specific instruction-set processors (ASIP). The internal processing of the instruction stream is not necessarily sequential. However, the execution model assumes that the processing is controlled by these instructions such that the data is processed in the specified order. Von-Neumann computing allows us to build very flexible processors that compute any type of algorithm. The use of such a processor is not efficient for every application. A processor may contain functions that is not used in every application. Also, the processing performance may not be sufficient for any application or algorithm. In addition, the storage, loading, and decoding of the instruction stream requires a significant amount of circuitry and energy. If the computation and control required by an application is of a more static kind of nature, it may be much more efficient to design customized digital hardware that realizes the computations. Instead of a sequential instruction stream that controls the operation of a generic machine, the operations are realized as a fixed digital circuit. Now, only the application data is supplied to the circuitry at runtime. Many electronic

devices employ this kind of computing paradigm in application-specific integrated circuits (ASIC) that implement selected parts of an application in a very efficient way. A major drawback of this technology is its static nature: after manufacturing, the functionality of an ASIC can not be changed. If the application changes, a new ASIC is to be designed and integrated into the computing system. A solution to this problem is provided by programmable logic. Programmable logic consists of a digital circuit that can be configured to implement a wide range of functions. Therefore, the interconnect and the function of the digital logic is programmable. The configuration describes not a sequential instruction stream as in Von-Neumann computing, but the logic function of the data processing hardware itself. It means programmable logic can be customized after the devices are being manufactured. Programmable logic is almost as flexible as an instruction-set processors and the configuration realizes a digital circuit that behaves similar to an ASIC. Hence programmable logic provides a very flexible hardware platform to implement different applications very efficiently.

Originally, programmable logic has been used to replace discrete logic and ASICs in low volume products. Later on, the use of programmable logic to build computing systems became apparent. The original concept of reconfigurable computing is attributed to Gerald Estrin [29] who proposed a computer consisting of a processor and additional reconfigurable hardware. In the 1960s, programmable logic was not available and therefore Estrin's idea did not have notable impact on computing architectures. It was not until the 1980s that powerful programmable logic appeared, which allowed to build complex computing systems. The new programmable logic devices were called field-programmable gate arrays (FPGAs). Driven by the idea of a regular array of programmable computing elements and interconnect, the research into reconfigurable computing broadened in the early 1990s. It has been in the focus of computer architecture research until today.

The advantages of reconfigurable computing may be summarized in short: programmability allows the hardware to be customized after fabrication. The inherent parallelism of the programmable hardware allows a significant speedup in algorithm execution over conventional processor-based computing, even at lower clock frequencies. The parallel computation at lower clock frequencies also increases the energy efficiency of the implementation. The mass fabrication of programmable logic gives a significant cost advantage over ASICs for low volume products.

The emergence of programmable logic that allows a runtime adaptation of the configuration and hence a change of the operational digital circuit, led to a whole set of new concepts for reconfigurable computing. The programming data for the programmable logic device was perceived software-like, called *configware*. Still, configware does not denote a sequential instruction stream as for processors but it denotes the configuration of an array of computing elements and interconnect. Execution models now include the frequent reconfiguration of the programmable logic: at runtime, the programmable logic does not implement fixed functions, but

the functionality is adapted as required by the application. Thus the functional diversity of the programmable logic appears much more versatile to the application compared to the functionality offered by a single configuration. The application perceives reconfigurable functions as virtual hardware, a concept similar to virtual memory in general purpose computing.

The reconfiguration of programmable hardware can significantly increase the efficiency of reconfigurable computing systems, but it also has drawbacks to be considered. Usually, the amount of programming data for the hardware is large, because the data configures all parts of the reconfigurable hardware directly, and the data are not as densely coded as software instructions. Hence if multiple configurations are used, a large amount of memory must be provided to store these data. Because the programming data can be transferred into the programmable logic device only at a limited bandwidth, the reconfiguration also introduces a time overhead. During the reconfiguration, parts of the programmable logic are not in a known state and can not perform meaningful computations. A reconfigurable computing system needs to handle these restrictions at runtime.

In the thesis at hand we investigate where the large amount of programming data originates. We develop a theoretical model that describes the relationship between the functionality of the hardware and the necessary configuration data. Based on this model we propose several techniques to reduce the amount of data needed to change the device's configuration in order to implement new functionality. The effect on reconfigurable computing systems is twofold: the amount of memory required to store programming data is reduced as well as the reconfiguration runtime overhead. Both leads to more efficient realizations of reconfigurable computing applications.

At the low level, we target the reduction of configuration data directly and take the description of the hardware functionality as an input. We demonstrate how similar hardware structures can be identified in the input. We describe how the similarity can be exploited to reduce configuration data while the description is mapped to the reconfigurable device.

We apply the reconfiguration model to a high-level specification of hardware functionality as well. It is assumed that the hardware functions are reconfigured at runtime, i.e. one function is executed after another. At the high-level, the specification is compiled into a detailed hardware description that contains hardware structures being optimized for similarity. Hence, we achieve a higher similarity of the hardware structures and thus less programming data overhead compared to a non-optimized hardware description. The high-level analysis further provides insight into the necessity of runtime reconfiguration. We provide exact information on how many resources are required with and without runtime reconfiguration for the same high-level specification. With our approach, designers can trade-off resource requirements against reconfiguration cost.

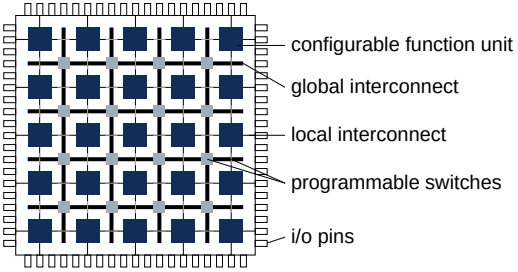


Figure 1.1: Components of a generic FPGA architecture.

1.1 Reconfigurable Computing

Many of today's applications have very demanding requirements on computation performance, flexibility, and power consumption. E.g. in the domain of high performance computing the highest possible performance is required for a wide range of commercial and scientific applications. Several vendors offer hybrid FPGA/CPU computers [62][22][83]. Other reconfigurable computing architectures target the multimedia application domain, cf. [58][66][93][110][3][85].

Reconfigurable computing combines the flexibility of a software processor with the performance of a dedicated hardware implementation. A reconfigurable computing fabric consists of a flexible array of configurable functional units and a configurable interconnect, cf. Figure 1.1. The fabric can be configured to function like specialized datapath similar to an ASIC or ASIP implementation. The fabric contains either dedicated control units or the datapath control must be realized by using generic logic elements. The performance and power efficiency of reconfigurable computing arises from these custom datapath configurations: The datapath is often built such that there is no sequential control flow as in Von-Neumann computing. The datapath can be deeply pipelined and a high degree of instruction level parallelism can be realized. All that can be tailored to the application at hand.

1.1.1 Reconfigurable System on a Chip (RSOC)

Most reconfigurable computing architectures can not only implement algorithm processing, but also a significant part of the system architecture: Memory interfaces, local on-chip caches, system buses, peripheral interfaces, and control processors – everything that is needed to build complex, reconfigurable systems on a chip (RSOC).

In 2005 we published an example of an RSOC [97]. At that time it was the first implementation of an RSOC that used the embedded PowerPC CPU in a VirtexII-Pro

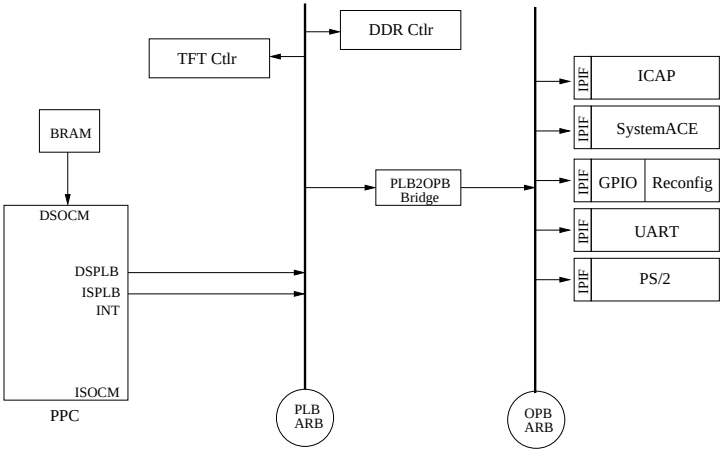


Figure 1.2: RSOC bus architecture and peripherals.

as the system's main processor. The complete system architecture is shown in Figure 1.2. The system contains one main processor (PPC), two system buses (PLB, OPB), and a number of peripherals. The system is based on an embedded Linux OS (Montavista Linux) that manages the software application and contains a driver that gives access to the device's configuration control. The high-bandwidth peripherals are connected to the CPU via the PLB: the main memory (DDR) controller, the TFT controller, and the PLB-to-OPB bridge. The low-bandwidth peripherals are connected to the CPU via the OPB and the PLB-to-OPB bridge: the serial port (UART), the mouse/keyboard (PS/2), the harddisk controller (SystemACE), the configuration controller (ICAP), and finally the generic interface to the reconfigurable area (GPIO/Reconfig).

While the overall system implements a general system on a chip, two peripherals are unique to RSOCs: the ICAP controller and the GPIO/Reconfig peripheral. The ICAP controller allows the system to access the internal configuration logic of the device, i.e. the software running on the CPU can change the configuration of the device at runtime. The RSOC contains resources that are dedicated to be reconfigured at runtime. In order to provide a generic interface to the logic that is configured onto these resources, we inserted a General Purpose IO (GPIO) interface to these resources. Special busmacros provide a static hookup to realize a physical connection to the reconfigurable circuits.

The limitations of the configuration architecture in the VirtexII-Pro devices and the I/O-Pin connections to external components led to the floorplan as shown in Figure 1.3.

So far we have identified the following key features that an architecture must

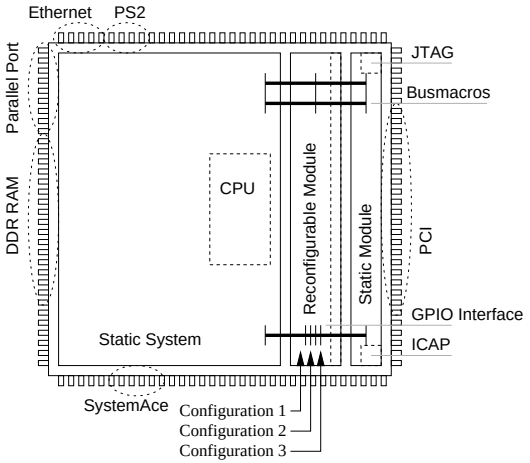


Figure 1.3: Floorplan of the RSOC depicted in Figure 1.2.

posses to be able to implement RSOCs: The architecture must provide means to realize a runtime control of reconfiguration. It must also provide access to the configuration interface of the architecture in order to customize the device at runtime. The device customization at runtime must be performed such that non-interruptable parts of the device stay operational while re-programming the dynamic part of the device. This feature is frequently called *partial runtime reconfiguration*.

1.1.2 Anatomy of an Application

Consider an application that is executed on an RSOC as depicted in Figure 1.4. The application consists of different tasks 1 to N that are implemented as hardware tasks to be executed on the reconfigurable fabric. The task execution control and the system management is usually performed by an operating system (OS) running on the CPU. Here we focus on the execution of the hardware tasks on the reconfigurable fabric. A hardware task is first loaded onto the reconfigurable fabric in step (c) and then the task is executed (e). The steps (c) and (e) are repeated for each task. The OS manages the resource allocation in the reconfigurable fabric, the task execution, and the communication between tasks.

On the example above several challenges in RSOC design can be identified:

Hardware/Software Partitioning The original application must be partitioned into hardware and software tasks. The decision is made on the basis of performance requirements and the available resources for the hardware tasks.

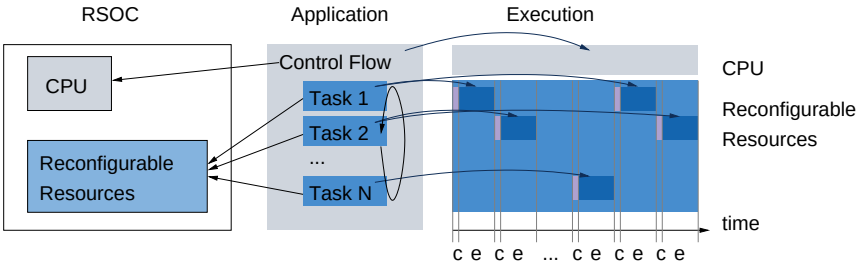


Figure 1.4: A reconfigurable application running on an RSOC. The reconfigurable resources are configured (c) to implement a task before the task can be executed (e).

Hardware Task Implementation The original application is usually specified in C or C++. However, the hardware tasks must be re-implemented in a hardware specific design language like Verilog/VHDL or re-implemented for a C-based synthesis tool. Hardware tasks can also be implemented with different resource requirements and performance trade-offs.

Hardware Reconfiguration and Hardware Task Execution Management At runtime, the reconfigurable fabric must be managed. The OS schedules the loading of tasks into the reconfigurable fabric, it manages the placement of tasks in the fabric and it controls the task execution. The loading of hardware tasks often requires considerable effort and requires efficient scheduling and placement strategies.

1.1.3 RSOC Design Characteristics and Trade-offs

Here we introduce several measures that characterize an RSOC implementation of an application. The measures are either a requirement or a result of the system design and implementation. At first, the characteristics are described in detail. After that the trade-offs between different characteristics are illustrated.

The system parameters include:

- The *code size* of the application determines the amount of data required to store the binary code of the application. This includes the processor's software libraries and configuration data of the configurable hardware.
- The *resource use* measures the amount of device resources that are required to implement the RSOC. These include general logic of the reconfigurable fabric and the use of macro blocks like on-chip RAM, multipliers/DSP blocks, clock resources, processor cores, and i/o resources etc.
- A RSOC may employ a number of different clock signals. The *clock period* of certain parts in the RSOC relates to the processing speed achieved within that part, even though other implementation parameters are relevant as well. The

CPU clock determines the processing speed of the software part of the application, the system bus clock determines largely the available communication bandwidth, and the clock period of the hardware task implementation defines the task's performance.

- The *execution time* of the application covers the execution time of the individual tasks as well as the operating system overhead and time overhead inferred by dynamic reconfiguration.
- The *reconfiguration time* measures the time required to reconfigure the device before the new task can be executed.

Apart from general system design issues, there are trade-offs specifically related to runtime reconfiguration. The RSOC designer needs to consider the following if he wants to implement the reconfigurable tasks: resource utilization of the task, the task execution time, the number of reconfigurable tasks (task granularity), and frequency of reconfiguration between tasks. The parameters may interact as follows:

- If a task is implemented such that more operations are executed in parallel then the resource utilization increases, but the task execution time decreases.
- A reconfigurable task that uses more resources also requires a longer time to be configured on the device, which partially outweights the gain in execution time.[1]
- A fine granularity of tasks may increase the efficiency of the data processing, but a fine granularity is likely to increase the amount of configuration data required as well as the number of reconfigurations at runtime.
- Usually the reconfigurable tasks exchange application data between each other. This data must be buffered at runtime in order to be transferred from the task, which existed before reconfiguration to the task after reconfiguration. The frequency of task reconfiguration also determines how much data must be buffered. If the same task runs for a longer period of time, more data to be buffered is produced. Thus frequent reconfiguration reduces the intermediate buffering required but also increases the time budget needed to perform partial reconfiguration.

The considerations discussed above are now illustrated on Example 1.1.

Example 1.1 *A synthetic example consists of a reconfigurable application that is implemented in three different versions A–C. On this example we demonstrate the relationship between reconfiguration time, throughput, latency and memory requirements. Version A is fully static with no reconfiguration overhead o (i.e.* $o_A = 0$*). However the throughput t is limited to* $t_A = 1\frac{data}{s}$ *because the implementation could not be optimized for any task in particular. The hardware contains a generic circuit that supports*

[1] Later on we will see that reconfiguration time is not a function of resource utilization but of configuration differences.

all parts of the application. Version B implements a very optimized circuit for each task of the application and reaches – without reconfiguration overhead – a throughput of $t_B = 5\frac{data}{s}$. *The reconfiguration overhead is assumed to be* $o_B = 10\frac{s}{rec}$ *(read: seconds per reconfiguration). We further assume a Variant C that is implemented such that the reconfiguration overhead is reduced to* $o_C = 2\frac{s}{rec}$ *at the expense of little reduced throughput of* $t_C = 4.5\frac{data}{s}$. *Let the reconfiguration rate* r *define how many data is processed before the tasks are reconfigured. Thus a low rate means more reconfigurations per time scale.*

The overall throughput $\overline{t}$ *of the application that includes reconfiguration overhead depends on the reconfiguration rate* r *which defines how many data is processed before the tasks are reconfigured:*

$$\frac{\overline{t}}{\left\lceil \frac{data}{s} \right\rceil} = \frac{\frac{r}{\left\lceil \frac{data}{rec} \right\rceil}}{\frac{o}{\left\lceil \frac{s}{rec} \right\rceil} + \frac{\frac{r}{\left\lceil \frac{data}{rec} \right\rceil}}{\frac{t}{\left\lceil \frac{data}{s} \right\rceil}}}. \tag{1.1}$$

The output data of a task must be buffered before it can be processed by the next task, hence the required memory is proportional to r. *The latency* l *of the application is given by the time required for reconfiguration* o *and the delay caused by the processing of the data – which depends on the reconfiguration rate* r*:*

$$\frac{l}{\left\lceil \frac{s}{rec} \right\rceil} = \frac{o}{\left\lceil \frac{s}{rec} \right\rceil} + \frac{\frac{r}{\left\lceil \frac{data}{rec} \right\rceil}}{\frac{t}{\left\lceil \frac{data}{s} \right\rceil}}. \tag{1.2}$$

In Figure 1.5 we plot the overall throughput of the implementation if the reconfiguration is performed at different intervals, i.e. after different amounts of data r *were processed. The static version A has a fixed throughput of* $\overline{t}_A = 1\frac{data}{s}$. *Although version A has a much lower throughput than version B and C, it's overall performance is better than version B and C if the reconfiguration is performed often. However, if the reconfiguration rate increases then the reconfiguration overhead becomes less compared to the processing time. In this case, the versions B and C converge to the throughput without reconfiguration overhead* t_B *and* t_C. *At the same time, the execution latency and memory requirements increase. Thus an efficient reconfigurable implementation must realize a high throughput at the lowest possible reconfiguration rate. In our example we demonstrate that this can be achieved by reduced reconfiguration overhead* o*, even if this sacrifices the maximum throughput. The plots in Figure 1.5 show that version C with lower reconfiguration overhead is superior to version B in the desired range of low latency and memory consumption.*

After a general introduction into reconfigurable computing we present a summary common reconfigurable architecture features.

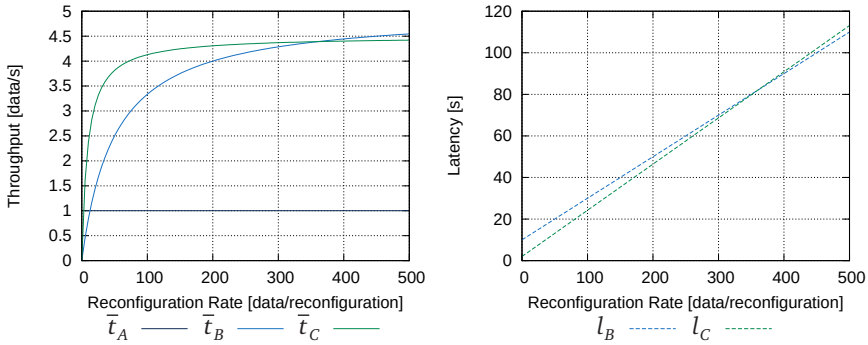


Figure 1.5: Plots for average throughput $\overline{t}$ and execution latency l of the variants A, B, and C for different reconfiguration intervals.

1.2 Classification of Reconfigurable Architectures

According to the literature [36][21][24][11][39][95][34], current reconfigurable architectures can be classified by the architecture properties described below. Note that an architecture possesses one or more of these properties.

1.2.1 Partial Reconfiguration

Partial reconfiguration is a feature that allows the reconfiguration of a subset of the device resources as opposed to full device reconfiguration. Partial reconfiguration can be realized if the configuration interface provides a mechanism to write new configuration data to selected device resources.

1.2.2 Runtime Reconfiguration (RTR)

This term describes the fact that the architecture can be reconfigured at runtime, i.e. while the device is active and the currently configured circuits are in use. It must be taken care that the reconfiguration does not interfere with the active circuitry. Non RTR devices are configured before they are activated with a so-called static configuration.

1.2.3 Multi-Context Configuration

Current commercial SRAM-based FPGAs contain a large on-chip memory that stores a single, active device configuration. Runtime reconfiguration speed can be improved if a device stores several device configurations at the same time, together with the possibility to select one of these configurations as an active configuration at a high rate. With multi-context configurations, reconfiguration rates in the order of nanoseconds are possible.

1.2.4 Fine-Grain Logic

Fine-Grain logic allows the implementation of functionality at bit-level. I.e. the fabric is organized such that the basic logic elements perform binary operations and the routing structure is able to route single-bit signals individually. In order to realize complex operations, word-level operations must be decomposed into bit-level operations. Fine-grain architectures allow a maximum of flexibility to implement digital circuits. However, each binary operation and the bit-level routing must be configured at a fine granularity as well, resulting in a large amount of configuration data.

1.2.5 Coarse-Grain Logic

Coarse-grain logic typically refers to reconfigurable architectures that realize word-level operations in the fabric. Word-level operations include simple arithmetic and logical operations, multiplication, and memory operations with fixed word width. The routing structure also implements configurable interconnect of fixed word length. Coarse-grain architectures are optimized for data processing and not for arbitrary digital logic. Because the logic configuration allows much less possibilities, the configuration data is significantly reduced.

Coarse-grain logic architectures can be divided further into configurable datapaths and weakly programmable processor arrays. Configurable datapath architectures consist of a fabric of ALU and memory elements and reconfigurable interconnect. The reconfiguration is initiated by a central configuration controller. The logic in the fabric can not control the mode of operation directly [66][58]. Weakly programmable processor array architectures consist of an array of simple, but small processors that are programmable [50][65][93]. Each processor contains a small instruction memory that configures the processor's operation, similar to a multi-context device. If the processor implements control operations like loops and branches then each processor in the array reconfigures itself by means of execution control. However, the instructions are loaded by a global configuration controller.

1.3 Reconfigurable Computing Specific Design Issues

We have already discussed that a fine grain architecture requires a huge amount of configuration data to program an entire device. In general, this is not an issue in coarse grain architectures. The amount of configuration can be reduced, if partial reconfiguration is employed. With partial configuration, only those resources in a device are reconfigured, which require changes to implement new functionality. Therefore the configuration contains only data for selected resources. Partial configuration requires support by the configuration architecture of the device as well as software support by the design tools. The development software must provide a mechanism to produce configuration data for the entire device as well as partial configuration data. The full configuration data is used to initialize the device at system start-up and the partial configuration data is used at runtime to reconfigure relevant parts of the device.

Partial runtime reconfiguration sets another requirement on the development software. In reconfigurable computing, the tasks are implemented as digital circuits that operate concurrently, distributed over the entire device. However, dynamic reconfiguration is not realized as an immediate, single-cycle reconfiguration of the device resources. During reconfiguration the digital circuit is transformed successively into the new circuit. In between, the active circuit may not perform a meaningful operation.

In Figure 1.6 the successive reconfiguration is illustrated. The new circuit is configured column by column which results in non-functional intermediate circuits. The busmacros prevent the intermediate circuit to interfere with the active part of the RSOC.

In runtime reconfigurable systems this problem is solved with the development software in conjunction with the runtime control of the RSOC. The designer must implement the reconfigurable tasks such that the tasks can be isolated from the RSOC during runtime reconfiguration, i.e. by disabling the bus interface. At runtime, the OS can disable the task before partial reconfiguration is performed. Furthermore the development software must place and route the RSOC design so that static parts are not affected by intermediate configurations that occur during reconfiguration.

In multi-context devices, the configuration data is loaded into a configuration context that is not active. The context is only activated when the reconfiguration is finished. Therefore, reconfiguration of the context is hidden to the logical operation of the device and thus no intermediate configuration can occur.

The huge amount of configuration data that is necessary to program a fine grain reconfigurable device has other implications as well. The configuration interface of a device has only a limited data bandwidth, which determines the speed at which

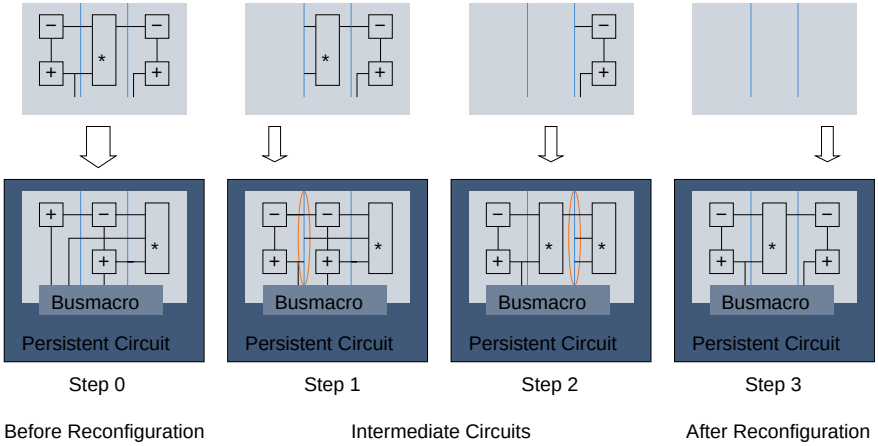


Figure 1.6: Partial reconfiguration in a single context device. The persistent, active circuit is isolated with busmacros to prevent interference from intermediate configurations. The new circuit is configured column by column which results in non-functional intermediate circuits.

new configuration data can be loaded into the device. The time required to load a new (partial) configuration determines the reasonable frequency of reconfiguration. Configuration caching and configuration prefetching techniques can mitigate the bandwidth limitations. The techniques are inspired by CPU caching strategies: In configuration caching, configuration data is kept as long as possible in the configuration memory of the device. If the data persists in the configuration memory until it is required for a reconfiguration later on, then the data must not be loaded over the relatively slow configuration interface. In configuration prefetching, the configuration data is loaded speculatively before the configuration is actually needed. Later on, the prefetched configuration can be activated much faster. Both configuration caching and prefetching require spare configuration memory to store the configuration data. These may be unused configuration contexts in multi-context devices or unused resources in a single-context device. The improvement depends on the performance of the reconfiguration prediction algorithms and the amount of available configuration memory.

Data compression is another option to increase reconfiguration speed. Because the configuration data exhibit a lot of redundancy, data compression can reduce the amount of data to store and transmit configurations. If the configuration architecture of a device supports compression directly, then the effective configuration bandwidth is increased and reconfiguration becomes more efficient.

Apart from the reconfiguration specific design issues, design productivity is a major concern in reconfigurable computing. Coarse grain reconfigurable architectures frequently provide a C-like programming language that is compiled and mapped to the device by the development software. Reconfigurable architectures can realize a high degree of instruction level parallelism. The development software needs to extract this instruction level parallelism from the C program in order to take full advantage of the target architecture. Today, synthesis from Hardware Description Languages (HDL) is the preferred design method for most fine grain reconfigurable architectures. HDLs require a detailed description of the datapath and control functionality that realizes a task, but it also offers unique control over the algorithm throughput and device resource usage. Using HDLs the design productivity is quite low. The RSOC design time can be considerably reduced by using system design tools (Xilinx EDK, Altera Nios II EDS) that integrate pre-designed IP Cores and custom-built HDL designs into an RSOC. High-level synthesis provides another method to implement tasks on reconfigurable architectures. Here behavioural VHDL or C-like programming languages are used to specify a task and the high-level synthesis tool translates this specification into an HDL design that is implemented on the reconfigurable architecture.

1.4 Overview of this Dissertation

In this work we focus on the most wide-spread class of reconfigurable architectures. Our methods are designed for the use with fine-grain, runtime reconfigurable architectures that support partial reconfiguration. Examples for such architectures are for instance the FPGAs (Spartan and Virtex) of Xilinx Inc. Our primary goal is to develop new methods for the design and implementation of RSOCs with minimal reconfiguration overhead on such architectures. Our methods can be applied to non-runtime reconfigurable architectures too, but only to a limited extend. Within this work we developed a number of design tools that are based on our methods. The tools enable the design of hardware tasks with minimal reconfiguration overhead. A recent summary of our work can also be found in [79].

This dissertation is structured as follows: In Chapter 2 we provide a comprehensive background on reconfigurable computing with references to other related work. We describe some examples of existing RSOCs and the design trade-offs involved. Then we summarize necessary technical details regarding the programmable logic and the reconfiguration mechanisms of the Xilinx VirtexII devices. We refer to this details later in this work. We also provide an overview about design entry, hardware task scheduling, and hardware task placement. Later we review some existing models to evaluate the efficiency of RSOCs in terms of energy consumption, resource utilization and application runtime. A larger part of the chapter is devoted to a review of different techniques that aim at the reduction of reconfiguration overhead.

We present the established methods for configuration data compression and for the reduction of reconfiguration overhead by exploiting the similarity of hardware tasks at several levels in the design process. Here we also point out what is missing in previous work and how the contributions presented in this work differ from previous approaches.

In Chapter 3 the fundamental modelling concepts that are used throughout this work are introduced. First an extensive motivational example is described in order to introduce the general mindset of our modelling concepts. Then we present the reconfiguration state graph that is used to model runtime reconfiguration. We provide a new concept to evaluate the reconfiguration overhead between different tasks. Our concept evaluates the finite differences between tasks. We derive two important measures for reconfiguration overhead: the overhead associated with both configuration data and reconfiguration time. The measures are applied to binary configuration data. A major contribution of this work is the application of the cost model to the structural representation of hardware tasks. Therefore we introduce a graph model for the structural representation and the notion of a virtual architecture. Finally, we present several methods how the structural representations of different hardware tasks can be mapped to a virtual architecture with the objective of minimal reconfiguration overhead.

In Chapter 4 we present two tools developed within this work. The tools can be used to implement hardware tasks, which are given as synthesized netlists, on an FPGA. During the implementation, the synthesized netlists are mapped to device specific netlists. First, we analyze different effects that occur in the mapping process. We show how the effects can be incorporated into a tool that computes the mapping of the synthesized netlists to a virtual architecture, based on the methods presented in Chapter 3. The mapping computed with the tool can be used as an input to our mapping tool. The mapping tool translates the synthesized netlists to device specific netlists. The translation is performed such that the resulting device specific netlists exhibit minimal reconfiguration cost for interconnect reconfiguration.

Chapter 5 is devoted to the domain of high-level synthesis. In this work we have developed a high-level synthesis tool that incorporates our reconfiguration cost models. The aim of the tool is the realization of hardware tasks as reconfigurable modules with minimum reconfiguration overhead. In Chapter 5, we introduce the design flow of our tool. The automatic synthesis of hardware tasks from high-level descriptions provides the possibility to apply several new concepts for the use of partial dynamic reconfiguration. We propose the synthesis of multiple tasks into one reconfigurable module in order to obtain a more efficient resource utilization and lower reconfiguration cost. Further we show how the reconfigurable modules can be adapted to different hardware tasks by dynamic reconfiguration with varying granularity: reconfigurable modules can be reconfigured to change the configuration of logic resources, of sub-modules within the reconfigurable module, and the control functionality can be adapted to support multiple hardware tasks. In order to

realize these concepts, we describe our approach for the synthesis of the datapaths in the reconfigurable modules. We extend previous synthesis models by the notion of the virtual architecture in order to incorporate our reconfiguration cost model. In the synthesis model we use a novel cost function that combines the implementation cost and the reconfiguration cost. We present several optimization strategies for the synthesis of the datapaths. The efficiency of our methods is demonstrated on a series of examples. The examples illustrate the application of our high-level synthesis tool to the proposed concepts for reconfigurable modules. The results provide insight into the trade-off between implementation cost and reconfiguration cost. In a final discussion we show the advantage of our new methodology compared to previous approaches.

Finally, in Chapter 6 we give a brief summary of our findings. This work is concluded with a proposal of a complete RSOC design flow that integrates our approach and existing methods. The primary aim of the design flow is an implementation with minimal reconfiguration overhead. We show that our tools are the key to such an RSOC design flow.

Chapter 2

Reconfigurable Computing Systems – Background

This chapter provides viable background information on reconfigurable computing systems. We review briefly some example RSOCs and discuss the related design trade-offs. We also describe the Xilinx VirtexII architecture in more detail in order to provide a profound understanding of the logic architecture and its versatility. In addition to the device architecture, we describe the partial dynamic reconfiguration mechanism. This highlights the restrictions in runtime reconfiguration for this architecture.

In the following, we will give a brief overview about major topics that are related to reconfigurable computing using FPGAs: design entry, reconfiguration/ task management, efficiency metrics, and methods to reduce reconfiguration overhead. An extensive discussion on previously published methods that optimize designs or configuration data follows next.

Finally we discuss the proposed techniques, their drawbacks, and open areas for research. We suggest a new methodology to tackle the major disadvantages of dynamic reconfiguration in FPGAs during the design and implementation process. Our methods targets the reduction of reconfiguration time and of configuration data. Thus, it is possible to create reconfigurable modules that can be configured much more efficiently.

2.1 Examples for RSOCs

In this section, reconfigurable computing systems are introduced from a system point of view. There are many existing realizations of reconfigurable systems. Even though reconfigurable devices itself can be found in many consumer and industrial products, runtime reconfigurability is rarely exploited. Runtime reconfigurable systems are often used in the context of academic research. These systems are either

general purpose development boards or systems especially designed for the needs of reconfigurability. A few popular examples are mentioned in the following.

In 2005 we presented a generic reconfigurable system [97] that contains a free area of reconfigurable resources. The area can be connected to the system bus of the static system. We also provided a Linux device driver that allows the embedded software to reconfigure the FPGA using the *internal configuration access port* (ICAP). As an example we realized two computational kernels that implement an integer transform and a motion estimation engine that can be used in MPEG4 video compression. The system is designed such that the software controls the reconfiguration and the computational kernels mapped to the reconfigurable area.

The Xilinx XUP board[111] has been used to realize a reconfigurable video processing system for automobile driver-assistance [18][20]. The reconfiguration in this system is controlled by one of the two PowerPC CPUs embedded in the FPGA device. The CPU triggers the reconfiguration that is performed by a hardware controller using ICAP The special design of the hardware controller allows a configuration data transfer rate that approaches the limits of the ICAP interface (8bit@66MHz) in the Virtex-II Pro devices. The configuration data is stored in an external DRAM. The video processing itself is performed in part by the software running on the PowerPC and in part by reconfigurable hardware accelerators. The realtime processing of these hardware accelerators is achieved by and efficient bus master operation, which requires no CPU control, and local storage of video data, which es very efficient for the selected algorithm. The selected application requires dynamic reconfiguration because the video processing algorithm depends on the driving environment, which is not known in advance.

The Erlangen Slot Machine (ESM) [53] is a complete hardware/software system that has been specifically designed for dynamic partial reconfiguration [12]. The ESM consists of two FPGA boards: A motherboard that hosts an embedded PowerPC CPU for system control and i/o interfaces for various video and audio standards. A babyboard contains a large Virtex-II 6000 FPGA as a reconfigurable device, several banks of SRAM, and a separate configuration controller. The configuration data is stored in a FLASH memory device on the babyboard. The ESM contains a special feature to allow flexible placement of reconfigurable modules on the FPGA. Many FPGA i/o pins is not connected directly to the motherboard's i/o interfaces, but to a programmable crossbar. Thus, the system software can connect the i/o interfaces to the reconfigurable modules depending on the module placement. The crossbar can also provide flexible interconnect between i/o pins of the FPGA. On the ESM, an embedded Linux runs on the CPU as central OS. The OS provides drivers to initiate partial reconfiguration, to set up the crossbar, and to transfer application data from the CPU to the FPGA. Several applications have been realized on the ESM: a reconfigurable video filter is provided as a tutorial, a car recognition video processor, and an object recognition algorithm [4].

All the above examples have the following building blocks in common: recon-

figurable device(s), large external memories for application and configuration data, and a configuration controller. An application designed for such a system is usually divided into hardware and software tasks. At runtime, the tasks are executed according to the needs of the application. Before a hardware task can run, the device must be configured such that it is capable to execute the assigned hardware task. The hardware task can require the whole device or only a fraction of it.

The design goal for any system is to achieve the most cost efficient solution for the given application. System cost is a function of device cost and memory cost: larger memory and more resources offered by a device increase system costs. Power dissipation increases with higher clock rates and for larger devices. Finally, execution time decreases with higher clock rates, more parallelism and increases by use of runtime reconfiguration because of the associated overhead. In the following, a more elaborate description of system design issues is given:

resources–time Often tasks can be implemented in space or time. By exploiting parallelism, more instructions can be executed in shorter time, which requires more computational resources. Vice versa, by executing the instructions more sequentially, less computational resources are required.

time–power High clock rates increases throughput, but also power dissipation in a device because of a higher switching activity in the circuit. Also, clock distribution consumes a considerable amount of power. Often, the operating voltage is increased to enable higher clock speeds, which also contributes to higher power consumption in the device.

reconfiguration–resources Reconfiguration offers the possibility to share resources in time. Consequently, less expensive devices are required to implement the same functionality. It is also possible to implement the application statically, which requires more resources but no reconfiguration – and thus no reconfiguration overhead.

memory–resources Sequential execution often requires less resources, but more memory to store intermediate data.

reconfiguration–memory Reconfiguration increases memory requirements, at first to store intermediate data between reconfigurable tasks in an application, and second, for the configuration data itself.

resources–power Power dissipation in electronic devices depends on the switching activity (dynamic power) and the die size (static power). It follows that devices providing more resources need a larger silicon area and consequently have a higher static power dissipation.

2.2 Partially Reconfigurable FPGAs: Xilinx Virtex Device Family

Xilinx Virtex FPGAs are a prominent example for today's reconfigurable devices. The Virtex device family has considerably evolved since 1998. In Table 2.1, major properties of the device series are presented. The table illustrates several major trends in FPGA architecture development: increase in total logic capacity, increase in logic complexity, and continued integration of macro blocks. The total functional capacity could be increased because of IC technology advancement. Smaller feature sizes allow the integration of more transistors per chip and the propagation delay of logic gates also decreases, which allows for higher clock frequencies. Unfortunately, the propagation delay of the interconnect does not decrease at the same rate. Instead, FPGA architectures incorporate special routes for the most frequently used direct connections. The introduction of specialized functions and more complex programmable logic also contributes to faster and smaller circuit realizations. Examples are the introduction of 6-Input LUTs, dedicated hardware multipliers/MAC units, and memory blocks. Integrated mixed signal functions (Clock Management, SerialIO and Ethernet MAC) reduce the complexity in system design and possibly the number of required external components. However, the specialized macro blocks and mixed signal functions occupy silicon area that can not be used by general reconfigurable logic. Therefore, FPGA vendors tailor the device capabilities to different application domains. In the newer generations, Xilinx provides device with different mixture of integrated functions: devices with higher capacity of general LUT logic, more MAC unite, more serial transceivers, or embedded PowerPC CPUs.

2.2.1 Virtex-II/Virtex-II Pro Logic Architecture

The programmable logic in a Xilinx FPGA is organized in Configurable Logic Blocks (CLBs) that are layed out on a regular two-dimensional array on the FPGA. Each CLB is composed of logic Slices and a switch box. The switch box provides programmable resources to connect both the logic to the routing network of the FPGA, different wires of the routing network itself.

The composition of the logic architecture differs substantially between device series. Here, we describe the Virtex-II/Virtex-II Pro device series [107][105] in greater detail because these series are used for all examples and benchmarks in this work.

A CLB consists of four slices and a switch box. The programmable logic is contained in the slices. Each slice consists of two, four-input LUTs (LUT F, LUT G) and two flipflops (FF Y, FF X). The LUTs can be configured to act as 16 bit ROM, 16 bit RAM, Shift Register or ordinary LUT. Both flipflops are driven by the same clock and can function as edge triggered registers or level triggered latches, both with different kinds of synchronous or asynchronous set/reset. Each slice also contains

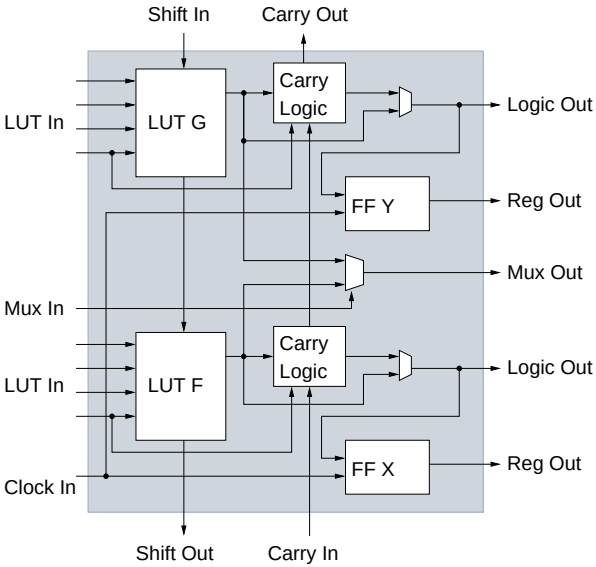


Figure 2.1: Simplified schematic of a Xilinx Virtex-II/Virtex-II Pro slice.

additional logic to allow an efficient implementation of large multiplexers, logic with carry propagation, and multipliers. A simplified schematic of a Virtex-II type slice is shown in Figure 2.1.

2.2.2 Reconfiguration Architecture and Reconfiguration Control

Here we describe the Virtex-II/Virtex-II Pro FPGAs from a reconfiguration point of view. The programmable logic of the FPGA is configured to realize a specific circuit. There is a distinct difference in terms of flexibility between the elements of the programmable logic architecture. E.g. the slice flipflops can be used as registers with synchronous set *or* reset. In circuit operation, the circuit logic directly controls the set or reset signal. However, whether the flipflops behave as synchronous set or reset registers is controlled by the configuration of the register.

The configuration of the FPGA is stored in SRAM memory cells that are distributed over the chip. The configuration architecture determines how the configuration is written to the memory cells and how the configuration defines the function of the programmable logic and interconnect. The configuration interface managed by a special *configuration controller*. The controller can be addressed through sev-

eral interfaces: a JTAG/ boundary scan programming mode, master and slave serial programming modes, and master and slave parallel programming modes. The slave parallel programming mode can be accessed from outside the FPGA via the SelectMAP interface or from inside the FPGA via the ICAP resource. The interfaces differ in the way the programming is controlled and in the data transfer bandwidth. The bandwidth ranges from 33 Mbit in boundary scan mode to 528 Mbit in the parallel programming modes. The configuration controller contains a set of control and status registers and is controlled by a set of configuration commands. A configuration bitstream for the FPGA is a sequence of data that is written to the configuration interface. The sequence contains a series of configuration commands and associated data. The data is written either into the control registers or to the FPGA configuration memory depending on the configuration command.

The configuration memory is organized as follows. The atomic unit of configuration is a *configuration frame*[103]. Each configuration frame is written to a memory location with a specific *frame address*. A complete configuration bitstream contains configuration data for all memory addresses of a device. Each resource on the device is configured by configuration data at specified frame addresses and bits within the configuration frame. Detailed information which pieces of configuration data configure which resources is not published by Xilinx. However, a few important properties are published that are relevant for our work, other details can be obtained through reverse engineering[64][109].

In Figure 2.2 the FPGA resources and the configuration memory addressing is shown for a simplified Virtex-II FPGA. The frame address encodes the configuration data block type, the major address, and the minor address. The block type differentiates between CLB logic and interconnect resources, BlockRAM memory contents, and global clock configuration. The major address selects the appropriate device column (CLB or BlockRAM column) and the minor address is used to address a subset of configuration memory within the device column. The number of major addresses depends on the size of the FPGA. Each CLB column is configured by 22 configuration frames. It can be assumed the association between slice resources and configuration data is equal across all CLB columns. However, this is not true for the configuration of the switch box because the hierarchical FPGA routing architecture is irregular with respect to the CLB column. In Figure 2.3 the association between some logic resources and the minor frame address is shown. It appears that the configuration of the slices in one CLB and hence of all slices in a CLB column can not be altered independently from each other.

The granularity of the configuration memory leads to the following situation: the reconfiguration overhead is the same for a resource in a single slice and for a resource in all slices of that column. This is especially useful for reconfigurable word level operations, were many logic resources of the same type are reconfigured at once. Both the logic and the reconfiguration architecture favour a vertical placement of the associated resources. It can also be observed that the reconfiguration of

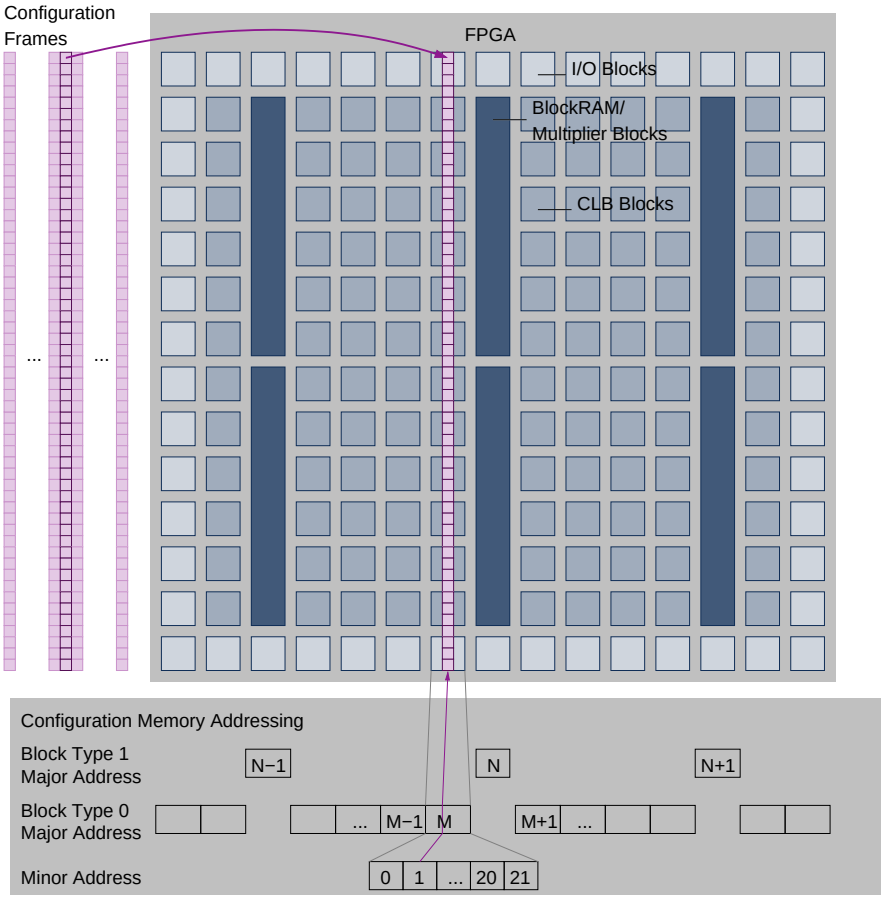
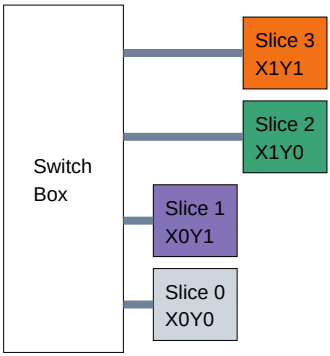


Figure 2.2: Configuration data organization and configuration memory addressing of the Virtex-II FPGAs.

the logic requires 6 configuration frames at the maximum, the configuration of the interconnect is contained in the frames 4–21. Thus, interconnect reconfiguration is more costly than logic reconfiguration.



(a)

Minor Frame Address	0				1				2				3				4				8			
CLB Slice	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
FF X, FF Y Type	✓	✓	✓	✓																				
FF X Init Value					✓	✓	✓	✓																
FF Y Init Value									✓	✓	✓	✓												
LUT F					✓	✓					✓	✓												
LUT G					✓	✓					✓	✓												
Slice Internal Routing	✓	✓	✓	✓										✓	✓									
Signal Inverters																	✓	✓	✓	✓	✓	✓	✓	✓

(b)

Figure 2.3: The association between logic resources and the minor frame address of a CLB column. (a) depicts the relative position of the slices within the CLB and (b) shows the resources within the slices that are configured with a particular configuration frame.

2.3 Methods for Design Entry

Design methods for reconfigurable systems can be done at different abstraction levels. In a software-like approach, the hardware task is described in a language borrowed from the software world, e.g. C/C++, or in application specific languages like Matlab/Simulink and SA-C [14]. The C-language derivatives often have special intrinsics to describe hardware-like behaviour. In a hardware design approach, circuit function is described in a hardware description language, i.e. VHDL or Verilog. The pros and cons of both approaches will be discussed at the end of the section. At first we will introduce both concepts in detail.

2.3.1 Behavioural Design Entry

The design entry is usually with done in a high level language. Popular examples are the ANSI-C derivatives Handel-C [1], Streams-C [33] and CatapultC [59]. The languages do not conform to a common standard, they are specific to the related design tools and libraries. Often, they are also specific to an architecture [8]. The languages provide macros to support special hardware related constructs, e.g. access to i/o-ports, extensions to describe parallelism in parallel loops, and timing with cycle constraints.

The tools offer some control over the compilation process that allows the designer to constrain design performance and resource allocation. After compilation, the design is represented in an intermediate representation. If the target is a fine grain architecture, then the intermediate representation might be a hardware description language or a target specific netlist. For coarse grain architectures the intermediate representation is closer related to an assembly language. Finally, a device mapper maps the constructs of the intermediate representation to specific resources in the architecture. The configuration data of the device can be directly generated from this mapping.

2.3.2 Design Entry at Register-Transfer Level (RTL)

This design style is very common for FPGA-like architectures. The behaviour of the hardware task is described in a hardware description language. These languages have an inherent model for parallelism and provide access to hardware functions at gate-level.

A synthesis tool compiles the hardware description into a device specific netlist, using target-dependent libraries. The RTL description can be target independent, at least to some extend. The description might contain instances of target-dependent modules to access special functions in the device. However most functionality can be described without those instances.

The compiled netlist is mapped to device resources with a device mapper. The result is a device specific netlist that contains instances of hardware architecture elements with the elements' configuration and the required connections. The compilation process succeeds by binding the elements to actual hardware resources, a step known as *placement*. A *router* realizes the connectivity by allocating appropriate wires between hardware resources.

The configuration of the device is fully specified after place and route. The configuration needs to be translated to a binary program, called *bitstream*, that contains the configuration data and the configuration commands for the target device.

Both methods for design entry have distinct advantages. Behavioural design entry allows high productivity in complex designs. Changes in the original specification can be incorporated quickly into an existing implementation. Behavioural

design entry is very tool dependent. RTL design is less flexible and productive, but designers can implement very optimized designs because the HDL allows a detailed description of the implemented architecture. HDL code is portable between vendors because VHDL/Verilog are standardized.

2.3.3 Xilinx Early Access Partial Reconfiguration (EAPR) Design Flow

Xilinx provides design tools and a special design flow specific for partial reconfiguration. The design flow is summarized in the following in order to explain available techniques and restrictions for partial reconfiguration that exist in current design tools. For more details refer to [106][108].

The EAPR flow provides guidelines on how to implement partial reconfigurable designs on Xilinx FPGAs. At first, the designer sets up a *top level design*. The top level design instantiates global resources, i/o-pins and static and partial reconfigurable *modules*. In addition, a floorplan for the top level design is created which assigns a placement to global resources and to the modules. The placement of the reconfigurable modules is fixed by this floorplan; they can only be placed freely at runtime with relocation methods [46]. The top level module also contains *busmacros* that provide a fixed interface between static and reconfigurable modules in the design.

In the next step, the static and reconfigurable modules are designed, verified and prepared for place and route by a synthesis tool.

Now, only the top level design is implemented, i.e. the design is placed and routed without the static or reconfigurable modules. This yields a static, base configuration of the device in which the modules will be integrated. Next each module is implemented independently of the other modules on top of the base configuration. In the merge step, all implemented modules are merged to complete configurations, from which the partial configuration data is generated. The complete configurations are used as initial device configurations and the partial configurations are used to reconfigure the device partially. The merge step produces as many initial and partial configurations as there are reconfigurable modules in the design.

The restrictions of the EAPR design flow clarify that partial reconfiguration in Xilinx devices is targeted at task-based reconfiguration. However, the partial configuration data can be processed to eliminate redundancy, see Section 2.5. At runtime, only configuration data that differs between configurations must be loaded into the device which allows further optimizations. The aim of the EAPR flow is to produce configuration data valid for partial reconfiguration. It provides predefined mechanisms to connect static and reconfigurable circuitry in a design. The reconfigurable modules are implemented independently, i.e similarities between the modules are not exploited in order to produce implementations with less reconfiguration overhead. Also, the design flow is not automated but must be carried out by the user

step by step. Many steps must be repeated if design changes must be incorporated. The runtime management of the reconfiguration is also completely left to the RSOC designer.

2.4 Task Management in Reconfigurable Computing

Applications use reconfigurable devices in different ways. With dynamic reconfiguration the device can be used as *virtual hardware* that provides different functions during application execution. Hardware virtualization is used in applications that exhibit a fairly static runtime behaviour, but where the full application would not fit into a single static configuration of the device. Other applications are inherently dynamic which means that they use several hardware functions in a way that can not be predicted at design time. One static hardware configuration is not efficient in that case because it requires to many resources. The dynamic allocation of reconfigurable resources during application runtime demands new execution control schemes that are not known from traditional computing systems.

The application can be specified as a set of tasks. The hardware tasks can be decomposed into subtasks where each subtask is executed on a circuit, implemented in a partial configuration of the reconfigurable device. Decomposing the complex task into subtasks is called *temporal partitioning* [11]. The execution of the subtasks requires (sub-) task scheduling and placement that respects both task dependencies and resource constraints. The scheduling and placement must be managed at runtime by some configuration controller which is often regarded as part of the reconfigurable system's operating system (OS) [15, 96, 87, 63, 35]. Design-time scheduling and placement is sometimes referred to as *temporal placement* in analogy to temporal partitioning. It can be applied only to applications with a task execution order known a priori.

Fu et al. [32] and Bazargan et al. [9] discuss the problem of task binding at runtime. An arriving task is executed on reconfigurable hardware, or on the host processor in software. In these approaches a task can have several hardware implementations with different resource demands, execution times, and energy consumption. The task scheduler has to bind the task to a suitable implementation.

In the following we present a summary of model properties that are relevant to the scheduling and placement strategies available in the literature. Most scheduling and placement algorithms have been developed with models that assume only some capabilities mentioned below. The scheduling and placement is usually handled together because both problems are highly interrelated.

2.4.1 Online and Offline Task Management

Task management can be divided into online and offline methods, In offline methods [9][92], the task scheduling and placement is performed at design time. These methods achieve very high quality results in terms of resource utilization and overall execution time, if the runtime and resource requirements of the tasks are known at design-time. Though the methods are usually computation intensive.

Online task management must be performed with limited knowledge of the tasks. Typically, the tasks arrive at a certain time and must be scheduled with the objective that they meet a given deadline or that the overall execution time of the application is minimized. Examples for these methods can be found in [2, 96, 9, 25, 15, 87]. In these works, several heuristics for scheduling and placement are described. They achieve high quality results with reasonable scheduling overhead. Chen et. al [16] show a typical area utilization of over 70 % with their approach.

There exist also combined approaches that combine online and offline task management in order to improve both application throughput and scheduling overhead [73][56].

The choice between online and offline task management depends on the application. If the tasks of an applications are executed in a predictable way, i.e. the tasks have a fixed execution time and known occurrence then offline scheduling is the best solution. Online methods are applied to dynamic applications or in a multitasking OS environment.

2.4.2 Task Scheduling

The function of the task scheduler is to arrange a number of tasks in time such that several requirements are met. The scheduler may have to observe data and control dependencies between tasks. Hence a task can not be scheduled before the task it depends on is finished. The scheduler can be designed to meet realtime requirements of an application, i.e. tasks must be finished at a certain *deadline*. Further, the total application execution time or *makespan* of all tasks can be of interest.

Many RSOCs are designed for dynamic real-time systems. Here, the tasks must be scheduled such that they are finished at a certain deadline. In this scenario, each task has a known execution time and a deadline to finish the execution. The schedulers in [2][96][9][87][92] decide on the basis of the current system load if the deadline can be met and accept or reject the task accordingly.

Some schedulers allow for *task preemption* [96][63] in order to improve the realtime behaviour of the system. This means a running task is interrupted and resources are assigned to another task. The realization of this feature requires the hardware tasks to store the current state of a running task, e.g. actively by the task itself or by using a state capture and readback mode of the device. After the

inserted task is finished, the preempted task is configured again and the state before task preemption is restored before the task continues it's execution.

In many applications, the individual tasks depend on each other. The tasks can have data dependencies, i.e. one task processes data that is produced by another task, and control dependencies, i.e. the execution of a task depends on the conditions produced by a previous task. Offline schedulers must take these dependencies into account as e.g. in [92]. In online scheduling, the dependencies between tasks are not known to the scheduler in an explicit way. Instead the tasks are presented to the scheduler when they are ready to be executed. Therefore, task dependencies can be neglected [5][2][96][25]. A hybrid approach of independent tasks scheduled at runtime that consist of dependent sub-tasks is presented in [73].

Even though the task execution can not be started until the data and control dependencies are satisfied it can be of advantage to the runtime behaviour that a task is configured before. Some schedulers [73][39] prefetch the configuration of tasks to speed up the execution.

The scheduler does not only need to manage the runtime behaviour of tasks but also the configuration port of the device as a unique resource. The configuration port is used to load configuration data into the reconfigurable resources of a device. Devices usually have only one configuration port. With partial configuration, it is assumed that only one task is reconfigured at a time [5][27]. Some models also support preemption of the resource reconfiguration [96].

2.4.3 Task Placement

Task placement describes the assignment of tasks to reconfigurable resources in the spacial domain. The device resources can be modelled as one-dimensional (1D) or two-dimensional (2D) array of resources. The 1D placement model is motivated by the partial reconfiguration mechanism in the Virtex and VirtexII-series devices [106].

The placement can be modeled such that the device area is partitioned into fixed blocks of resources. Alternatively, the placement model allows that tasks are placed anywhere on the device as unless the area is already occupied. Sometimes, fixed blocks can be joined to form larger blocks of resources to accommodate larger tasks. Several authors describe the use of a model that partitions the FPGA resources into fixed blocks of resources, e.g. [5][73][96][63][56]. Managing the reconfigurable resources as distinct blocks greatly simplifies the problem of task placement because only a limited number of possible placements exists. Thus the placement becomes similar to the problem of scheduling n tasks on m parallel processors, cf. [5]. On the contrary, several authors also describe methods that manage the FPGA resources as a homogeneous array of logic elements where the tasks can be freely placed [16][2][9][92][87]. The resources required by a task are defined as – in

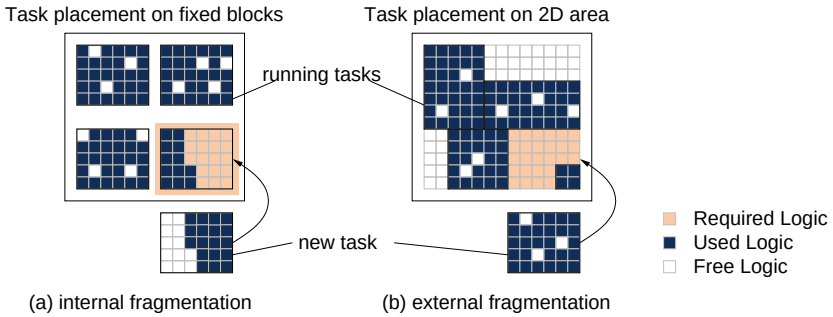


Figure 2.4: Internal and external fragmentation. In (a) the new task can not be placed because all slots are occupied, in (b) the new task can not be placed because the existing tasks are improperly placed. In both cases there are enough unused logic elements to run the new task.

case of a 2D resource model – a rectangular area of resources, or a column of device resources in case of a 1D resource model. If free placement of tasks is allowed for, an online task placement algorithm has to manage the allocation of device resources using a free space manager such that a suitable placement can be found quickly. The authors in [9][2] propose several heuristics where to place incoming tasks based on resource demand, free space, and task deadlines.

If the RSOC uses fixed blocks to place tasks then small tasks leave resources in such a block unused, leading to *internal fragmentation*. If free placement is applied there may occur small areas of unused resources between tasks during runtime, called *external fragmentation*. In consequence, the placer may fail to assign a rectangular area to a task even though enough free resources are available in the device. Both cases are depicted in Figure 2.4. Some models allow for online defragmentation [25] of the placement or task relocation [63], if necessary. Both techniques require the implementation of task preemption, too. Diessel [25] notes that task preemption and online defragmentation is only efficient if the task runtime is large compared to reconfiguration time. In FPGAs, configurations are specific for a task placement. If tasks must be configured elsewere, module relocation techniques are required, cf. [46].

Usually, due to the large effort to perform placement and routing of a task, the task implementations have a fixed layout. However, it is possible to adjust the task implementation dynamically at runtime to the available resources by using online routing [43].

The placement algorithm also needs to respect some requirements special to reconfigurable devices. Because the device resources are not necessarily homogeneous, some tasks can only be placed at specific positions in the device. E.g. the

tasks may require access to specific i/o resources, embedded memory blocks, and DSP blocks. Further, the hardware tasks must be integrated into the communication infrastructure on the RSOC. This can be, e.g. a fixed bus with a static interconnect infrastructure [87] or a reconfigurable network on chip [10].

Configuration reuse is the capability to use the same hardware configuration of some resources for the execution of another task or of another instance of the same task [16][73]. Configuration reuse is very similar to configuration caching [26] were the placer tries to keep partial configurations on chip until they can be reused. In the context of task scheduling and placement the configuration reuse usually means the reuse of the resource configuration of the entire area of a task. Thus configuration reuse is not exploited at logic element granularity [40] or at configuration frame granularity as in [20][54].

2.4.4 Reconfiguration Runtime Overhead

The runtime overhead that is caused by the dynamic reconfiguration is treated very different in the methods presented above. Frequently, the runtime overhead is completely neglected [2][9][87]. This may be a valid assumption if the reconfiguration can be performed very fast, as in [56], or if the reconfiguration time is sufficiently small compared to the task runtime. If the tasks are allocated to fixed blocks of resources, several authors assume a fixed reconfiguration time when such a partition is reconfigured, e.g. in [16][5][96]. In Teich et al. [92] it is assumed that a fixed reconfiguration time is included in the task's execution time. However, this prevents the scheduler from splitting the configuration of a task from its actual execution. Thus configuration prefetch can not be applied.

Note that the reconfiguration overhead metric must be consistent with the reconfiguration architecture of the device. Some approaches assume that the fixed blocks of resources or a whole device column is partially reconfigured, e.g. [5][63][56]. This type of partial reconfiguration can be applied to current architectures. It is supported by the EAPR design flow (see Section 2.3.3) and the reconfiguration architecture of the Xilinx Virtex series devices. The models that assume a rectangular area for each task demand that the related resource area can be reconfigured individually [16][92][87]. This requirement is not fullfiled in current partial reconfigurable architectures in the Virtex series, because the logic elements can not be configured individually as it was possible in Xilinx 6200 devices [102]. However, if 1D task placement is used, these models can be applied to the column-based reconfiguration architecture of the Virtex and VirtexII devices.

In practical realizations of RSOCs the reconfiguration overhead can be quite high. Angermeier et al. [5] quote a reconfiguration time of 130 ms for a task slot in a huge VirtexII-6000 device. This figure includes the OS overhead and the time for data transfer in their Erlangen Slot Machine [12]. In the best case individual

CLB columns could be configured in less than 0.4 ms. This shows that the reconfiguration time is critical for dynamic applications and there can be a a huge gap in reconfiguration overhead between device capabilities and practical realizations.

In our review on task scheduling and placement methods we observed that reconfiguration overhead is often neglected even though it is an important limitation of reconfigurable architectures. Moreover, there is no consideration of partial reconfiguration in the true sense, because partial reconfiguration is interpreted as reconfiguration of individual tasks, not of resources and interconnect. There are only few methods that include methods of configuration prefetch and reuse into scheduling and placement [16][73][39]. It is also interesting to note how the paradigm of partial configuration shifted along with the evolution of reconfigurable architectures. Early approaches consider small circuits or swapable logic units [15] as reconfigurable functions or reconfigurable instruction set extensions [100], today reconfigurable tasks are executed on complex hardware accelerators.

2.5 Configuration Data Compression

The configuration data for a fine grain reconfigurable architectures set up the behaviour of a huge number of reconfigurable resources at bit level. The logic elements itself contain – apart from LUTs and flipflops – many other reconfigurable resources. These resources control more specialized circuitry in the logic elements, e.g the use of distributed RAM, shift registers, carry chains, inverters, and multiplexers for internal routing. The configuration of those elements usually differs only from the default configuration if they are in use. In many designs the digital circuits which are mapped to the LE use only a fraction of the LE resources and routing switches. Hence a lot of configuration bits are set to the default value and are therefore redundant. Likewise, configuration data of resources in the same configuration and configuration data of resources in different configurations can be equal, too. The redundancy in the configuration data can be used by compression schemes to reduce the amount configuration data in an RSOC. The configuration data must be stored and transferred during runtime in an RSOC. Compressed configuration data provides an advantage to both. However, the configuration port of a device must support the decompression in order to be efficient. Frequently, the decompression scheme must be implemented in the reconfigurable part of the device because it is not supported directly by the configuration port. This overhead must be taken into account when implementing a compression scheme.

Apart from data compression without knowledge of the configuration data properties, the compression scheme can exploit several distinct properties of the configuration data. This may lead to more efficient compression or simpler and more resource efficient implementations of the compression scheme. In the following we describe redundancy properties that are specific to the frame-based bitstream format

of Xilinx Virtex devices.

Intra-Frame Redundancy Each configuration frame contains data for a column of resources of the same type. Several of these resource may use partially the same configuration. E.g. the LUT contents of all LEs that are part of an adder function can be identical. The redundancy may become only apparent if the assignment of configuration data to these resources is respected and not the rather arbitrary composition of binary data, i.e. the bit positions of relevant data in a frame must be known.

Inter-Frame Redundancy Using the same argument as before, the configuration data is partially identical across different frames if the LEs or routing switches are configured equally. Often, similar structures like logic functions and multiplexers are used repeatedly across the device.

Intra-Configuration Redundancy The compression scheme may exploit the intra-frame and the inter-frame redundancy within a single configuration bitstream only. This makes the decompression of the bitstreams for different configurations independent of each other.

Inter-Configuration Redundancy The compression ratio might be improved if the redundancy between different configuration bitstreams is exploited, too.

An early approach [38] to exploit redundancy in configuration data uses a device specific feature of the Xilinx 6200-series [102]. Using the "wildcard registers" the same configuration data could be written to the different resources simultaneously. There are other approaches that are targeted specifically at frame-based configuration data. Zhiyuan Li et al. [52] investigate configuration data compression using a dictionary based approach. In their work, the symbols in the dictionary are chosen to match the regularity in the configuration data of the reconfigurable resources. This enables an efficient use of intra-frame regularities. They propose a reordering algorithm to change the sequence of frame data such that the dictionary data provides good inter-frame regularity matches. The data compression after reordering is performed by a Lempel-Ziv based algorithm [112][88]. The authors [52] describe two variants, both rely on a modified configuration controller that must be implemented in the FPGA. The configuration frame buffer in the configuration controller is extended to hold two frames which serve as a dictionary for decompression. In their study the compare the two proposed compression methods to standard, entropy coding techniques, namely Huffman coding [45] and fixed precision arithmetic coding [101]. While entropy coders reduce the configured data to about 50–60 %, their proposed Lempel-Ziv variant with configuration frame reordering achieves 25 % on average, even for configurations with high resource utilization. The authors note that the manually placed designs achieve even better coding efficiency. We suspect that this arises from the higher regularity compared to designs placed by automated tools. The approach presented by Li [52] is taken one step further by Pan et al. [67]. The authors made two important modifications. Instead of the dictionary based compression they assume that the regularity

in the frame data is located at the same bit position in any two frames. Hence, they propose to encode the difference vector between two binary configuration frames using a run-length code combined with a Huffman coding of the run-length data. Frame data reordering is applied as before. As an extension targeted at runtime reconfigurable systems, the algorithm takes not only intra-configuration frame data into account, but also the configuration data of the already configured circuit in the device. This allows them to exploit inter-configuration redundancy for configuration data compression. However, this approach can be used only if the sequence of runtime configurations is known in advance. The authors in [67] observe that the inter-bitstream regularity is especially efficient if the sequence of configurations contains statically configured circuitry. They note that inter-configuration compression can be as much as 2× better than intra-configuration compression only. In [23] another dictionary-based configuration compression method is described. The authors propose to use global dictionary for several different configurations. However, their approach achieves only a small compression ratio – about 70 %. This may be due to their dictionary that is not tuned to the relation between configuration data and configurable resources in the FPGA.

A more fundamental analysis of the bitstream entropy is presented by Malik et al. [55]. They investigate how many configuration data deviates from the default configuration of a device. They observe that in many designs, even if the designs occupy a large amount of the device's logic resources, only few bits are changed compared to the default device configuration. The authors suggest that the run-length of zeros in the configuration data bitstream are random symbols, which is supported by statistic analysis of some example bitstreams. With only few relevant bits in the configuration data, the authors suggest that run-length based compression will always be more efficient than entropy coding like Huffman [45] and dictionary based methods like Lempel-Ziv [112][88]. They propose a hierarchical vector compression method that achieves an average compression of 10 % on the presented examples, which is very close to the theoretical bounds that are derived in [55]. In this paper, it is also suggested that hierarchical vector compression is well suited to a fast and area-efficient hardware implementation.

Hübner et al. [44][94] describe the implementation of a so-called LZSS decompression algorithm [88] in the reconfigurable hardware. They selected the LZSS algorithm because it achieves a good compression ratio (around 25 %), it can be implemented with high throughput, and the algorithm can be implemented resource efficient. The decompressor requires only 129 Virtex Slices and 1 BlockRAM. It can be clocked at 75 MHz and thus provides enough throughput to decode the compressed bitstream such that the reconfiguration speed is not limited by the decompression. An example for configuration data decompression using the embedded software in an RSOC is presented in [98].

Research in improving configuration data compression provides an insight into FPGA reconfiguration at bit-level. It can be concluded that the most efficient com-

pression schemes can also be implemented at low hardware cost for the decompression functionality. Although the approaches are difficult to compare directly — because there exits no set of standard bitstreams — the best known method by Malik et al. [55] achieves a compression of up to 10 % of the original configuration data size. The efficiency of the compression methods also supported by a more recent study on Virtex4 FPGAs [86]

The authors of the compression schemes also made an important observation that supports the approaches presented in our work. They observed that inter-configuration data compression becomes much more efficient if the configurations contain parts that remain static between configurations. Later on we will describe in detail how this can be achieved with an automated design flow.

2.6 Evaluation of Reconfigurable Systems

There are several approaches to evaluate the efficiency of reconfigurable computing systems. Efficiency can be measured for different system parameters: energy, area, and execution latency. Here we only summarize models that are targeted specifically at reconfigurable computing architectures. For a discussion on how reconfiguration overhead is incorporated into runtime management refer to Section 2.4.4.

2.6.1 Energy Efficiency Models

A major concern in today's computing systems is energy consumption. In line-powered systems the energy consumed in high-performance computing causes mainly thermal design challenges – in battery powered devices (mobile computing, wireless sensor networks etc.) the energy consumption determines the required battery capacity and the system runtime. Thus the computational requirements must be fulfilled with limited amount of energy. In instruction stream processors the operation and the data to be processed are controlled by a continuous instruction stream and hence the average energy efficiency can be given in million instructions per second per Watt (MIPS/W).

In [41] the *reconfigurable architectures energy throughput ratio* (RETR) is defined to quantify energy efficiency for reconfigurable computing. The metric is separated into reconfiguration of the hardware and the data processing itself. It is given by:

$$\text{RETR} = \frac{E_{\text{ex}} + E_{\text{rec}}}{T} \tag{2.1}$$

$$\text{RETR} = \left[\frac{C_{\text{ex}}}{(N_{\text{a}} \Phi_{\text{op}} up)^2} + \frac{C_{\text{rec}} \alpha R}{N_{\text{a}} \Phi_{\text{op}} up} \right] \frac{V_{\text{DD}}^2}{f_{\text{clk}}}. \tag{2.2}$$

The terms E_{ex} and E_{rec} denote the average energy consumption per operation for

data processing and reconfiguration, respectively. T is the throughput of the architecture. The terms in Equation 2.2 are defined as follows: C_{ex} and C_{rec} define the average effective switching capacity during data processing and reconfiguration, respectively; V_{DD} is the supply voltage, f_{clk} the systems clock frequency. N_a, Φ_{op}, u, p are the number of available operational resources, the operator's performance, the utilization factor of the operators and the penalty in execution delay caused by reconfiguration. α denotes the effective reconfiguration activity. R is the ratio of performed operations to reconfigured operations. In his work, Hinkelmann draws several notable conclusions regarding the energy efficiency of reconfigurable systems from the metric described by Equation 2.2. The main argument is to optimize both throughput and reconfiguration of the device to increase energy efficiency. Hinkelmann's conclusions are as follows:

- Increasing the number of available resources N_a will also increase C_{ex} and C_{rec}, but more resources allow to increase throughput. It is expected that the energy efficiency for reconfiguration will decrease at the same time.
- If the throughput Φ_{op} of the operators is increased, overall throughput increases, too. At the same time reconfiguration takes place more often.
- The overall throughput can also be increased if the execution is not delayed by reconfiguration. This can be achieved by using reconfiguration sparingly, enabling fast reconfiguration, or execute reconfiguration and execution in parallel.
- Reconfiguration efficiency can be increased if the redundancy in the configuration data is exploited for reconfiguration.
- Another method to increase reconfiguration efficiency is to separate reconfiguration that is required frequently and reconfiguration that remains constant over longer time periods [68].
- The granularity of reconfiguration is also important. If it is too high, reconfiguration becomes less efficient if only few operators must be reconfigured. On the other hand, a low reconfiguration granularity will increase the reconfiguration cost per operator, i.e. C_{rec} is increased.
- In multi-context reconfigurable architectures, the configurations are cached on-chip. This increases the reconfiguration efficiency by decreasing the execution delay required for reconfiguration and by reducing the energy consumption for loading configuration data from off-chip memory.
- The number of reconfigurations and hence the energy for reconfiguration E_{rec} also depends on the available resources N_a. If the same functionality is implemented in a device with less resources, reconfigurations occur more frequently which increases E_{rec}.

2.6.2 Area Efficiency Models

Another approach to measure the computational efficiency has been developed by DeHon [24]. The main interest of his model is the area efficiency of a reconfigurable computer for general purpose computing. He proposes the so-called *RP*-space model that allows him to compute several efficiency measures. The *functional density* F_{density} is defined as the number of gate evaluations N_{ge}, e.g. 4-input LUTs, per unit space-time $t_{\text{cycle}} \cdot A$:

$$F_{\text{density}} = \frac{N_{\text{ge}}}{t_{\text{cycle}} \cdot A}. \tag{2.3}$$

Similarly, DeHon defines the *functional diversity* or *instruction density* I_{density} as the number of distinct function descriptions $N_{\text{instruction}}$ that are present per unit area A:

$$I_{\text{density}} = \frac{N_{\text{instruction}}}{A}. \tag{2.4}$$

The *RP*-space model describes an estimation function for the required device area that depends on several architectural parameters: the number of processing elements, the datapath width, the number of on-chip instructions (or contexts), the size of the instruction word and the data memory. The total area is composed of the device area allocated to interconnect, instruction memory, data memory and control.

The model provides guidelines for the design of reconfigurable architectures if some of the aforementioned parameters are known for an application domain. These guidelines give hints to design an architecture such that the functional density and the instruction density is acceptable for a large range of applications. DeHon proposes the general rule that the instruction memory should account for one half of the processing cell area.

A major advantage of the *RP*-space model is that it allows to relate the functional density of reconfigurable architectures to other general purpose computing architectures. It is found that the functional density of FPGAs can be up to 100 times better then general purpose processors in regular, highly pipelined computations.

The model calculates the functional diversity that originates from the configuration (multi-context) memory inside the architecture only. The increase in functional density that can be achieved with runtime reconfiguration is not covered.

2.6.3 Runtime Efficiency Models

Wirthlin et al. [99] investigate the functional density of statically versus runtime reconfigured circuits. Therefore the reconfiguration time t_{rec} is introduced into the

functional density metric:

$$F_{\text{density,rec}} = \frac{N_{\text{ge}}}{(t_{\text{cycle}} + t_{\text{rec}}) \cdot A} \tag{2.5}$$

$$F_{\text{density,rec}} = \frac{N_{\text{ge}}}{t_{\text{cycle}}(1+f) \cdot A} \quad \text{with } f = \frac{t_{\text{rec}}}{t_{\text{cycle}}} \tag{2.6}$$

The equations above suggests that the relationship between reconfiguration time and execution time affects the functional density. Hence, if the reconfiguration time is small compared to the execution time, the increase in functional density is more prominent. Note that the advantage of a runtime reconfigurable circuit stems from the possibly smaller area and less execution time of a task.The theoretical maximum improvement is achieved if the reconfiguration time can be neglected, i.e. $F_{\text{density,max}} = \lim_{f \to 0} F_{\text{density,rec}}$.

More important is the relationship of the functional density between the statically and the runtime reconfigurable circuit. In order to be more efficient, the functional density of the runtime reconfigurable circuit must be higher than the functional density of the statically configured circuit, i.e. $F_{\text{density,rec}} \geq F_{\text{density}}$. This yields by substitution of $F_{\text{density,max}}$:

$$\frac{F_{\text{density,max}}}{F_{\text{density}}} - 1 \geq f. \tag{2.7}$$

In [99], the authors conclude that the maximum allowable configuration ratio f must be less then the maximum potential improvement in functional density, in order to be more efficient. They suggest that if a runtime reconfigurable circuit has a greater advantage over a static circuit then the reconfiguration time is a less important limitation.

2.7 Similarity Based Reduction of Reconfiguration Overhead

We have already seen that reconfiguration overhead can be reduced by compression of the configuration data and by smart scheduling of the reconfiguration which includes configuration reuse and configuration prefetch. While the data compression incorporates inherent properties of the configuration data, the scheduling techniques use a very coarse model of the reconfiguration overhead. In the following we describe approaches that consider either the reuse of individual resources or interconnect within applications or increase the reuse of these resources by specific circuit design or mapping techniques. Both techniques are very closely related to our work. We will highlight the notable difference in the relevant sections later on.

The improvements in the reuse of resources inside tasks can increase the efficiency of both aforementioned methods. Compression can be higher because inter-configuration redundancy is increased. Further the application execution latency is decreased by partial resource reuse, too. If resource reuse is taken into account, the reconfiguration overhead depends not only on the size of the reconfigured area for a task, but it also depends on how much configuration data must be loaded to transform the configuration of one task to another. More details on that will be given in Chapter 3.

2.7.1 Configuration Data Generation Methods

One trivial method to take advantage of partial reconfiguration is the direct comparison of configuration frames. Claus et al. [18][20] developed a framework in which all configuration bitstreams are compared to each other. If the data of a configuration frame differs between any two configurations then this frame is considered a dynamic configuration frame. The tool produces new bitstreams for each configuration, where the bitstreams contain only dynamic configuration frames. The static frames are configured with the initial configuration. The methods aims to reduce the bitstream size and hence the configuration overhead. With this method, the bitstream size is not equivalent to the area occupied by a task but it depends on the differences of the configuration of resources that are associated to a configuration frame. For a detailed comparison to our approach cf. [77].

Kennedy [48] proposed a method that exploits unused interconnect and logic resources in a configuration bitstream. He presents a detailed analysis of the differences in configuration bits and relates those differences to different classes of FPGA resources. He finds that most differences occur in the LUT contents and in the configuration of the routing multiplexers that drive the LUT inputs. For typical designs, the number of bit-level differences between two configurations is about 8–10 % over all resources. For the reduction of differences in the bitstream he proposes the following method: Consider the reconfiguration from configuration A to B. Any configuration bits that are contained in B and configure resources that are unused and thus do not interfere with any circuitry in A, are added to A. Now we have configuration A^+ which contains these *advance configuration bits*. The reconfiguration from A to B is replaced by a reconfiguration from A^+ to B. Reconfiguration from A^+ to B is more efficient because some circuitry of design B is already present in the current configuration A^+. Thus on reconfiguration fewer configuration data must be loaded into the device. Kennedy notes that, by taking into account only some resource classes, about 10 % of the bit-differences can be turned into advance configuration bits. The method takes advantage of the fact that the resources – especially for interconnect – in an FPGA are highly under-utilized. Hence there is a high probability that resources can be configured in advance without interfering

with the active configuration. However, it is not investigated what secondary effects these randomly configured resources can have in terms of power consumption. Also the method relies on random similarities between two circuits and random unused resources. With the approaches presented in our work, we can identify similarities in the original design specification. It is then possible to produce configurations that have a higher similarity in the configuration data. Our approach also considers more than two reconfigurable designs.

2.7.2 Device Mapping Methods

We already observed that configuration data itself depends on the mapping of the digital circuit to the reconfigurable resources. Here were present existing methods that perform circuit mapping with the aim of reduced configuration overhead.

The following method tries to reduce the amount of configuration data for individual configurations, independent of each other. Tan et al. [90] propose a set of guidelines for the placement of logic elements such that fewer configuration frames are required by the implementation. Although the authors claim a reduction of configuration data of 30 % for a set of very simple examples, it seems that this method is not useful for more complex designs. This method may be efficient if the circuit occupies only a some of the resources provided by a partially reconfigurable area, hence it can only be applied to modules with high internal fragmentation. The authors mention that only 2 out of 22 configuration frames are related to LUT contents in a VirtexII architecture. We conclude that, even if there are unused columns of LUTs in an area, the circuit router will produce implementations that use routing resources in these unused columns. Hence 20 out of 22 frames can still carry configuration bits, which renders this approach quite useless.

An alternative placement method that exploits the similarity between circuits is described by Shirazi et al. [84]. The authors propose a heuristic to find the similarity in two circuits. The method is based on bipartite weighted graph matching of the elements in both circuits. Any two elements in the circuits that can be implemented on the same device resources can be matched in the bipartite graph. The weights of the edges in the bipartite graph are calculated from three terms: the similarity of the logic function, the placement on the device, and a metric that describes the similarity in the connectivity. The results of these automated similarity matching is used twofold: to produce partially reconfigurable designs with minimal reconfiguration cost or to produce designs with both circuits integrated such that a dedicated control signal can switch the functionality of the implementation. The last method is very similar to the approach in [61]. According to the results the method is able to identify good matchings between the structure of similar circuits. However, the given examples are simple and exhibit a similar structure in general. It is unclear how this method performs for larger, more complex circuits. The similarity in inter-

connect is only identified if the nodes have a similar position in the circuits. Instead we propose that the type of interconnect determines whether an interconnect must be partially reconfigured, cf. Section 3.4.

Huang et al. [42] present a mapping method that aims to reduce the reconfigurable interconnect overhead. Although they target a different architecture model, their method is still relevant to our research. The authors use a high level synthesis tool to generate datapath descriptions for kernel loops (similar to tasks in our model) from a high level language. The datapath description is then mapped to a reconfigurable datapath that consists of fixed functional units, registers, a reconfigurable control unit, and reconfigurable interconnect. The mapping method translates the datapath description of the tasks into a directed graph, the task graph. Two such tasks are then mapped to a merged graph using a bipartite weighted matching. The task graphs are subgraphs of the merged graph. The bipartite weighted matching is used as a heuristic to minimize the number of edges in the graph and hence the number of reconfigurable interconnect in the final datapath implementation. The method has several drawbacks for interconnect minimization: The datapath mapping is performed after resource binding which hides part of the optimization space. The datapaths are merged successively, thus the result depends in the processing order and no global optimization is performed. The weight assignment heuristic takes into account only the possibility of interconnect sharing for independent nodes. However, if there are many instances of the same resource type then the realization of interconnect sharing becomes less likely. In the ADPCM codec example, the authors report an interconnect reduction of 22 % compared to a datapath without interconnect sharing.

The partial configuration of routing is considered in Rakhmatov et al. [71]. Here it is assumed that a configuration occupies a number of 1D channels in a routing track. The method identifies a channel assignment for another configuration such that the channels overlap as much as possible. Hence only few switches must be dynamically reconfigured to adapt the channel routing to a new configuration.

An example on how the configuration data architecture and the device mapping can be used to reduce configuration overhead for LUT contents is described by Raghuraman et al. [70] for Xilinx Virtex devices [104]. The authors propose a technique to map the logic functions to LUT tables such that the difference between successive configurations is minimal in the related reconfiguration frames. The method is limited to reconfigurable devices where the LUT configuration bits (1–16) are stored in separate configuration frames.

2.7.3 Circuit Design Methods

The device mapping methods in the previous section depend very much on the similarity of the structure in the input circuits. The structure of a design can be

improved by manual circuit design or by automated methods. In this section we will review several methods that produce designs that are optimized for dynamic reconfiguration.

Merged Dataflow Graphs

Moreano et al. [61] describe a technique to synthesize reconfigurable datapaths from a set of dataflow graphs (DFGs). The dataflow graphs represent different tasks that must be executed by the application. It is assumed that these tasks can time-share the same hardware by means of runtime reconfiguration. The method can be outlined as follows. The input dataflow graphs are mapped successively to a *merged dataflow graph*. The target of the mapping is a datapath for this merged dataflow graph with minimum area. The problem is described as a *compatibility graph* that contains nodes for each pair of DFG nodes and edges which can share a datapath resource. Nodes in the compatibility graph are weighted with the area reduction achieved if this resource sharing is employed. Resource sharings that can be realized concurrently are connected by an edge in the compatibility graph. The authors propose that the maximum weighted clique in the compatibility graph represents the merged DFG that yields a datapath implementation with minimum area. The datapath can be reconfigured by setting up the steering multiplexers accordingly. Thus, the method produces a custom coarse grain reconfigurable design that can be implemented as IP in an ASIC or FPGA. Because the reconfiguration is performed by configuring multiplexers only, the implementation requires only a small configuration memory.

The method considers no intra-DFG resource sharing, but only shared resources between configurations. Hence, resource selection and resource binding of each node in the merged DFG becomes trivial. The method explicitly models the interconnect area associated to the dataflow multiplexers. In order to reduce datapath size further, the method takes into account the commutativity of operations. The authors provide an interesting comparison to another approach in which the DFGs are combined in the high-level description for a commercial high-level synthesis tool. The authors claim that their datapath merging approach achieves about 20 % lower area compared to the result of the Synopsys Design Compiler for combined HDL code.

Common Subgraph Extraction

In contrast to the merged DFGs the authors in [6] propose a partitioning method for reconfigurable DFGs. They describe a method to extract a *dominant common subgraph* from two DFGs and assume that this common subgraph can be implemented with low reconfiguration cost. The remaining parts of the DFGs are implemented as reconfigurable modules. The common subgraph is scheduled in the context of the

original DFGs for each task. Next, the common subgraph is mapped to a datapath which can be configured to be part of the full datapath implementation of the original DFGs. The authors highlight that the common subgraph does not necessarily result in a static datapath module because the original DFGs define different constrains on the datapath operation schedules. The method reduces reconfiguration overhead because the dynamic reconfiguration of the common part is minimized. However it does not take advantage of the similarity in the rest of the DFGs. Resource sharing between configurations is only exploited for the common subgraph. Furthermore resource selection and binding possibilities are not investigated in this method.

Architectural Template

Another approach to optimize the design of reconfigurable circuits with low reconfiguration overhead is described by Heron et al. [40]. They present a *design for reconfiguration flow*. At each stage in the design flow, the new design is compared to existsing designs in a library in order to identify common, static circuitry. The authors propose that *commonality* of circuits is a key design goal for fine grain reconfigurable circuits. They propose the use of a common architectural style for all circuit blocks in a design and an efficient floorplanning strategy. As an example, circuits for multiplication, division and square root targeted at the Xilinx 6200 series are developed manually. The result is a regular array of logic for each function. The relatively simple architecture of this FPGA surely was an advantage for this methodology. The authors demonstrated an overall reduction in reconfiguration overhead between those functions. However it is observed that, when a large circuit is replaced incrementally by a smaller circuit, considerable overhead is spent on removing the large circuit. For example an FIR-Filter MAC-unit the reconfiguration time could be reduced from 1500 μs to 322 μs with partial reconfiguration.

Temporal Placement

Boden et al. [13] use a special High Level Synthesis framework to implement applications on partially reconfigurable architectures. The application is compiled to a *binary macro tree* representation. The representation contains control, dataflow and reconfiguration operations. On this tree a temporal modularization is performed in which the application is partitioned into configurations on the time axis and *temporal reusable modules* that are implemented as partially reconfigurable modules in the device. The authors derive a placement of functional modules such that *temporal sharing*, i.e. inter-configuration reuse of the modules is exploited in order to reduce reconfiguration overhead. Although this is an interesting approach, the authors do not reveal how this is accomplished in detail. Furthermore, their model of reconfiguration differs from our model: we assume applications that are already

partitioned into tasks and want to exploit the similarities in those tasks to reduce reconfiguration overhead.

2.7.4 Model for Partial Configuration

The reconfiguration model used by Heron et al. [40][82] describes the configuration of a device as a *configuration state*. The authors introduce a *reconfiguration state graph* (RSG) which describes the reconfiguration of a device as a transition between two configuration states. The model is used for partial reconfiguration with a *virtual hardware handler* that manages the change of the configuration state. The virtual hardware handler knows about the current configuration state and the difference in terms of configuration data to the next configuration state. Thus, the handler is able to perform partial reconfiguration by loading the new configuration data only - in contrast to approaches that always load the configuration data of a complete model. With their experimental setup, a Xilinx 6200-based system, the authors observed a decrease in configuration overhead by leaving configuration data of the previous state. The authors also explore the possibility to reuse remains of any previous configuration that occurs when traversing the RSG.

The RSG model describes the reconfiguration of a device during runtime in a very intuitive way. We adapted this model to describe partial reconfiguration, too. Instead of using the RSG model for runtime management only, we employ the model in the circuit design phase as well.

2.8 Contributions of this Work

In the beginning of this chapter we presented a few RSOCs that employ partial dynamic reconfiguration of FPGAs. We observed, when mapping applications onto these RSOCs, only one design methodology is supported by the vendor's implementation tools: the design of partial reconfigurable modules using the EAPR design flow. The partial configurations occupy a fixed area of device resources, which increases internal fragmentation and thus can lead to inefficient implementations. Also, modules are difficult to relocate. The effort to design and implement reconfigurable modules is usually high. In practice, designers have no capacity to evaluate different implementations regarding the overall efficiency. We also observed that the reconfiguration overhead using standard module based reconfiguration is high — both in terms of reconfiguration time and configuration data storage. Both can be reduced with appropriate methods: configuration data can be reduced with data compression and reconfiguration time can be reduced by smart scheduling and placement. However both methods are not targeting the origins of reconfiguration overhead, they are merely treating the symptoms. In this work we develop methods

that allow us to decrease configuration overhead during circuit design. The methods are orthogonal to the data compression and scheduling methods described in this chapter.

Existing theoretic models predict a twofold gain when reducing reconfiguration overhead: The RETR model predicts a higher energy efficiency because the switching activity due to reconfiguration is reduced. The runtime efficiency model predicts a higher functional density that results in either less area for the implementation or in faster overall throughput.

There exist other approaches that try to minimize differences in the configuration data: either by placement and routing or by design for similarity. However we found the proposed methods to be incomplete, which results in major limitations regarding the applicability and the efficiency. Either placement and routing does not take into account the similarity between configurations, the methods consider only two configurations at a time or the measure for similarity does not fit well into the resources configuration model of the devices. Moreover, the authors often propose only a method with a secondary objective, without defining a concise cost model for partial reconfiguration as e.g. the RSG. From the designers point of view, there exists no continuous design flow that allows to evaluate potential reconfiguration cost at all levels and that allows to evaluate trade-offs between reconfigurable and static implementations.

In this work, we employ a uniform reconfiguration cost model that can be applied at all levels of design and implementation. It provides precise information how much of the device must be reconfigured between tasks. Our model is suiteable to do both: a calculation of reconfiguration costs and an optimization of the implementation regarding those costs. The model can be applied at different design phases: during high-level synthesis, in the mapping phase, the place-and-route phase, and for generating the final bitstreams. Our high-level synthesis framework provides a unique method evaluate the trade-offs between static and reconfigurable implementations. We also provide several solutions in the design space with different reconfiguration overhead and resource requirement trade-offs, which have not been explored before. Further, we propose the separation of tasks and configurations, because one configuration may contain the functionality required for multiple tasks. Our method allows to optimize overhead in interconnect reconfiguration and in logic reconfiguration.

Table 2.1: Xilinx Virtex device families. MAC denotes the Multiply-Accumulate units, Gb Serial IOs are serial gigabit transceiver blocks, and Ethernet MACs provide blocks for ethernet media-access control.

Virtex Family	Year	Technology (nm)	Slices/ CLB	LUT Inputs	Largest Device	LUTs	Memory (Kbits)	Reconfiguration	Macro Blocks
Virtex	1998	220	2	4	xcv1000	24576	128	CLB Column	Memory
Virtex-E	1999	180	2	4	xcv3200E	64896	832	CLB Column	Memory
Virtex-II	2000	150/120	4	4	xc2v8000	93184	3024	CLB Column	Memory, Multiplier
Virtex-II Pro	2002	130/90	4	4	xc2vp100	88192	7992	CLB Column	Memory, Multiplier, CPU, Gb Serial IO
Virtex-4	2004	90	4	4	xc4vlx200	178176	6048	16 CLB/ Column	Memory, MAC, CPU, Gb Serial IO, Ethernet MAC
Virtex-5	2006	65	2	6	xc5vlx330	207360	10368	20 CLB/ Column	Memory, MAC, CPU, Gb Serial IO, Ethernet MAC
Virtex-6	2009	40	2	6	xc6vlx760	474240	25920	?	Memory, MAC, Gb Serial IO

Chapter 3

Runtime Reconfiguration Cost and Optimization Methods

Appropriate models are essential to assess the cost associated with runtime reconfiguration, especially in comparison to static implementations. Cost models are also used to measure improvements in reconfiguration costs achieved by any design methodology. The models introduced here describe the cost associated directly with the dynamic reconfiguration at runtime. The overhead caused by system design efforts or additional resources to enable runtime reconfiguration are discussed elsewere, e.g. in [19].

The main objective of this chapter is to provide a formal model for reconfiguration cost, which is not specific for any reconfigurable architecture. It enables us to assess the reconfiguration effort that is caused by different implementations of reconfigurable tasks independently of the reconfigurable device. It appears that runtime reconfiguration cost can be estimated directly for the binary configuration data. However we provide a new approach that allows the us to assess and optimize the designs for runtime reconfiguration during the design implementation stages.

In this chapter, we review the idea of a reconfiguration state graph (RSG) introduced by Heron et al. [40]. Here, we extend the RSG substantially in order to describe the device configuration and the reconfiguration requirements in detail. These properties allow us to assess the reconfiguration overhead based on binary configuration data directly. We introduce a new formalism that allows us to compute the reconfiguration cost for structural design representations, e.g. netlists and dataflow graphs. Based on this model we investigate how the mapping of the designs to an architecture influences the reconfiguration cost. Based hereon we describe several methods to optimize the mapping in order to achieve minimal reconfiguration cost. The chapter is concluded with a series of experiments that demonstrate the potential of our approach.

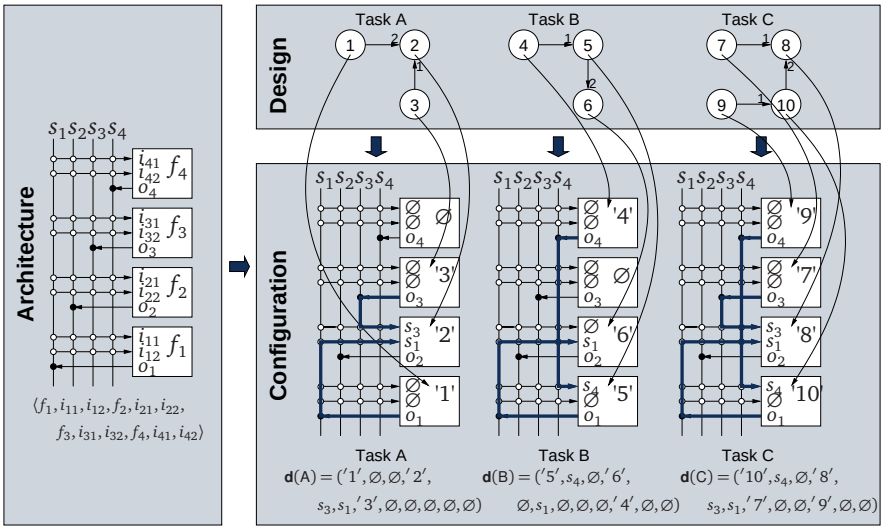


Figure 3.1: An example reconfigurable architecture, three (reconfigurable) designs and the configurations that result from a specific design-to-architecture mapping.

3.1 Motivation

Before we introduce the RSG model in all details, we present an example that illustrates some important aspects of device reconfiguration from the implementation tools' point of view. The example motivates our generic notation of the device configuration. Further we give hints on the mapping process and on the challenge of configuration cost assessment.

For now we assume a simple, reconfigurable architecture as shown in Figure 3.1 (left). The architecture features four identical functional units (FUs). Each unit contains a configurable function f_x and two programmable input ports i_{x1} and i_{x2}. The function f_x can drive the output signal o_x directly to an individual vertical wire s_x. Each input signal can be driven by one of the vertical wires $s_1 \ldots s_4$. The configuration of the architecture is specified by 12 reconfigurable elements. The resources associated with the reconfigurable elements are listed in the 12-tuple $\langle f_1, i_{11}, i_{12}, \ldots, f_4, i_{41}, i_{42} \rangle$. The configurable function f_x is not specified further in this example, but the input signals i_{x1}, i_{x2} may take any value from $\{s_1, s_2, s_3, s_4\}$.

The example designs in the top of Figure 3.1 constitute tasks that are mapped to the architecture. At runtime, the mapped tasks will be configured on the device on demand. We assume that the designs comprise a set $\mathcal{N}_\mathsf{T}$ of tasks. In our example $\mathcal{N}_\mathsf{T} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. The designs are represented as graphs which is a reasonable ab-

straction from the original netlist format. The nodes represent logical or arithmetic functions that can be executed on the configurable functions in the architecture. The edges indicate a transfer of data between functions or nodes, respectively. The edges are labelled with the targeted input port. E.g. node '2' of Task A receives data from node '3' for on port 1 and data from node '1' on port 2. The designs contain only information on the configuration of individual functions but this information is unrelated to the device configuration at this stage.

A device configuration is derived from the design by a mapping step. A mapping describes how the nodes of a design are realized on the resources of an architecture and how the data transfers between resources are realized using the interconnect. In our example in Figure 3.1, the nodes from the designs are mapped to resources f_1 to f_4 and the data transfers are realizes by connecting a vertical wire s_1 to s_4 to the input pin of a resource. Consider the edge $(1,2)$ in Task A: the node '1' is mapped to f_1 and the node '2' to f_2. The output o_1 drives bus s_1 directly. Hence, the input i_{22} must be connected to s_1 in order to receive the data from f_1 as indicated by the design. In Figure 3.1 the mapping of nodes to resources for all designs is indicated by the arcs from nodes to resources.

The mapping of a design specifies how a device is configured in order to realize the given functionality. Hence, the configurations that are used at runtime are known in detail after mapping. In this work, we denote a device configuration with a vector $\mathbf{d}(n) = (\mathsf{d}(n)_1, \mathsf{d}(n)_2, \ldots, \mathsf{d}(n)_m)$ that is a function of the task n with $n \in \mathcal{N}_\mathsf{T}$. The vector element $\mathsf{d}(n)_k$, $k = 1 \ldots m$ describes the configuration data for the reconfigurable element k. In our example we assume that all resources and input select switches are independently reconfigurable elements, i.e. $m = 12$. The relationship between the reconfigurable elements k and the configurable resources are given by the 12-tuple $\langle f_1, i_{11}, i_{12}, \ldots, f_4, i_{41}, i_{42} \rangle$ in Figure 3.1: $\mathsf{d(A)}_1$ describes the configuration of resource f_1 for Task A, $\mathsf{d(B)}_2$ describes the configuration of resource i_{11} for Task B etc. The complete configurations for the tasks A, B, C are given by $\mathbf{d}(\mathsf{A})$,$\mathbf{d}(\mathsf{B})$, and $\mathbf{d}(\mathsf{C})$, respectively. The symbol $\varnothing$ denotes the default configuration of the appropriate resource.

Finally, the process of device reconfiguration is illustrated on our example from Figure 3.1. At power up, the device is reset to an initial state with all reconfigurable elements being $\varnothing$. Hence, if e.g. Task A is loaded first then at least the reconfigurable elements $k = 1, 4, 5, 6, 7$ must be configured with the appropriate configuration data $\mathsf{d(A)}_k$. In this case, partial reconfiguration is used. Alternatively, all reconfigurable elements $k = 1, \ldots, 12$ can be loaded using $\mathbf{d}(\mathsf{A})$, which is called full reconfiguration. Apparently, full reconfiguration is independent of the previous configuration and does not require additional information about which elements must be reconfigured. A major drawback is the number of elements that are reconfigured: in our example all 12 elements are loaded instead of just 5 elements as with partial reconfiguration. In the case of partial reconfiguration, it must be known at runtime (1) what is the current configuration of the device and (2) which elements must be

reconfigured with what data to realize a new configuration. The efficiency of both approaches depends on the configuration architecture of the device, i.e. how much data is required for a reconfigurable element, and on the runtime management of configuration information.

A compromise between partial and full reconfiguration is achieved by using a static partitioning for the reconfigurable elements. This approach is taken e.g. by the Xilinx EAPR Flow, see Section 2.3.3. At design time it is determined which reconfigurable elements will be loaded at runtime for each reconfiguration. All other reconfigurable elements remain static. Therefore, the information which elements are reconfigurable is the same for all configurations. In the case of our example, the reconfigurable elements changed between any configuration are $k = 1, 2, 4, 5, 7, 10$. Hence, at runtime we need to store the configuration data for six reconfigurable elements for each configuration.

Within this section we have discussed the relationship between architecture, design, and configuration. We have shown that the reconfigurable tasks itself provide very little information on the expected reconfiguration effort, because the configuration of the device depends on the mapping of each design. We further introduced a detailed, abstract representation of configuration data and highlighted the different reconfiguration effort for partial and full reconfiguration. In the following we will introduce a generic model to describe runtime reconfiguration. The model is applied at two different levels of design representation: at device configuration level that consists of binary configuration data and at structural level that describes the original design prior device mapping. We further discuss various methods to minimize device reconfiguration by a proper mapping of the tasks.

3.2 Reconfiguration State Graph

We have already shown that the amount of necessary reconfiguration depends on the differences between device configurations, i.e. on the element-wise differences between $\mathbf{d}(n_i)$ and $\mathbf{d}(n_j), n_i, n_j \in \mathcal{N}_\mathsf{T}$. In order to model the device reconfiguration depending on these differences, we introduce the Reconfiguration State Graph (RSG) model. The RSG is a digraph $G(\mathcal{N}_\mathsf{T}, \mathcal{E}_\mathsf{T})$. In this model, the active configuration is associated with a task n from a set $\mathcal{N}_\mathsf{T}$ of possible tasks. A possible change at runtime from the active configuration, associated with task $n_i \in \mathcal{N}_\mathsf{T}$ to a new configuration, associated with task $n_j \in \mathcal{N}_\mathsf{T}$ is described by a directed edge $(n_i, n_j) \in \mathcal{E}_\mathsf{T}$. An example RSG is shown in Figure 3.2.

The transition between the RSG states, or the equivalent reconfiguration, is performed by loading new configuration data into the device. There are as many different reconfigurations as edges in an RSG. If the RSG is complete, i.e. there exists an edge between each possible pair of nodes, the number of possible transitions is $|\mathcal{E}_\mathsf{T}| = |\mathcal{N}_\mathsf{T}|(|\mathcal{N}_\mathsf{T}| - 1)$.

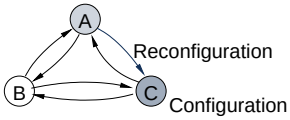


Figure 3.2: A reconfiguration state graph example. The edge from A to C represents the reconfiguration of the device from configuration A to C. The node C symbolizes the new configuration of the device after the reconfiguration is complete.

Each state $n \in \mathcal{N}_\mathsf{T}$ in the RSG is associated with a device configuration $\mathbf{d}(n) = (\mathsf{d}(n)_1, \mathsf{d}(n)_2, \ldots, \mathsf{d}(n)_m)$, that is active in the device's configuration memory when the state n is the active state. Here, we assume that the configuration $\mathbf{d}$ consists of m individually reconfigurable elements, e.g. configuration frames. The vector element $\mathsf{d}(n)_k$, $k = 1 \ldots m$ describes the configuration data for the reconfigurable element k.

Each transition $e = (n_i, n_j) \in \mathcal{E}_\mathsf{T}$ in the RSG requires a change in the active device configuration, i.e. the active configuration changes from $\mathbf{d}(n_i)$ to $\mathbf{d}(n_j)$. This exchange of configuration data causes runtime reconfiguration cost that can be described with the reconfiguration cost $\mathsf{t_e}(e)$. The RSG and the device configuration is illustrated in Example 3.1.

Example 3.1 *Consider the transition from state n_i to n_j with the device configurations $\mathbf{d}(n_i) = (6,5,3,7,3)$ and $\mathbf{d}(n_j) = (6,5,2,7,5)$. In order to perform partial reconfiguration, the configuration of the elements $k = 3$ and $k = 5$ must be changed, causing reconfiguration cost of $\mathsf{t_e}((n_i, n_j)) = 2$ elements.*

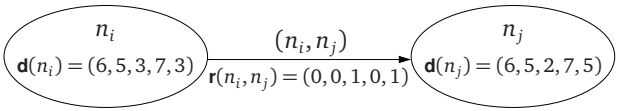


Figure 3.3: Illustration of Example 3.1.

In order to quantify the reconfiguration cost $\mathsf{t_e}$, we define a reconfiguration bitmap $\mathbf{r} : \mathcal{E}_\mathsf{T} \mapsto \{0,1\}^m$, where $\mathbf{r}(e = (n_i, n_j)) = (\mathsf{r}(e)_1, \mathsf{r}(e)_2, \ldots, \mathsf{r}(e)_m)$ describes which of the reconfigurable elements have to be reconfigured in order to realize the transition $e \in \mathcal{E}_\mathsf{T}$. An element $\mathsf{r}(e)_k = 1$ denotes that new configuration data $\mathsf{d}(n_j)_k$ must be loaded into the device's configuration memory at position k. In the RSG model, it is assumed that for a state n the configuration of all m reconfigurable elements is specified. We further assume that each reconfigurable element can be reconfigured independently. Hence, $\mathbf{r}(n_i, n_j)$ can be derived from the difference between the configuration data of both configurations $\mathbf{d}(n_i)$ and $\mathbf{d}(n_j)$:

$$\mathsf{r}((n_i, n_j))_k = \begin{cases} 1 & \text{if } \mathsf{d}(n_i)_k \neq \mathsf{d}(n_j)_k \\ 0 & \text{otherwise} \end{cases} . \tag{3.1}$$

Now, we define the reconfiguration cost $\mathsf{t}_\mathsf{e}(e)$ for a transition e as follows:

$$\mathsf{t}_\mathsf{e}(e) = \sum_{k=1}^{m} \mathsf{w}_\mathsf{t}(k)\mathsf{r}(e)_k, \tag{3.2}$$

where the term $\mathsf{w}_\mathsf{t}(k)$ denotes the reconfiguration cost of element k.

3.2.1 Reconfiguration Time Overhead

Based on the RSG $G(\mathcal{N}_\mathsf{T}, \mathcal{E}_\mathsf{T})$ and the reconfiguration cost $\mathsf{t}_\mathsf{e}(e)$ for a single transition $e \in \mathcal{E}_\mathsf{T}$, we define the total reconfiguration time t.

Definition 3.1 *The* total reconfiguration time t *is the sum of reconfiguration costs over all transitions* $e \in \mathcal{E}_\mathsf{T}$ *in* G*:*

$$t = \sum_{e \in \mathcal{E}_\mathsf{T}} \mathsf{t}_\mathsf{e}(e). \tag{3.3}$$

The total reconfiguration time t describes the reconfiguration cost that occur if all transitions in G are performed once. The measure represents the time overhead at runtime associated with partial reconfiguration. The total reconfiguration time can be normalized in order to describe the average time per reconfiguration:

$$\overline{t} = \frac{t}{|\mathcal{E}_\mathsf{T}|}. \tag{3.4}$$

It provides a reasonable measure to compare different implementations of reconfigurable applications in terms of reconfiguration runtime overhead.

3.2.2 Dynamic Configuration Data Overhead

The time required to reconfigure a device is only one aspect of the reconfiguration overhead. Another one is the amount of configuration data that must be available at system runtime. In FPGAs the amount of configuration data can be huge and hence, is usually stored in external memory. Here we introduce a measure that enables us to determine how much data must be available at runtime. We determine the amount of data for the configurations independent of each other. Moreover, we assume that the device is initialized with a configuration $n \in \mathcal{N}_\mathsf{T}$ at system start-up. Together with the initial configuration, all fully static configuration data is loaded. The amount of fully static configuration data is neglected here, because it is necessary in any case.

At first, consider all reconfigurations represented by the edges $e \in \mathcal{E}_{\mathsf{T},n}$ which lead to configuration n were $\mathcal{E}_{\mathsf{T},n} := \{e \in \mathcal{E}_\mathsf{T} : e = (n', n), n' \in \mathcal{N}_\mathsf{T}\}$. At runtime, any configuration data that changes on any transition $e \in \mathcal{E}_{\mathsf{T},n}$ must be available.

Applied to the configuration n it means that for any reconfigurable element k which is configured with new data on any transition $e \in \mathcal{E}_{T,n}$, the configuration data $\mathsf{d}(n)_k$ are required. Overall, the element k in the reconfiguration bitmap $\mathbf{r}'(n)$ indicates what configuration data $\mathsf{d}(n)_k$ must be present at runtime for a configuration n. The reconfiguration bitmap $\mathbf{r}'(n)$ is now defined as:

$$\mathsf{r}'(n)_k = \bigvee_{e \in \mathcal{E}_{T,n}} \mathsf{r}(e)_k. \tag{3.5}$$

Subsequently, the configuration size $\mathsf{s}_\mathsf{n}(n)$ for a configuration n is defined as:

$$\mathsf{s}_\mathsf{n}(n) = \sum_{k=1}^{m} \mathsf{w}_\mathsf{s}(k)\,\mathsf{r}'(n)_k, \tag{3.6}$$

where $\mathsf{w}_\mathsf{s}(k)$ denotes the configuration size of the element k. The computation of the configuration size $\mathsf{s}_\mathsf{n}(n)$ is illustrated next:

Example 3.2 *Consider an RSG with the reconfiguration bitmaps* $\boldsymbol{r}((n_1, n_3)) = (0, 0, 1, 1, 0)$ *and* $\boldsymbol{r}((n_2, n_3)) = (0, 1, 1, 0, 0)$*. According to Equation 3.5, the reconfiguration bitmap evaluates to* $\boldsymbol{r}'(n_3) = (0, 1, 1, 1, 0)$ *and hence with* $\mathsf{w}_\mathsf{s} = 1$*, the configuration size yields* $\mathsf{s}_\mathsf{n}(n_3) = 3$*.*

Now, we are able to define the total configuration size s:

Definition 3.2 *The* total configuration size *s is the sum of the configuration sizes for all configurations $n \in \mathcal{N}_\mathsf{T}$ that are required at runtime for dynamic reconfiguration:*

$$s = \sum_{n \in \mathcal{N}_\mathsf{T}} \mathsf{s}_\mathsf{n}(n). \tag{3.7}$$

Similar to the reconfiguration time, the total configuration size can also be normalized in order to describe the average amount of data required to store the *dynamic part* of a configuration:

$$\bar{s} = \frac{s}{|\mathcal{N}_\mathsf{T}|}. \tag{3.8}$$

The reconfiguration bitmap $\mathbf{r}'(n)$ determines which elements k of a configuration are dynamic at runtime. Moreover, if the RSG is a complete graph, then the reconfiguration bitmaps $\mathbf{r}'(n)$ are equal for all $n \in \mathcal{N}_\mathsf{T}$: The reconfiguration bitmap $\mathsf{r}'(n)_k$ can only be 0 if all reconfiguration bitmaps $\mathsf{r}((n', n))_k$ are 0, too. This is only true if the configuration data $\mathsf{d}(n')_k = \mathsf{d}(n)_k$ for all $n', n \in \mathcal{N}_\mathsf{T}$. Otherwise, if $\mathsf{d}(n')_k \neq \mathsf{d}(n'')_k$ then at least $\mathsf{r}((n', n))_k = 1$ or $\mathsf{r}((n'', n))_k = 1$ and hence, in any case $\mathsf{r}'(n)_k = 1$.

Although external memory in reconfigurable systems is inexpensive today, the amount of storage needed for configuration data impacts the external memory size, the system's memory bandwidth, the system's energy consumption and the size of

on-chip caches for configuration data. Therefore, the total configuration size must be reduced to allow for more efficient reconfigurable systems.

In the following we will use the RSG model to compute reconfiguration cost for both, binary configuration data and structural design representations.

3.3 Configuration Cost at Bitstream Level

At bitstream level, the data that constitutes the device configuration is well-known for each task. Initially, each task is associated with an individual bitstream produced by the FPGAs implementation tools. From that starting point, the RSG model can be established and subsequently the expected costs for reconfiguration time and configuration data can be computed as discussed in Section 3.2.

For a more detailed illustration consider the Xilinx EAPR flow in Section 2.3.3: In an initial floorplanning step, the designer selects a fixed, reconfigurable area on the device where the reconfigurable tasks will be implemented on. After the final place and route of the reconfigurable modules, a bitstream for each module is generated. Each bitstream contains the data that configures the reconfigurable area such that it realizes the desired functionality.

By selecting the reconfigurable area, it is already defined which configuration data is contained in the bitstream. Let the reconfigurable area to be configured by the configuration frames starting at address a_1 and ending at address a_m. Consistently with the RSG model, the bitstream of task n contains the configuration data $\mathbf{d}(n) = (\mathsf{d}(n)_1, \ldots, \mathsf{d}(n)_m)$, where each element $\mathsf{d}(n)_k, k = 1 \ldots m$ comprises the actual data of a configuration frame associated with the configuration $n \in \mathcal{N}_\mathsf{T}$. Hence, the element $\mathsf{d}(n)_1$ contains the configuration data that is written to address a_1, $\mathsf{d}(n)_2$ the frame data for address a_2 and so forth. The elements $\mathsf{d}(n)_k$ contain a sequence of binary data that has the size of the configuration frame of a particular device. For such a sequence it can be easily decided whether the elements $\mathsf{d}(n_i)_k$ and $\mathsf{d}(n_j)_k$ contain equal configuration data or not, as required by Equation 3.1.

Example 3.3 *Consider a reconfigurable area that covers two columns of CLB logic in a VirtexII-6000 device. The size of a each frame in this device is* 246×32 *bit. The bitstreams of each configuration contain* $m = 44$ *such frames, associated with the frame address* $a_1 \ldots a_{44}$*. As an example, there may be four tasks* $\mathcal{N}_\mathsf{T} = \{1, 2, 3, 4\}$ *and the reconfiguration bitmap is given as follows:*

$$\begin{aligned} \boldsymbol{r}((1,2)) = \boldsymbol{r}((2,1)) = (&0,1,0,0,0,0,1,0,0,0,0,0,1,1,0,0,0,1,1,0,0,0,\\ &0,0,0,0,0,1,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0) \end{aligned} \tag{3.9}$$

$$\begin{aligned} \boldsymbol{r}((2,3)) = \boldsymbol{r}((3,2)) = (&1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,1,0,0,0,0,\\ &0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0) \end{aligned} \tag{3.10}$$

$$\begin{aligned}\boldsymbol{r}((3,4)) = \boldsymbol{r}((4,3)) = (&1,1,0,0,0,0,1,1,1,0,0,0,0,1,0,0,0,1,1,0,0,0,\\ &0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1,0,0,0,0,0)\end{aligned} \tag{3.11}$$

$$\begin{aligned}\boldsymbol{r}((1,3)) = \boldsymbol{r}((3,1)) = (&1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,1,1,0,0,0,\\ &0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0,0)\end{aligned} \tag{3.12}$$

$$\begin{aligned}\boldsymbol{r}((1,4)) = \boldsymbol{r}((4,1)) = (&0,1,0,0,0,0,1,1,1,0,0,0,1,0,0,0,0,1,1,0,0,0,\\ &0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0)\end{aligned} \tag{3.13}$$

$$\begin{aligned}\boldsymbol{r}((2,4)) = \boldsymbol{r}((4,2)) = (&0,1,0,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,\\ &0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0).\end{aligned} \tag{3.14}$$

The average reconfiguration time can be evaluated as, (with $\mathsf{w}_\mathsf{t}(k) = 1\,\mathrm{ms}$*):*

$$\bar{t} = \frac{1}{12}(12+12+10+10+11+11+12+12+12+12+11+11)\,\mathrm{ms} = 11.33\,\mathrm{ms} \tag{3.15}$$

whereas the average reconfiguration time using a bitstream with all 44 frames would be 44 ms per reconfiguration.

The dynamic configuration data overhead can be derived from the reconfiguration bitmap $\boldsymbol{r}((\cdot,n)), n \in \mathcal{N}_\mathsf{T}$*:*

$$\begin{aligned}\boldsymbol{r}((\cdot,n)) = (&1,1,0,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,\\ &0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0).\end{aligned} \tag{3.16}$$

The given frame size is $\mathsf{w}_\mathsf{s}(k) = 7872\,\mathrm{bit}$*. Thus the average amount of dynamic configuration data becomes:*

$$\bar{s} = \frac{1}{4}(15+15+15+15) \times 7872\,\mathrm{bit} = 118080\,\mathrm{bit}. \tag{3.17}$$

This amount of data is only about one third of the original bitstream data that would be required for all 44 frames.

At this point the information about the reconfiguration costs seems to be contradictory. We want to clarify that there is a difference between the configuration data that must be sent to the device in order to perform a reconfiguration between two states—and the configuration data that must be available at runtime in order to be used by any reconfiguration in the RSG. In many existing reconfigurable computing platforms, the bitstreams used for partial reconfiguration must be available at system start-up and can not be compiled at runtime from raw configuration frame data. In [77] we describe several optimization problems which enable us to find a compact set of pre-compiled bitstreams that achieve low average reconfiguration times.

3.4 Configuration Cost at Structural Level

So far we have explained in greater detail, how the RSG model is applied to configuration data that is available in the configuration bitstreams. The approach is relatively straightforward and the results can be directly measured in approximate configuration time and data. But we have already shown in the motivational example in Section 3.1 that the binary configuration data is only one possible instance of a complicated mapping process. The mapping assigns the elements of a structural circuit representation to resource instances in the FPGA. Often there exists a multitude of mapping variants for a structural circuit. The binary configuration data is eventually a transcript of *one* such mapping.

With a focus on reconfiguration cost, we are interested in mappings that exhibit the smallest possible differences in terms of configuration data at binary level. Currently, the existing tools perform the circuit mapping to resource instances with constraints that arise from timing and area restrictions only. Thus, our idea requires a significant effort in new mapping models and algorithms. Further on in this chapter, the appropriate modeling concepts, problem formulations and solution methods are introduced.

The mapping of synthesized netlists and the placement and routing of such a circuit can be done only with very large computational efforts. It is therefore unreasonable to perform this process in order to generate many different mapping variants – from which the ones with the lowest configuration cost could be selected. Instead, we apply the RSG model to higher levels of abstraction, to the structural representations of tasks that are used prior mapping, placement and routing. We develop new methods to assess the reconfiguration cost within the structural representations, but without the creation of different mapping variants and the subsequent computation from binary configuration data.

Our method is based on the idea to find similarities within the structural representations of different tasks or circuits. Hence, the comparison is made with the original task/circuit representation without complicated mapping. Our method identifies elements in the structural representation that can be mapped to the same FPGA resources – for both logic and interconnect. For now, assume that the structural representation of a task/circuit is a graph that consists of nodes and edges. We use the following important assumptions for finding similarities that minimize reconfiguration costs after mapping:

1. Two nodes that are mapped to the same resource instance using the same configuration, require no reconfiguration.
2. Two edges that connect the same two resource instances, as a consequence of the node mapping, can be realized such that they require no routing reconfiguration.

Example 3.4 *For an example to illustrate the assumptions above, please refer to Figure 3.1. The nodes 1 and 2 of task A are mapped to the resources f_1 and f_2, respectively as well as the nodes 5 and 6 of task B. In this scenario, the configurations of the resources f_1 and f_2 differ between both tasks and reconfiguration must be performed. In the contrary, the edge between the nodes 1 and 2 and the edge between the nodes 5 and 6 result in the same interconnection configuration ($\mathsf{d}(n_1)_6 = \mathsf{d}(n_2)_6 = s_1$), due to the chosen mapping of the nodes.*

In the following we refine the model of the structural representation of tasks and derive the configuration cost definition for the structural level. In Section 3.5 we present several methods on how to identify structural similarities such that minimal reconfiguration cost are achieved.

3.4.1 Definitions

Here, we introduce a labelled multidigraph in order to have a formal description for tasks. The notation will be used to describe structural representations of tasks. Using the labelled multidigraph we are able to define structural similarities between two tasks more formally. The notion of structural similarity is used for the optimization methods in Section 3.5 and throughout the Chapters 4 and 5.

Definition 3.3 *A* labelled multidigraph *(LMG) is a graph defined as a 9-tuple $G(\mathcal{N}, \mathcal{E}, \mathcal{S}_\mathsf{n}, \mathcal{S}_\mathsf{p}, \mathsf{l}_\mathsf{n}, \mathsf{s}, \mathsf{d}, \mathsf{l}_\mathsf{s}, \mathsf{l}_\mathsf{d})$ where:*

- $\mathcal{N}$ *is the set of nodes,*
- $\mathcal{E}$ *is the set of directed edges,*
- $\mathcal{S}_\mathsf{n}$ *is the finite set of configurations of the nodes,*
- $\mathcal{S}_\mathsf{p}$ *is the finite set of labels for the source and drain of an edge,*
- $\mathsf{l}_\mathsf{n} : \mathcal{N} \mapsto \mathcal{S}_\mathsf{n}$ *assigns to each node a configuration,*
- $\mathsf{s} : \mathcal{E} \mapsto \mathcal{N}$ *assigns to each edge a source node,*
- $\mathsf{d} : \mathcal{E} \mapsto \mathcal{N}$ *assigns to each edge a drain node,*
- $\mathsf{l}_\mathsf{s} : \mathcal{E} \mapsto \mathcal{S}_\mathsf{p}$ *assigns to each edge a source label, and*
- $\mathsf{l}_\mathsf{d} : \mathcal{E} \mapsto \mathcal{S}_\mathsf{p}$ *assigns to each edge a drain label.*

We assume that each node is labelled with a configuration, provided by the function l_n. In addition to a general directed graph, an LMG allows multiple edges between the same source and drain node. An edge $e \in \mathcal{E}$ of the LMG can be described as $e := (\mathsf{s}(e), \mathsf{d}(e), \mathsf{l}_\mathsf{s}(e), \mathsf{l}_\mathsf{d}(e))$. The functions l_s and l_d are called port labelling functions. The formal definition of the edges is illustrated in Figure 3.4.

The formal definition of an LMG is illustrated now on two examples that constitute possible structural representations of tasks, a netlist of a digital circuit and a dataflow graph.

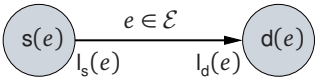


Figure 3.4: Illustration of an edge with $(\mathsf{s}(e), \mathsf{d}(e), \mathsf{l}_\mathsf{s}(e), \mathsf{l}_\mathsf{d}(e))$.

Example 3.5 *In Figure 3.5(a) a schematic of a netlist is shown. The netlist is described as an LMG (Figure 3.5(b)) as follows:*

- $\mathcal{N} = \{\mathit{In}, \mathit{Clk}, \mathit{Out}, \mathit{Lut2}, \mathit{Reg}\}$,
- $\mathcal{E} = \{e_1, \ldots, e_5\}$ *with* $e_1 := (\mathit{In}, \mathit{Lut2}, o, i0)$, $e_2 := (\mathit{In}, \mathit{Reg}, o, d)$, $e_3 := (\mathit{Clk}, \mathit{Reg}, o, \mathit{clk})$, $e_4 := (\mathit{Reg}, \mathit{Lut2}, q, i1)$, $e_5 := (\mathit{Lut2}, \mathit{Out}, o, i)$,
- $\mathcal{S}_\mathsf{n} = \{\varnothing, 0, 1, i0\,i1, \overline{i0}\,i1, i0\,\overline{i1}, \overline{i0\,i1}\}$ *is the set of node configurations; the exact meaning depends on the node where they are used,*
- $\mathcal{S}_\mathsf{p} = \{o, i, i0, i1, \mathit{clk}, d, q\}$ *is the set of labels for the source and the drain of an edge.*

The remaining functions of the multidigraph are illustrated on specific elements. E.g. node Lut2 may have the configuration $\mathsf{l}_\mathsf{n}(\mathit{Lut2}) = i0\,\overline{i1}$. *The edge* e_1 *is represented by the 4-tuple* $(\mathit{In}, \mathit{Lut2}, o, i0)$ *which indicates that the source node is* $\mathsf{s}(e_1) = \mathit{In}$ *and the drain node is* $\mathsf{d}(e_1) = \mathit{Lut2}$. *The edge connects the output port* $\mathsf{l}_\mathsf{s}(e_1) = o$ *on the source side to the input port* $\mathsf{l}_\mathsf{d}(e_1) = i0$ *on the drain side of edge* e_1. *Also note that the wire which is connected to the netlist element In is represented as two independent edges* e_1, e_2 *in the LMG. The edge* e_1 *indicates the connection from node In to node Lut2 and the edge* e_2 *indicates the connection from node In to node Reg.*

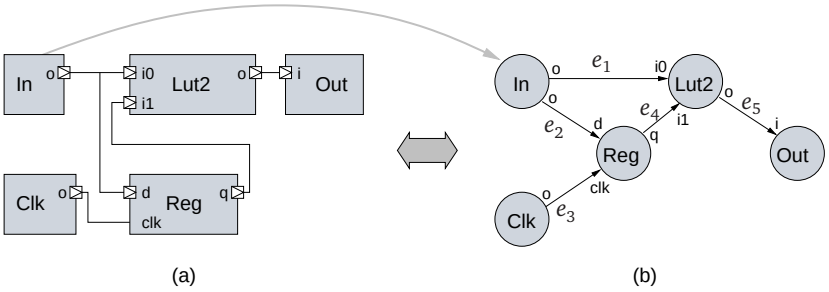


Figure 3.5: (a) netlist of a digital circuit as a schematic. (b) LMG of the netlist.

Example 3.6 *In Figure 3.6 an LMG that describes the dataflow graph of the function d = (a-b)c is shown. Again, the formal description of the LMG is very straightforward:*

- $\mathcal{N} = \{a, b, c, d, -, \times\}$,

- $\mathcal{E} = \{e_1, \ldots, e_5\}$ *with* $e_1 := (a, -, o, i0)$, $e_2 := (b, -, o, i1)$, $e_3 := (-, \times, o, i0)$, $e_4 := (\times, d, o, i)$, *and* $e_5 := (c, \times, o, i1)$,
- $\mathcal{S}_n = \emptyset$, *hence for dataflow graphs the node configuration is not used,*
- $\mathcal{S}_p = \{o, i, i0, i1\}$ *is the set of labels for the source and the drain of an edge.*

In the LMG, both the variables and the operations are represented by nodes. The use of an input variable such as a,b,c *or of an result from an operation such as* $-, \times$ *is indicated by an edge. The example shows that the port labelling functions* l_s, l_d *can be used to indicate the argument where the data is used. The variable* a *is used as an input for the first argument (i0) and the variable* b *as an input for the second argument (i1) of the node* $(-)$. *Thus the expression (*a-b*) is built.*

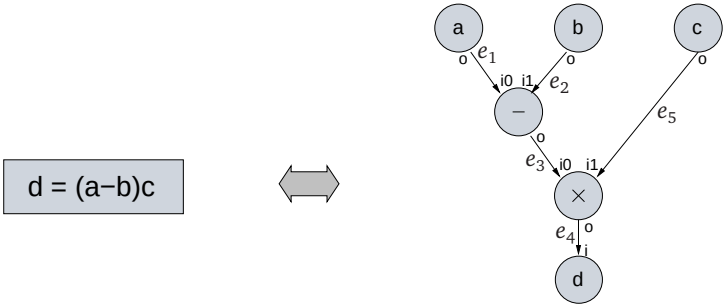


Figure 3.6: An LMG which describes the dataflow graph for the function d = (a-b)c.

We have already observed that the reconfiguration cost depend on the mapping of the structural representation of tasks to common resources and to common interconnect. Now we introduce a formalism to describe such a mapping in detail.

Given the set $\mathcal{N}_T$ of original tasks. Each task can be represented by an LMG as described above. Now these LMGs are called *input graphs* $G(\mathcal{N}, \mathcal{E}, \mathcal{S}_n, \mathcal{S}_p, l_n, s, d, l_s, l_d)$. In the mapping process each input graph is mapped by a transformation to another labelled multidigraph, hence $G \mapsto G'(\mathcal{N}', \mathcal{E}', \mathcal{S}'_n, \mathcal{S}'_p, l'_n, s', d', l'_s, l'_d)$. The graph G' is called the *image graph* of G. The transformation is enabled by three allocation functions a, a_s, and a_d as follows:

- The resource allocation $a : \mathcal{N} \mapsto \mathcal{N}'$ maps each node in G to a new node in G'.
- The two port re-labelling functions $a_s, a_d : \mathcal{E} \mapsto \mathcal{S}'_p$ assign to each edge $e \in \mathcal{E}$ port labels in the image graph G'. Whereas $a_s(e)$ assigns the label of the edge source and $a_d(e)$ assigns the label of the edge drain, respectively. Port re-labelling assists in the mapping formulation and can enhance the similarity of the interconnect.

In a nutshell, the edge $e := (\mathsf{s}(e), \mathsf{d}(e), \mathsf{l}_\mathsf{s}(e), \mathsf{l}_\mathsf{d}(e)) \in \mathcal{E}$ of the input graph G is mapped to the edge $e' := (\mathsf{s}'(e'), \mathsf{d}'(e'), \mathsf{l}'_\mathsf{s}(e'), \mathsf{l}'_\mathsf{d}(e')) \in \mathcal{E}'$ of the image graph G' by the allocation functions a, a_s, and a_d as follows:

- $\mathsf{s}'(e') = \mathsf{a}(\mathsf{s}(e))$,
- $\mathsf{d}'(e') = \mathsf{a}(\mathsf{d}(e))$,
- $\mathsf{l}'_\mathsf{s}(e') = \mathsf{a}_\mathsf{s}(e)$,
- $\mathsf{l}'_\mathsf{d}(e') = \mathsf{a}_\mathsf{d}(e)$.

In the following this edge mapping is abbreviated with the edge allocation $\mathsf{a}_\mathsf{e} : \mathcal{E} \mapsto \mathcal{E}'$.

If the input graph is a netlist, we consider the allocation as a mapping of netlist elements to device resources. In this context we assume that the allocation maps only netlist elements to resources that provide the required functionality. Besides, each node $n' \in \mathcal{N}'$ of an image graph G' is allocated exclusively by a node $n \in \mathcal{N}$ of the input graph G because a netlist describes a digital circuit with concurrently active elements. The port re-labelling allows us to exploit different port-mappings that are available for some resources. For example the input ports of LUTs can be swapped (concurrently with a LUT content modification) while the circuit functionality is retained.

However, if the input graph is a dataflow graph, then two nodes of the same input graph G can possibly be mapped to the same node in image graph G'. For a dataflow graph it is known at design time, when the function associated with a node is executed. Therefore, two nodes that are not executed at the same time interval can share a resource and hence, can be allocated to the same node in the image graph. Just like netlist nodes, port re-labelling can be applied to exploit several different mappings for commutative functions.

Now we consider the mapping of multiple, reconfigurable tasks. Each task $i, j \in \mathcal{N}_\mathsf{T}$ is represented by a separate graph $G_i(\mathcal{N}_i, \mathcal{E}_i, \mathcal{S}_\mathsf{n}, \mathcal{S}_\mathsf{p}, \mathsf{l}_{\mathsf{n},i}, \mathsf{s}_i, \mathsf{d}_i, \mathsf{l}_{\mathsf{s},i}, \mathsf{l}_{\mathsf{d},i})$ and $G_j(\mathcal{N}_j, \mathcal{E}_j, \mathcal{S}_\mathsf{n}, \mathcal{S}_\mathsf{p}, \mathsf{l}_{\mathsf{n},j}, \mathsf{s}_j, \mathsf{d}_j, \mathsf{l}_{\mathsf{s},j}, \mathsf{l}_{\mathsf{d},j})$. The graphs G_i and G_j are completely unrelated, i.e. $\mathcal{N}_i \cap \mathcal{N}_j = \varnothing$ and $\mathcal{E}_i \cap \mathcal{E}_j = \varnothing$ for $i \neq j$. However, the image graphs G'_i, G'_j may have a number of common elements, i.e. if $G_i \mapsto G'_i$ and $G_j \mapsto G'_j$ then both – the sets $\mathcal{N}'_i$ and $\mathcal{N}'_j$ of resulting nodes and the sets $\mathcal{E}'_i$ and $\mathcal{E}'_j$ may have common elements, i.e. $\mathcal{N}'_i \cap \mathcal{N}'_j \neq \varnothing$ and $\mathcal{E}'_i \cap \mathcal{E}'_j \neq \varnothing$. This situation is illustrated in Figure 3.7.

With that concept in mind it is straightforward to define the graph similarity in terms of individual elements more formally. For this purpose we introduce a matching between nodes and between edges of input graphs. Note that this matching is not equivalent to the matching in graph theory.

Definition 3.4 (Matching nodes) *Given the set of nodes $\mathcal{N}_i$ and $\mathcal{N}_j$ of the input graphs G_i and G_j, a resource allocation function* a *and hence the set of nodes $\mathcal{N}'_i$ and $\mathcal{N}'_j$ of the image graphs G'_i and G'_j.*

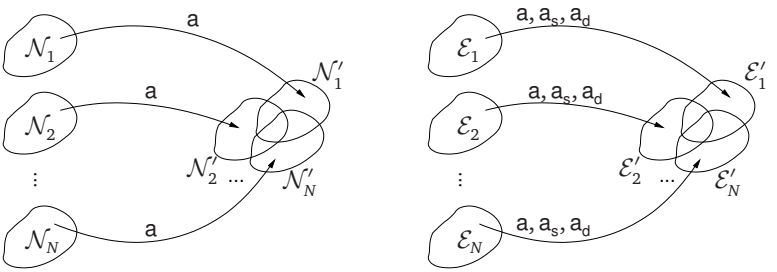


Figure 3.7: The allocation functions a, a_s, and a_d map the nodes and edges of the input graphs $G_1, G_2, \ldots, G_N$ to a image graphs $G'_1, G'_2, \ldots, G'_N$ where the set of nodes and the set of edges may have a common subset.

Two nodes $n_i \in \mathcal{N}_i, n_j \in \mathcal{N}_j$ match with respect to an allocation function a*, if both are mapped to node n' with $n' \in \mathcal{N}'_i, n' \in \mathcal{N}'_j$, i.e.*

$$a(n_i) = a(n_j) = n', n' \in \mathcal{N}'_i, n' \in \mathcal{N}'_j. \tag{3.18}$$

The nodes n_i and n_j are called matching nodes.

Definition 3.5 (Matching edges) *Given the set of edges $\mathcal{E}_i$ and $\mathcal{E}_j$ of the input graphs G_i and G_j, the resource allocation functions* a*,* a_s*, and* a_d *and hence the set of nodes $\mathcal{E}'_i$ and $\mathcal{E}'_j$ of the image graphs G'_i and G'_j. Two edges $e_i \in \mathcal{E}_i$ and $e_j \in \mathcal{E}_j$ match with respect to the allocation functions* a*,* a_s*, and* a_d*, if both are mapped to the edge $e' = (s'(e'), d'(e'), l_s'(e'), l_d'(e'))$ with $e' \in \mathcal{E}'_i, e' \in \mathcal{E}'_j$, i.e.*

- $a(s_i(e_i)) = a(s_j(e_j)) = s'(e')$,
- $a(d_i(e_i)) = a(d_j(e_j)) = d'(e')$,
- $a_s(e_i) = a_s(e_j) = l_s'(e')$,
- $a_d(e_i) = a_d(e_j) = l_d'(e')$.

The edges e_i and e_j are called matching edges.

The definition of matching nodes and matching edges are illustrated by the following example:

Example 3.7 *Figure 3.8 shows an example set of three input graphs G_1, G_2, G_3 and the image graphs G'_1, G'_2, G'_3. The allocation functions* a*,* a_e *are given in Table 3.1. E.g. the nodes 1, 5, and 10 match because they are mapped to the same resource f_1, i.e.* $a(1) = a(5) = a(10) = f_1$*. The node allocation defines also an edge matching. For instance the three edges $(1, 2, o, i_2)$, $(5, 6, o, i_2)$, and $(10, 8, o, i_2)$ are mapped to the same edge (f_1, f_2, o, i_2) and are therefore matching edges.*

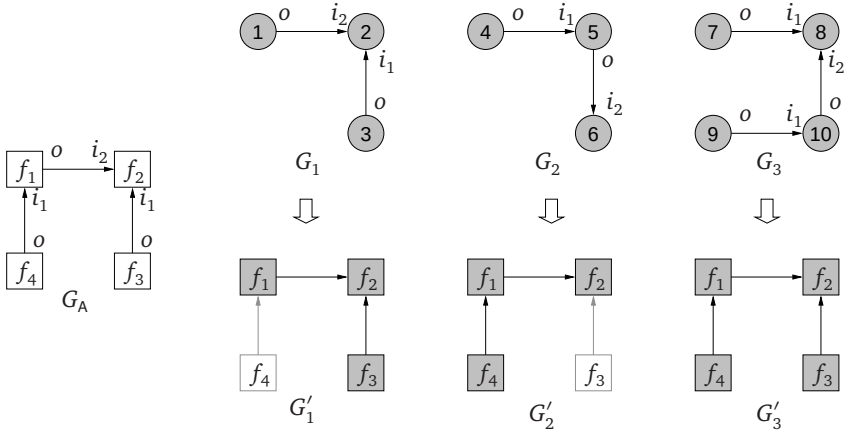


Figure 3.8: Input graphs G_1, G_2, G_3 for Example 3.7. The edges are depicted with with the port labels of the source and drain nodes. The input graphs are mapped to the image graphs G'_1, G'_2, G'_3. The VA G_A provides a common reference for the image graphs.

3.4.2 Virtual Architecture

With the definition of the allocation functions a, a_s, and a_d at hand, we introduce a central concept of our work, the *virtual architecture* (VA). As already mentioned, we need a common reference model in order to assess the reconfiguration cost metrics in the RSG at structural level. This common reference is provided by the VA. We implicitly assumed that the allocation functions map the input graphs to image graphs that are related to each other, i.e. the nodes and edges of the image graphs may have common subsets. In order to built a common reference model for the image graphs, we define the VA as a supergraph that contains all image graphs as a subset. The

Table 3.1: Allocation of nodes and edges as used in Example 3.7. The input graphs are depicted in Figure 3.8.

G_i	G_1	G_2	G_3	$a(n)$ $a_e(e)$	u()
Nodes $n \in \mathcal{N}_i$	1	5	10	f_1	3
	2	6	8	f_2	3
	3		7	f_3	2
		4	9	f_4	2
Edges $e \in \mathcal{E}_i$	$(1, 2, o, i_2)$	$(5, 6, o, i_2)$	$(10, 8, o, i_2)$	(f_1, f_2, o, i_2)	3
$(s_i(e), d_i(e),$	$(3, 2, o, i_1)$		$(7, 8, o, i_1)$	(f_3, f_2, o, i_1)	2
$l_{s,i}(e), l_{d,i}(e))$		$(4, 5, o, i_1)$	$(9, 10, o, i_1)$	(f_4, f_1, o, i_1)	2

VA itself is defined as an LMG denoted as $G_\mathsf{A} = (\mathcal{N}_\mathsf{A}, \mathcal{E}_\mathsf{A}, \mathcal{S}_{\mathsf{A,n}}, \mathcal{S}_{\mathsf{A,p}}, \mathsf{l}_{\mathsf{A,n}}, \mathsf{s}_\mathsf{A}, \mathsf{d}_\mathsf{A}, \mathsf{l}_{\mathsf{A,s}}, \mathsf{l}_{\mathsf{A,d}})$. We require the set $\mathcal{N}_\mathsf{A}$ of nodes and the set $\mathcal{E}_\mathsf{A}$ of edges to be enumerable sets, i.e. the elements of these sets can be completely enumerated by an index l as follows: $n_l \in \mathcal{N}_\mathsf{A} = \{n_1, \ldots, n_{|\mathcal{N}_\mathsf{A}|}\}, l = 1, \ldots, |\mathcal{N}_\mathsf{A}|$ and $e_l \in \mathcal{E}_\mathsf{A} = \{e_1, \ldots, e_{|\mathcal{E}_\mathsf{A}|}\}, l = 1, \ldots, |\mathcal{E}_\mathsf{A}|$. For a set of input graphs G_i with $i \in \mathcal{N}_\mathsf{T}$ and the allocation functions a, a_s, and a_d, the VA observes the following conditions regarding the image of the input graphs:

$$\bigcup_{i \in \mathcal{N}_\mathsf{T}} \mathcal{N}_i' \subset \mathcal{N}_\mathsf{A}, \tag{3.19}$$

$$\bigcup_{i \in \mathcal{N}_\mathsf{T}} \mathcal{E}_i' \subset \mathcal{E}_\mathsf{A}. \tag{3.20}$$

There exists a very straightforward way to construct a VA for a set of image graphs. For a known allocation function, the VA can be constructed from the image graphs by setting up the nodes and edges according to:

$$\mathcal{N}_\mathsf{A} := \bigcup_{i \in \mathcal{N}_\mathsf{T}} \mathcal{N}_i', \tag{3.21}$$

$$\mathcal{E}_\mathsf{A} := \bigcup_{i \in \mathcal{N}_\mathsf{T}} \mathcal{E}_i'. \tag{3.22}$$

Alternatively, the VA may be given along with the input graphs. In this case, the nodes and edges in the VA may be available resources that can be allocated by the nodes and edges of the image graphs. In this situation, the problem consists of finding valid allocation functions such that the conditions in the Equations 3.19 and 3.20 hold.

The VA provides the ideal tool to establish the reconfiguration cost model based on the RSG. As the name *virtual architecture* suggests, the VA defines an assumed architecture model. The reconfiguration cost can now be established with the VA as a reference.

The VA defines the reconfigurable elements of a device configuration $\mathbf{d}(i) = (\mathsf{d}(i)_1, \ldots, \mathsf{d}(i)_m), i \in \mathcal{N}_\mathsf{T}$. The device configuration $\mathbf{d}(i)$ consists of $m = |\mathcal{N}_\mathsf{A}| + |\mathcal{E}_\mathsf{A}|$ elements. We assume all elements of a VA are reconfigurable. For the VA model, the elements $\mathsf{d}(i)_k$ with $k = 1, \ldots, m$ describe the configuration as follows: the elements $\mathsf{d}(i)_1, \ldots, \mathsf{d}(i)_{|\mathcal{N}_\mathsf{A}|}$ are associated with the resource configuration of the nodes $n_1, \ldots, n_{|\mathcal{N}_\mathsf{A}|} \in \mathcal{N}_\mathsf{A}$ and the elements $\mathsf{d}(i)_k, k = |\mathcal{N}_\mathsf{A}| + 1, \ldots, m$ are associated with the interconnect use of the edges $e_1, \ldots, e_{|\mathcal{E}_\mathsf{A}|} \in \mathcal{E}_\mathsf{A}$.

In the following we describe the device configuration $\mathbf{d}(i)$ for an input graph G_i with $i \in \mathcal{N}_\mathsf{T}$, an allocation a and a VA G_A. The device configuration describes how the resources and interconnect of a VA must be configured in order to implement the image graph G_i' on the VA such that it realizes the functionality described in the input graph G_i.

The configuration of a resource $n_k \in \mathcal{N}_A$ for an input graph G_i depends on the configuration required by the node $n_i \in \mathcal{N}_i$ that is allocated to the resource n_k. More precisely, the configuration of resource $n_k \in \mathcal{N}_A, k \in \{1,\ldots,|\mathcal{N}_A|\}$ is specified by $\mathsf{d}(i)_k = \mathsf{l}'_{n,i}(n_k)$ if a node $n_i \in \mathcal{N}_i$ exists with $\mathsf{a}(n_i) = n_k \in \mathcal{N}'_i, n_k \in \mathcal{N}_A$, otherwise $\mathsf{d}(i)_k = \varnothing$. The nodes n_k in the VA may have a different configuration for each task $i \in \mathcal{N}_T$.

The edges $e_{k-|\mathcal{N}_A|} \in \mathcal{E}_A, k \in \{|\mathcal{N}_A|+1,\ldots,m\}$ in the VA represent possible connections between nodes. The edges given by an input graph are mapped to edges in an image graph. Depending on this mapping, the interconnect in the VA is either used by the image graph or it is unused. The configuration of interconnect $e_{k-|\mathcal{N}_A|} \in \mathcal{E}_A, k \in \{|\mathcal{N}_A|+1,\ldots,m\}$ is given with $\mathsf{d}(i)_k = 1$, if an edge $e_i \in \mathcal{E}_i$ is mapped to an edge $e_{k-|\mathcal{N}_A|} \in \mathcal{E}_A$ i.e. $\mathsf{a}_e(e_i) = e_{k-|\mathcal{N}_A|} \in \mathcal{E}'_i$, otherwise $\mathsf{d}(i)_k = 0$.

The device configuration that describes the mapping of each input graph to a configuration of the VA model fully defines the reconfiguration cost within the RSG model. The reconfiguration bitmap is computed from the device configuration as given in Equation 3.1. The definitions used so far are summarized in Figure 3.9.

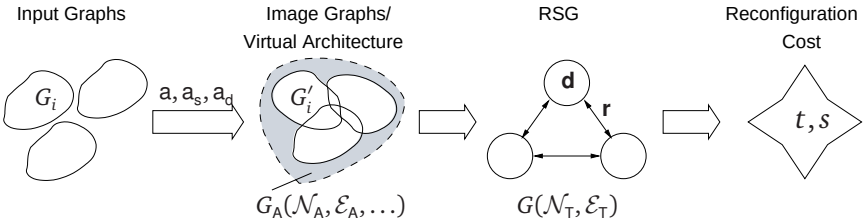


Figure 3.9: The relationship between input graphs, image graphs, the VA, and the RSG cost model. The reconfiguration cost t, s are defined in Section 3.2.

Example 3.8 *In this example we want to illustrate the reconfiguration cost that arise for the tasks G_1, G_2, G_3 given in Table 3.1. The VA graph is given by the nodes $\mathcal{N}_A := \{f_1, f_2, f_3, f_4\}$ and edges $\mathcal{E}_A := \{(f_1, f_2, o, i_2), (f_3, f_2, o, i_1), (f_4, f_1, o, i_1)\}$. The allocation functions a, a_e given in the table yield the device configurations $\boldsymbol{d}(i), i = 1,2,3$ for each task: $\boldsymbol{d}(1) = ('1','2','3',\varnothing,1,1,0)$, $\boldsymbol{d}(2) = ('5','6',\varnothing,'4',1,0,1)$, and $\boldsymbol{d}(3) = ('10','8','7','9',1,1,1)$. Note that these device configurations are not related to a specific mapping to the target architecture as in Section 3.1. The reconfiguration bitmap can be derived from the device configurations $\boldsymbol{d}(i)$. In the example the bitmaps are: $\boldsymbol{r}(1,2) = \boldsymbol{r}(2,1) = (1,1,1,1,0,1,1)$, $\boldsymbol{r}(2,3) = \boldsymbol{r}(3,2) = (1,1,1,1,0,1,0)$, and $\boldsymbol{r}(1,3) = \boldsymbol{r}(3,1) = (1,1,1,1,0,0,1)$.*

The reconfiguration cost (with unit weight for w_t and w_s) in the example evaluate to $\overline{t} = \frac{16}{3}$ and $\overline{s} = 6$.

The VA serves as an internal reference architecture in order to optimize the allocation and to identify similarities between the input graphs. However, the mapping of the input graphs to device resources during the implementation process is a different step. The allocation found for the mapping to a VA provides important allocation constraints for the device mapping: nodes that are mapped to the same node in the VA must be mapped to the same resource in the FPGA and edges that are mapped to the same edge in the VA must be realized by the same interconnect in the FPGA, too. If these constraints are observed, the reconfiguration cost can be reduced as expected from the VA/RSG based cost model.

3.4.3 Reconfiguration Costs in the VA Context

Our reconfiguration cost model describes two cost functions: reconfiguration time t (Equation 3.3) and configuration size s (Equation 3.7). The initial definition of these cost functions is purely based on the RSG model and the associated reconfiguration bitmap. Now we present an alternative way to compute reconfiguration cost for structural models, based on the reuse function. The aims of the new formulation are twofold: (1) it shows the relationship between resource reuse and reconfiguration cost and (2) the cost function can be evaluated more efficiently, if the allocation function and the reuse function is modified iteratively as it is required in many heuristic optimization methods. For our derivation we assume that the RSG is complete and that the allocation a_e describes a one-to-one mapping with $|\mathcal{E}| = |\mathcal{E}'|$. We further set the weight functions to unit weight, i.e. $\mathsf{w}_\mathsf{s} = \mathsf{w}_\mathsf{t} = 1$.

The *reuse function* $\mathsf{u} : \mathcal{N}_\mathsf{A} \cup \mathcal{E}_\mathsf{A} \mapsto \mathbb{N}$ defines how often the nodes and edges in the VA are allocated by the nodes and edges of the input graphs. However, the reuse function does not depend on the configurations of the nodes that are required by the input graphs. Therefore the reconfiguration cost can only be derived from the reuse function if the cost depends purely on the utilization of that resource, e.g. in the case of interconnect or other non-reconfigurable resources. In this section, we limit the computation of the total reconfiguration time to the reconfiguration cost caused by interconnect only. The cost term is denoted as $t_\mathcal{E}$.

First, consider a reconfiguration for two input graphs G_i, G_j with the image graphs G'_i, G'_j and an allocation a. The reconfiguration cost for a transition from task $i \in \mathcal{N}_\mathsf{T}$ to task $j \in \mathcal{N}_\mathsf{T}$ consists of the de-configuration of all edges $\mathcal{E}'_i$ and the configuration of all edges $\mathcal{E}'_j$. Because we reconfigure only the differences between both tasks, the common subset $\mathcal{E}'_i \cap \mathcal{E}'_j$ with respect to the VA is not reconfigured and hence the reconfiguration cost evaluate to:

$$t_\mathcal{E}(i,j) = |\mathcal{E}'_i| + |\mathcal{E}'_j| - 2|\mathcal{E}'_i \cap \mathcal{E}'_j|. \tag{3.23}$$

As the RSG is complete, the total reconfiguration costs are the sum of all possible

transitions between the graphs:

$$t_{\mathcal{E}} = \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} t_{\mathcal{E}}(i,j) = \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} \left[|\mathcal{E}'_i| + |\mathcal{E}'_j| - 2|\mathcal{E}'_i \cap \mathcal{E}'_j| \right] \tag{3.24}$$

The number of edges in all input graphs is constant and because we require a one-to-one mapping by the allocation function we let

$$E = \sum_{i \in \mathcal{N}_\mathsf{T}} |\mathcal{E}_i| = \sum_{i \in \mathcal{N}_\mathsf{T}} |\mathcal{E}'_i| = \sum_{e \in \mathcal{E}_\mathsf{A}} \mathsf{u}(e) = \text{const.} \tag{3.25}$$

be the total number of edges. With this definition, Equation 3.24 can be simplified as follows: at first the double sum is expanded to

$$t_{\mathcal{E}} = \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i| + \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_j| - 2 \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j|. \tag{3.26}$$

In the first term we swap the inner and outer sum and substitute the term according to Equation 3.25. The sum over j repeats $|\mathcal{N}_\mathsf{T}| - 1$ times. In the second term, the double sum over $|\mathcal{E}'_j|$ is equivalent to $\sum_{i \in \mathcal{N}_\mathsf{T}} \left[E - |\mathcal{E}'_i| \right]$. Hence we receive

$$\begin{aligned}
t_{\mathcal{E}} &= (|\mathcal{N}_\mathsf{T}| - 1)E + \sum_{i \in \mathcal{N}_\mathsf{T}} \left[E - |\mathcal{E}'_i| \right] - 2 \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j| && (3.27) \\
t_{\mathcal{E}} &= (|\mathcal{N}_\mathsf{T}| - 1)E + |\mathcal{N}_\mathsf{T}|E - E - 2 \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j| && (3.28) \\
t_{\mathcal{E}} &= 2(|\mathcal{N}_\mathsf{T}| - 1)E - 2 \underbrace{\sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j|}_{x_\mathsf{t}} && (3.29) \\
t_{\mathcal{E}} &= 2(|\mathcal{N}_\mathsf{T}| - 1)E - 2x_\mathsf{t}. && (3.30)
\end{aligned}$$

The term x_t represents the number of edges in the VA which are allocated in both, the configuration i and the configuration j, for all transitions (i, j) in the RSG. It can be computed alternatively—in the case of full reconfigurability—by using the reuse function. The reuse function $\mathsf{u}(e_{k-|\mathcal{N}_\mathsf{A}|})$ for an interconnect element $e_{k-|\mathcal{N}_\mathsf{A}|} \in \mathcal{E}_\mathsf{A}$ describes in how many configurations $\mathbf{d}(i), i \in \mathcal{N}_\mathsf{T}$ the element $e_{k-|\mathcal{N}_\mathsf{A}|}$ is used, i.e. $\mathsf{d}(i)_k = 1$. Instead of computing $|\mathcal{E}'_i \cap \mathcal{E}'_j|$, we investigate the reconfiguration cost for a single interconnect $e_{k-|\mathcal{N}_\mathsf{A}|} \in \mathcal{E}_\mathsf{A}$ in the VA.

The RSG $G(\mathcal{N}_\mathsf{T}, \mathcal{E}_\mathsf{T})$ is a complete graph with the node set $\mathcal{N}_\mathsf{T}$ and hence, any subgraph of G with $\mathcal{N}'_\mathsf{T} \subset \mathcal{N}_\mathsf{T}$ is also complete. The subgraph is given by $\mathcal{N}'_\mathsf{T} = \{i \in \mathcal{N}_\mathsf{T} : \mathsf{d}(i)_k = 1\}$. It follows that $|\mathcal{N}'_\mathsf{T}| = \mathsf{u}(e_{k-|\mathcal{N}_\mathsf{A}|})$. For a complete (sub)graph with

$\mathsf{u}(e_{k-|\mathcal{N}_\mathsf{A}|})$ nodes it is known that $\mathsf{u}(e_{k-|\mathcal{N}_\mathsf{A}|})(\mathsf{u}(e_{k-|\mathcal{N}_\mathsf{A}|})-1)$ edges exist. Hence, the term x_t is now given by:

$$x_\mathsf{t} = \sum_{e \in \mathcal{E}_\mathsf{A}} \mathsf{u}(e)(\mathsf{u}(e)-1). \tag{3.31}$$

Equation 3.30 can be further simplified using the Equations 3.25 and 3.31 as follows:

$$\begin{aligned} t_\mathcal{E} &= 2(|\mathcal{N}_\mathsf{T}|-1)E - 2\sum_{e \in \mathcal{E}_\mathsf{A}} \mathsf{u}(e)(\mathsf{u}(e)-1) && (3.32)\\ &= 2(|\mathcal{N}_\mathsf{T}|-1)E + 2\sum_{e \in \mathcal{E}_\mathsf{A}} \mathsf{u}(e) - 2\sum_{e \in \mathcal{E}_\mathsf{A}} \mathsf{u}^2(e) && (3.33)\\ &= 2|\mathcal{N}_\mathsf{T}|E - 2\sum_{e \in \mathcal{E}_\mathsf{A}} \mathsf{u}^2(e). && (3.34) \end{aligned}$$

It appears that the total reconfiguration time $t_\mathcal{E}$ for the interconnect is minimized by increasing the reuse of interconnect. Moreover, the effect of the reuse function is squared and it follows for a VA mapping that fewer elements which are reused more often are preferred to more elements which are reused less often. Equation 3.34 also indicates that the reconfiguration time t and the configuration size s are not necessarily optimized at the same time. If we assume that the VA interconnect contains only elements which are reconfigured at runtime, i.e. $1 \leq \mathsf{u}(e) < |\mathcal{N}_\mathsf{T}|$, then the configuration size $s_\mathcal{E}$ for interconnect depends only on the amount of interconnect in the VA, i.e. $s_\mathcal{E} = |\mathcal{E}_\mathsf{A}|$. However, the reconfiguration time $t_\mathcal{E}$ depends on the sum of squares of the reuse function, whereas the reuse function does not matter for the configuration size.

3.5 Allocation Functions with Minimal Reconfiguration Costs

In Section 3.4.1 it was shown how the reconfiguration costs can be defined for input graphs. The costs depend on the structure of the input graphs, which are given by design, but the costs also depend on the allocations a, a_s, and a_d. There are many different allocations to choose from, but we are interested in the allocations that reduce reconfiguration costs as much as possible, which constitutes the allocation problem:

Definition 3.6 *Consider the RSG $G(\mathcal{N}_\mathsf{T}, \mathcal{E}_\mathsf{T})$ and the associated input graphs G_i, $i \in \mathcal{N}_\mathsf{T}$. The allocation functions* a, a_s, *and* a_d *map the input graphs to image graphs $G'_i, i \in \mathcal{N}_\mathsf{T}$. The* allocation problem *consists of finding such allocation functions so that the resulting reconfiguration cost s or t become minimal.*

In order to solve the allocation problem, we developed different approaches to compute allocations that achieve minimal costs. The cost assessment is based on the VA as a reference. We assume a complete RSG as before and for this reason the reconfiguration cost can be calculated by using the reuse function.

The nodes and edges of the input graphs induce very different reconfiguration costs. In FPGAs, nodes describe logical resources that require a small, fixed amount of configuration data only. Conversely, edges describe connections between logical resources. The connections are realized by using configurable interconnect resources in several switch boxes, distributed on the FPGA area. The switch box configuration requires much more configuration data than the configuration of logical resources. Hence, reconfiguration of interconnect causes substantially more reconfiguration costs. We will therefore neglect the reconfiguration costs of logic resources for now and focus on methods to minimize reconfiguration costs for interconnect-related configuration data instead. A method for the simultaneous optimization of the allocation functions with respect to reconfiguration cost for both logic and interconnect reconfiguration is discussed in the Sections 3.5.2 and 5.4.3.

In the following we describe a method that finds as many as possible matching edges for a set of input graphs (Section 3.5.1). In Section 3.5.2 we describe a method which directly enumerates different allocation functions and selects the best solution.

3.5.1 Allocation of Node Pairs

We have already shown that the reuse of edges in the VA decreases reconfiguration cost. We want to identify an allocation of edges that maximizes the reuse in order to minimize reconfiguration cost. The edge mapping is a result of the node allocation. Originally it does not seem possible to allocate the edges in the input graphs directly, without node allocations. Here, we derive a problem formulation that allows us to optimize the edge allocation that achieves maximum reuse directly; the node allocation is a mere by-product.

First we describe how the input graphs can be decomposed in order to treat edges independently of the node allocation. For the decomposition we show that the interconnect-related reconfiguration cost can be split into terms that can be accumulated linearly without interference. The optimization problem consists of the selection of edge allocations from a set of possible edge allocations such that the interconnect-related reconfiguration cost are minimal. Meanwhile it must be possible to realize the edge allocation with a valid allocation for nodes.

The input graphs can be decomposed into *node pairs*. A node pair can be any combination of nodes $n_{i,1}, n_{i,2} \in \mathcal{N}_i$ from the same input graph G_i. There may be several edges between the nodes in such a pair. An example is shown in Figure 3.10(a). Each node pair may be allocated to a small VA with two resources and the required

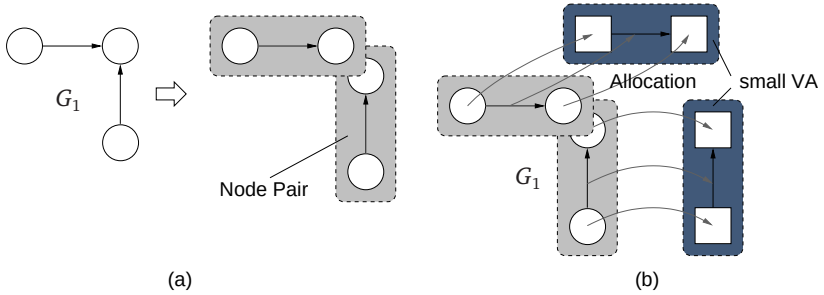


Figure 3.10: (a) The graph G_1 is decomposed into pairs of nodes including the related edges. (b) The allocation of a node pair also allocates the related edge.

interconnect as shown in Figure 3.10(b). Note that the edges running between the same node pair can not be mapped independently.

Consistently with our VA model we can allocate node pairs from different input graphs to the same resources in a VA. A node pair of one input graph may be allocated to the same resources in the VA as the node pair of another input graph, thus the allocation results in matching nodes and matching edges at the same time. The node pairs that are allocated to the VA resources are treated independently of node pairs allocated to other VA resources (Figure 3.10(b)). This is reasonable because the reuse of edges in the VA can be treated independently in this case.

Now we introduce the notion of a *node pair combination* $\mathbf{n} = (\mathbf{n}_1, \ldots, \mathbf{n}_{|\mathcal{N}_\mathsf{T}|})$: A node pair combination is a vector of 2-tuples, i.e. each vector element $\mathbf{n}_i = (n_{i,1}, n_{i,2}), i = 1, \ldots, |\mathcal{N}_\mathsf{T}|$ consists of two nodes $n_{i,1}, n_{i,2}$. The nodes $n_{i,1}$ and $n_{i,2}$ represent a node pair from task $i \in \mathcal{N}_\mathsf{T}$, i.e. $n_{i,1}, n_{i,2} \in \mathcal{N}_i$. We assume the node pairs in a node pair combination are all allocated to common VA nodes as follows: $\forall i \in \mathcal{N}_\mathsf{T} : \mathsf{a}(n_{i,1}) := n_{k_1} \in \mathcal{N}_\mathsf{A}$ and $\mathsf{a}(n_{i,2}) := n_{k_2} \in \mathcal{N}_\mathsf{A}$. For this allocation, the matching edges can be determined accordingly.

In order to find the node pairs that induce matching edges for low reconfiguration cost, any node pair combination over all input graphs is considered. A weight is assigned to each node pair combination in order to formulate an optimization problem. More precisely, the weight reflects the reuse of VA elements as defined by the node pair combination. Here we are only interested in the reuse of interconnect. The example below illustrates the allocation of a node pair combination to a common VA.

Example 3.9 *In Figure 3.11 an example with two input graphs is shown. The node pairs are allocated to a small VAs. Consider the node pair combination* $\mathbf{n} = ((1,2), (4,5))$*, which consists of the node pair (1,2) and the node pair (4,5): the nodes 1,4 are allocated to one resource in VA A, and the nodes 2,5 are allocated to the other*

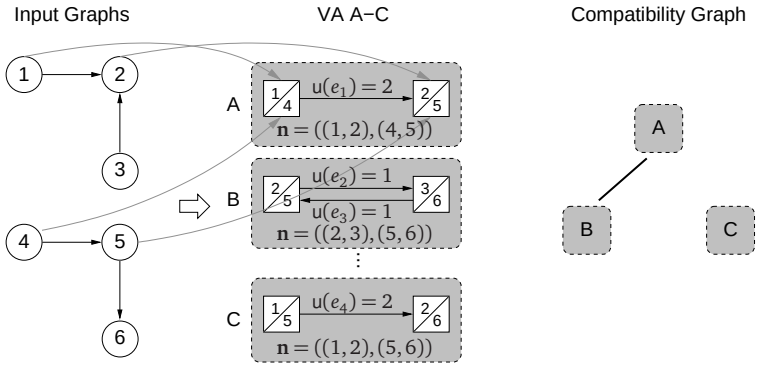


Figure 3.11: Allocation of node pairs from two different input graphs to the VA. The compatibility graph shows which allocations can be realized at the same time.

resource in VA A. The VA A also contains one interconnect to realize the edge described by the original node pairs (1,2) and (4,5). The interconnect is allocated by edges from two input graphs, i.e. $u(e_1) = 2$.

Obviously not all node pair combinations can be allocated as required at the same time, thus some node pair combinations are incompatible to each other. This is illustrated in the example below:

Example 3.10 *Consider the node pair combinations allocated to the VAs A–C and the compatibility graph in Figure 3.11. The allocations chosen in VA A and B are compatible because they are based on a consistent node allocation. In both VAs, the nodes 2,5 are allocated to the same VA resource and the nodes 1,4 and 3,6 are completely independent, i.e. they have no node in common. By contrast, the allocations in the VAs A and C are incompatible because, e.g. the nodes 1,4 are allocated to one resource in VA A and at the same time, the nodes 1,5 are allocated to one resource in VA B. For design netlists it is not possible two allocate the nodes 4,5 of the same input graph to the same node in the VA model. In the compatibility graph in Figure 3.11, all node pair combinations that enforce a compatible allocation are connected by an edge, incompatible node pair combinations are not connected.*

Here we propose that the lowest reconfiguration cost in terms of routing reconfiguration can be achieved as follows: (1) generate a set of all possible node pair combinations and (2) derive a compatibility graph for this set; (3) select the largest subset of compatible node pair combinations. (4) derive an allocation that complies with this subset. The steps 1–3 are described in the following.

Creation of Node Pairs

All possible node pair combinations can be generated by enumeration. The number of node pairs combinations is huge, but can be reduced in practice, cf. Sections 3.5.1 and 3.5.3. The enumeration is very straightforward. For one input graph $G_i, i \in \mathcal{N}_\mathsf{T}$, every possible combination of two nodes is considered as a node pair. Each such node pair is combined with any other node pair from another input graph $G_j, j \in \mathcal{N}_\mathsf{T}$ and $i \neq j$ to form a node pair combination. The enumeration is realized with Algorithm 1, which creates all possible node pair combinations for the of input graphs.

Algorithm 1 Enumerate all node pair combinations

```
1:  cont = false
2:  n = ∅
3:  repeat
4:    i = N_T
5:    while not cont and i > 0 do
6:      if not n_{i,2} last element in N_i then
7:        n_{i,2} = choose next element after n_{i,2} from N_i
8:        cont = true
9:      else if not n_{i,1} last element in N_i then
10:       n_{i,1} = choose next element after n_{i,1} from N_i
11:       n_{i,2} = ∅
12:       cont = true
13:     else
14:       n_{i,1} = ∅
15:       n_{i,2} = ∅
16:       i = i − 1
17:       cont = false
18:     end if
19:     store n
20:   end while
21: until cont = false
```

The algorithm does enumerate all node pair combinations and stores them as vectors $\mathbf{n}$, it does not distinguish between valid, invalid, and symmetric node pair combinations.

A vector $\mathbf{n}$ is an invalid node pair combination, if the node pair of one task $i \in \mathcal{N}_\mathsf{T}$ is allocated to two different resources $n_{k_1} \neq n_{k_2}$ in the VA while the node pair of another task $j \in \mathcal{N}_\mathsf{T}$ is allocated to two identical resources $n_{k_1} = n_{k_2}$. More formally, a node pair combination $\mathbf{n} = (\ldots,(n_{i,1},n_{i,2}),\ldots,(n_{j,1},n_{j,2}),\ldots), i \neq j$ is

invalid if:

$$\begin{aligned} &n_{i,1} \neq n_{i,2}, \text{ and thus } \mathsf{a}(n_{i,1}) = n_{k_1} \neq \mathsf{a}(n_{i,2}) = n_{k_2} \text{ and} \\ &n_{j,1} = n_{j,2}, \text{ and thus } \mathsf{a}(n_{j,1}) = n_{k_1} = \mathsf{a}(n_{j,2}) = n_{k_2}. \end{aligned} \tag{3.35}$$

Algorithm 1 generates all possible node pair combinations for all input graphs. We already observed that edges running between two nodes can not be allocated independently. Hence, for any node pair $n_{i,1}, n_{i,2}$ any existing edge $(n_{i,1}, n_{i,2}) \in \mathcal{E}_i$ and $(n_{i,2}, n_{i,1}) \in \mathcal{E}_i$ would be included in the weight $\mathsf{w}(\mathbf{n})$ for the node pair combination. Nevertheless, the Algorithm 1 produces a vector $\mathbf{n}$ with the reverse order of the node pair $(n_{i,1}, n_{i,2})$, which is fully symmetric and hence the allocation of both vectors is compatible to each other. In consequence, the solution algorithm will include both vectors in the solution, which results in an invalid cost computation and in a much more complex problem formulation. Therefore we include only one vector of the two symmetric vectors. Two vectors $\mathbf{n} = (\ldots, (n_{i,1}, n_{i,2}), \ldots)$ and $\mathbf{n}' = (\ldots, (n'_{i,1}, n'_{i,2}), \ldots)$ are symmetric if:

$$n_{i,1} = n'_{i,2} \text{ and } n_{i,2} = n'_{i,1} \; \forall i \in \mathcal{N}_\mathsf{T}. \tag{3.36}$$

Algorithm 2 determines whether a vector can be discarded because a symmetric vector is produced by Algorithm 1. Algorithm 2 assumes a partial order between the nodes $n_{i,1}, n_{i,2} \in \mathcal{N}_i$.

Algorithm 2 identify dual match pair vectors

```
1: sym = false
2: for all i such that 1 ≤ i ≤ |N_T| do
3:    if n_{i,1} > n_{i,2} then
4:       sym = true
5:    else if n_{i,1} ≠ n_{i,2} then
6:       break;
7:    end if
8: end for
9: return sym
```

We have already shown that the reconfiguration time for interconnect depends on the reuse of interconnect in the VA. If the sum over the interconnect reuse is maximized then the reconfiguration time for interconnect will be minimal, cf. Equation 3.34. The reuse can be determined for each node pair combination independently, as described in Example 3.9. Suppose the nodes that are contained in the vector $\mathbf{n}$ are allocated to a VA. The VA is denoted as $G_\mathsf{A}^{(\mathbf{n})}(\mathcal{N}_\mathsf{A}^{(\mathbf{n})}, \mathcal{E}_\mathsf{A}^{(\mathbf{n})})$. The weight w associated with the vector $\mathbf{n}$ is then given as:

$$\mathsf{w}(\mathbf{n}) = \sum_{e \in \mathcal{E}_\mathsf{A}^{(\mathbf{n})}} \mathsf{u}^2(e). \tag{3.37}$$

Next, we describe which of the generated node pair combinations are compatible with each other.

Compatible Match Pair Vectors

Here we define whether the allocation assumed for two node pair combinations $\mathbf{n} = (\ldots,(n_{i,1},n_{i,2}),\ldots)$ and $\mathbf{n}' = (\ldots,(n'_{i,1},n'_{i,2}),\ldots)$ can be realized at the same time, i.e. whether the two vectors are compatible to each other. An example has been discussed in Example 3.10. The allocation induced by two such vectors can be realized, if the following conditions hold:

$$\forall i \in \mathcal{N}_\mathsf{T} : n_{i,1} = n'_{i,1} \qquad \vee\, \forall i \in \mathcal{N}_\mathsf{T} : n_{i,1} \neq n'_{i,1}, \tag{3.38}$$

$$\forall i \in \mathcal{N}_\mathsf{T} : n_{i,2} = n'_{i,2} \qquad \vee\, \forall i \in \mathcal{N}_\mathsf{T} : n_{i,2} \neq n'_{i,2}, \tag{3.39}$$

$$\forall i \in \mathcal{N}_\mathsf{T} : n_{i,1} = n'_{i,2} \qquad \vee\, \forall i \in \mathcal{N}_\mathsf{T} : n_{i,1} \neq n'_{i,2}, \text{ and} \tag{3.40}$$

$$\forall i \in \mathcal{N}_\mathsf{T} : n_{i,2} = n'_{i,1} \qquad \vee\, \forall i \in \mathcal{N}_\mathsf{T} : n_{i,2} \neq n'_{i,1}. \tag{3.41}$$

The conditions above are the inverse of Equation 3.35. Two vectors are compatible if the vector elements, which may be mapped to the same VA node, are identical, i.e. in case of Condition 3.38 it means that all nodes are equal ($n_{i,1} = n'_{i,1}, \forall i \in \mathcal{N}_\mathsf{T}$). Alternatively, the nodes may be completely independent, i.e. in case of Condition 3.38 all nodes are unequal ($n_{i,1} \neq n'_{i,1}, \forall i \in \mathcal{N}_\mathsf{T}$).

Maximum Weighted Clique Problem

Until now we have described how the problem of finding an edge allocation with minimal reconfiguration cost can be transformed into another problem. The allocation of edges from the input graphs is described by the vectors $\mathbf{n}$. Each vector is associated with a weight $\mathsf{w}(\mathbf{n})$ that describes the quality of the allocation. Further we have derived which allocations defined in the vectors $\mathbf{n}$ can be realized simultaneously. Now, we denote the set of all possible node pair combinations $\mathbf{n}$ as $\mathcal{N}_\mathsf{C}$. The set $\mathcal{E}_\mathsf{C}$ contains all pairs of node pair combinations that are compatible with each other, i.e. $(\mathbf{n}_1,\mathbf{n}_2) \in \mathcal{E}_\mathsf{C}$ if $\mathbf{n}_1,\mathbf{n}_2 \in \mathcal{N}_\mathsf{C}$ are compatible.

Now, the reconfiguration cost are minimal if we choose a subset $\mathcal{N}'_\mathsf{C} \subset \mathcal{N}_\mathsf{C}$ with the following properties:

- the weight over all nodes $\mathbf{n} \in \mathcal{N}'_\mathsf{C}$ is maximal compared to all other possible subsets of $\mathcal{N}_\mathsf{C}$, and
- all nodes in $\mathcal{N}_\mathsf{C}$ are compatible to each other, i.e. $(\mathbf{n}_1,\mathbf{n}_2) \in \mathcal{E}_\mathsf{C}, \forall \mathbf{n}_1,\mathbf{n}_2 \in \mathcal{N}'_\mathsf{C}$.

The problem of finding a subset $\mathcal{N}'_\mathsf{C}$ is known as maximum weighted clique problem (MWCP) in graph theory. The MWCP is known to be NP complete [47]. There exist

different of algorithms to solve the problem. Note that with the MWCP, we can only optimize interconnect reconfiguration but not the resource reconfiguration.

In the following, $G(\mathcal{N}_C, \mathcal{E}_C)$ defines the input to the MWCP. Each node in $\mathcal{N}_C$ defines a part of the allocation function. All nodes of any maximal clique define an allocation function that is sufficient to compute the term x_t, i.e. $x_t = \sum_{\mathbf{n} \in \mathcal{N}'_C} \mathbf{w}(\mathbf{n})$.

In this work, the MWCP has been solved with the algorithm described by Babel [7]. The algorithm performs fast and is easy to implement. It uses a weighted colouring heuristic that computes upper and lower bounds. The branch and bound algorithm takes advantage of the computed bounds. Alternatively, there exist a number of optimal and heuristic methods to solve the MWCP problem.

Maximum Weighted Clique Problem Size

Because the MWCP is a very challenging optimization problem, we derive the problem size in terms of nodes, i.e. the number of node pair combinations $|\mathcal{N}_C|$ here. It is assumed that the input consists of $|\mathcal{N}_T|$ tasks and each task consists of N nodes. The value for $|\mathcal{N}_C|$ is calculated from four components:

$$|\mathcal{N}_C| = C_{\text{all}} - C_{\text{inv}} - C_{\text{sym}} + C_{\text{both}} \tag{3.42}$$

C_{all} describes the number of all possible node pair combinations. A single node pair is one of $(N+1)^2$ combinations since each part of a pair may be one of the N nodes of a task or the void node $\varnothing$. Each node pair can be combined with any node pair of the other tasks, hence

$$C_{\text{all}} = (N+1)^{2|\mathcal{N}_T|}. \tag{3.43}$$

An invalid node pair combination occurs if at least one node pair consists of two equal nodes and one other node pair consists of unequal nodes, which violates condition 3.35. In the following we derive how many of all possible node pair combinations are invalid, i.e. C_{inv}:

$$C_{\text{inv}} = \sum_{i=2}^{|\mathcal{N}_T|} \underbrace{\binom{|\mathcal{N}_T|}{i}}_{\text{(d)}} \left[\sum_{j=2}^{i} \underbrace{\binom{i}{j-1}}_{\text{(c)}} \underbrace{N^{j-1}}_{\text{(a)}} \underbrace{2^{i-j+1} \left(\frac{N(N+1)}{2} \right)^{i-j+1}}_{\text{(b)}} \right]. \tag{3.44}$$

At first we assume that a node pair from one task has been chosen such that it consists of two identical nodes, which is true for exactly N node pairs (cf. Equation 3.44 (a)). Any other task contains now $N+1$ node pairs that are also equal and $2^{\frac{N(N+1)}{2}}$ node pairs that cause an invalid node pair combination ((b)).

The number of combinations of all such invalid node pair combinations is computed for a fixed number of equal node pairs (index j). Now the terms ((a)) and

((ⓑ)) have to be multiplied with the number of combinations that exist for this fixed number of equal node pairs ((ⓒ)). For a node pair that consists of two void nodes there exist only invalid node pair combinations. The number of combinations of void node pairs is accounted for in term ((ⓓ)).

The number of symmetric node pair combinations can be computed as follows: for a single node pair, there exists a symmetric node pair if $n_{i,1} \neq n_{i,2}$. This is true for $\frac{N(N+1)}{2}$ node pairs. For each such node pair, all node pair combinations that contain the symmetric node pair are symmetric. This are $(N+1)^{2(|\mathcal{N}_\mathsf{T}|-1)}$ node pair combinations. The $(N+1)$ node pairs with $n_{i,1} = n_{i,2}$ also part of symmetric node pair combinations, their size is given by $C_{\mathsf{sym}}(|\mathcal{N}_\mathsf{T}| - 1)$. The number of symmetric node pair combinations denoted as a recurrence equation as follows:

$$C_{\mathsf{sym}}(|\mathcal{N}_\mathsf{T}|) = \frac{N(N+1)}{2}(N+1)^{2(|\mathcal{N}_\mathsf{T}|-1)} + (N+1)C_{\mathsf{sym}}(|\mathcal{N}_\mathsf{T}| - 1). \tag{3.45}$$

The recurrence Equation 3.45 can be rewritten in closed form as:

$$C_{\mathsf{sym}} = \sum_{t=1}^{|\mathcal{N}_\mathsf{T}|}(N+1)^{t-1}\frac{N(N+1)}{2}(N+1)^{2(|\mathcal{N}_\mathsf{T}|-t)}. \tag{3.46}$$

The number of the node pair combinations denoted as C_{inv} and C_{sym} count several node pair combinations twice. For each invalid node pair combination there exists a symmetric node pair combination, too. Thus the term C_{both} in Equation 3.42 is given by:

$$C_{\mathsf{both}} = \frac{C_{\mathsf{inv}}}{2}. \tag{3.47}$$

Here, we derive only an upper bound for the number of edges in the MWCP. At the maximum, every node in the MWCP can be connected with any other node and hence:

$$|\mathcal{E}_\mathsf{C}| \leq \frac{1}{2}|\mathcal{N}_\mathsf{C}|(|\mathcal{N}_\mathsf{C}| - 1). \tag{3.48}$$

In Table 3.2, the upper bound from Equation 3.48 and the exact number of edges $|\mathcal{E}_\mathsf{C}|$ in the MWCP are compared for different parameters $N, |\mathcal{N}_\mathsf{T}|$. The values give a hint on the complexity of the MWCP that results from input graphs of a certain size. For instance if we have four input graphs with four nodes each, the number of node pair combinations is already 97865! We observe that, for larger N the number of compatibility edges increases and reaches a large fraction of the upper bound in Equation 3.48.

Table 3.2: Number of compatible nodes $|\mathcal{E}_C|$ in the MWCP compared to the number of edges in a complete graph with $|\mathcal{N}_C|$ nodes. Ratio is the quotient of $|\mathcal{E}_C|$ and $\frac{1}{2}|\mathcal{N}_C|(|\mathcal{N}_C|-1)$.

$\lvert\mathcal{N}_T\rvert$	N	$\lvert\mathcal{N}_C\rvert$	$\lvert\mathcal{E}_C\rvert$	$\frac{1}{2}\lvert\mathcal{N}^C\rvert(\lvert\mathcal{N}^C\rvert-1)$	ratio
2	3	100	1 917	4 950	0.39
2	4	245	11 070	29 890	0.37
2	5	516	50 520	132 870	0.38
2	10	6 281	10 084 815	19 722 340	0.51
2	20	89 061	2 729 326 230	3 965 886 330	0.69
3	3	1 162	89 451	674 541	0.13
3	4	4 755	1 536 879	11 302 635	0.14
3	5	15 111	18 256 305	114 163 605	0.16
4	3	14 536	4 907 025	105 640 380	0.05
4	4	97 865	267 046 572	4 788 730 180	0.06

3.5.2 Direct Allocation of Nodes

In Section 3.5.1 we described a method to derive an edge allocation with minimal reconfiguration cost. The method allocates the edges in the input graphs independently, if possible. Now, we describe a method that determines an optimal allocation for the nodes, such that reconfiguration cost are minimized. This direct method eliminates the need to build an intermediate MWCP to find an optimal allocation. The problem still remains complex to solve: a suitable heuristic is described below, but first an exact algorithm is described.

Iterative Node Allocation

The algorithm discussed here follows an iterative approach to investigate all possible node allocations. The allocations are enumerated in a predefined order. In the algorithm, the allocation is determined successively. For each new node allocation it is computed whether it contributes to a quality solution or whether it can be omitted in the search of further solutions. Therefore, for each node allocation the added reuse and the yet undetermined reuse is computed. If the reuse can not exceed a previously found, best solution then all solutions that contain the unfavourable allocation are discarded from the further iterative exploration. Hence, our algorithm reduces the overall runtime by two methods: (1) the reuse is computed successively for each added allocation and (2) inefficient allocations are discarded from the solution space.

The iterative node allocation algorithm consists of three parts. First, the successive node allocation must be realized such that all possible allocations are enumerated. Second, the quality of the actual allocation is evaluated. And third a

termination condition is computed that excludes inefficient allocations from the solution space. The overall algorithm is given in Algorithm 3. Before the algorithm is discussed in detail, we introduce the used variables and data structures.

Variables and Data Structures In the algorithm it is assumed that the set $\mathcal{N}_i$ of nodes in each input graph is a list with a fixed order, i.e. the nodes in this list are given by $n_{i,v_i} \in \mathcal{N}_i, v_i = 1,\ldots,|\mathcal{N}_i|$. Moreover, nodes can be removed from or inserted into $\mathcal{N}_i$ on specified positions v_i. The variable $\mathbf{v} = (v_1,\ldots,v_{|\mathcal{N}_\mathsf{T}|})$ stores an index vector for the sets $\mathcal{N}_i$. Each element $v_i, i = 1,\ldots,|\mathcal{N}_\mathsf{T}|$ refers to one node n_{i,v_i} in the set $\mathcal{N}_i$. The set $\mathcal{P}$ forms a stack of index vectors $\mathbf{v}$.

An allocation vector $\mathbf{a} = (a_1,\ldots,a_{|\mathcal{N}_\mathsf{T}|})$ contains all nodes $a_i \in \mathcal{N}_i, i = 1,\ldots,|\mathcal{N}_\mathsf{T}|$ that are allocated to the same node in the VA. The current allocation of input nodes is stored in the stack $\mathcal{A}$. The stack contains allocation vectors $\mathbf{a} = (a_1,\ldots,a_{|\mathcal{N}_\mathsf{T}|})$. The best allocation found so far is stored in the stack $\mathcal{A}_\mathsf{max}$.

The quality of the solution, i.e. the metric for the current allocation is stored in the variable w. The metric associated with the best allocation $\mathcal{A}_\mathsf{max}$ is stored in w_max. The potential increase in the metric is stored in w_x. In addition, the stack $\mathcal{W}$ holds the metric w associated with the allocations found in $\mathcal{A}$.

The vector $\mathbf{f} = (f_1,\ldots,f_{|\mathcal{N}_\mathsf{T}|})$ stores the number $f_i, i = 1,\ldots,|\mathcal{N}_\mathsf{T}|$ of currently unallocated edges. The stack $\mathcal{F}$ holds the unallocated edges associated with the allocations found in $\mathcal{A}$.

Table 3.3 summarizes the variables.

Table 3.3: Variables used in the algorithm for iterative node allocation.

Variable	Symbol
Input Nodes	$\mathcal{N}_i$
Index Vector	$\mathbf{v}$
Index Vector Stack	$\mathcal{P}$
Allocation Vector	$\mathbf{a}$
Allocation Stack	$\mathcal{A}$
Allocation Stack of the best Solution	$\mathcal{A}_\mathsf{max}$
Current Solution Metric	w
Solution Metric of the best Solution	w_max
Potential Solution Metric	w_x
Solution Metric Stack	$\mathcal{W}$
Unallocated Edges	$\mathbf{f}$
Unallocated Edge Stack	$\mathcal{F}$

Enumeration of Allocations All different allocations of nodes from the input graphs to VA nodes are enumerated in Algorithm 3. The algorithm ensures that an allocation is not considered more than once. The enumeration is realized as

an iterative algorithm. It is assumed that the nodes are partially allocated during the iterations, i.e. some nodes from the input graphs are allocated whereas other nodes are not allocated. For any partial allocation, all possible allocations for the yet un-allocated nodes are investigated. If the algorithm determines that for a partial allocation, the currently best solution can not be improved then the exploration of un-allocated nodes is discarded.

In more detail, the algorithm works as follows: At first, the allocation of all nodes specified by the index vector **v** is stored in the allocation vector **a**. The allocated nodes are temporarily removed from the sets $\mathcal{N}_i$. In preparation of the next iteration, the vector **f** is stored on $\mathcal{F}$. Next, we add the cost savings caused by the reuse of edges to the metric w. Here, the metric is incremented according to the reuse that is induced by the existing allocation $\mathcal{A}$ and the additional allocation in **a**. Hence, all edges between the nodes in $\mathcal{A}$ and the nodes in **a** are considered, as opposed to the node pair combinations discussed in Section 3.5.1. Now the number of unallocated edges is updated in **f** and subsequently a potential metric increase is calculated. In preparation of further allocations with the current partial allocation as a starting point, the allocation vector **a** is added to $\mathcal{A}$ and the index vector **v** to $\mathcal{P}$. Note that the set $\mathcal{N}_i$ of input nodes only contain unallocated nodes and hence, the exploration starts again with $\mathbf{v} := (1, \ldots, 1)$.

The exploration of unallocated nodes and the enumeration of different allocations is realized in lines 17–30 of Algorithm 3. First, the index vector **v** is incremented if possible using *permuteNext* (Algorithm 4). Usually this yields a new allocation of nodes that is processed as described above, which implements a depth-first exploration of possible allocations. Otherwise, if the current allocation can not surpass the metric of the best solution, i.e. if $w + w_x < w_{\max}$, then the algorithm discards the current allocation induced by the incremented index vector **v** and continues from the last allocation stored on $\mathcal{A}$. Therefore, the variables are retrieved back from the stacks $\mathcal{A}$, $\mathcal{P}$, $\mathcal{W}$, and $\mathcal{F}$.

Algorithm 3 Iterative Node Allocation

1: **Input:** $\mathcal{N}_i, i = 1, \ldots, |\mathcal{N}_\mathsf{T}|$
2: **Output:** $\mathcal{A}_{\max}$, $w_{\max}$
3: $\forall i \in \{1, \ldots, |\mathcal{N}_\mathsf{T}|\} : v_i = 1$;
4: **repeat**
5: $\quad \forall i \in \{1, \ldots, |\mathcal{N}_\mathsf{T}|\} : a_i = n_{i,v_i}$, erase all nodes at $\mathcal{N}_{i,v_i}$
6: $\quad$ put **f** on stack $\mathcal{F}$ and w on stack $\mathcal{W}$
7: $\quad w \mathrel{+}= \mathit{getMatchWeight}(\mathcal{A}, \mathbf{a})$;
8: $\quad \mathbf{f} = \mathit{getFreeEdges}(\mathcal{A}, \mathbf{a}, \mathbf{f})$;
9: $\quad w_x = \mathit{getPotentialWeight}(\mathbf{f})$;
10: $\quad$ put **a** on stack $\mathcal{A}$ and **v** on stack $\mathcal{P}$
11: $\quad \forall i \in \{1, \ldots, |\mathcal{N}_\mathsf{T}|\} : v_i = 1$;
12: $\quad$ **if** $w > w_{\max}$ **then**

13: $w_{\max} = w$;
14: $\mathcal{A}_{\max} = \mathcal{A}$;
15: **end if**
16: valid = **false**;
17: **repeat**
18: (**v**, valid) = *permuteNext*(**v**, $\mathcal{N}_1, \ldots, \mathcal{N}_{|\mathcal{N}_T|}$);
19: **if** valid = **false and** $\mathcal{A} = \emptyset$ **then**
20: **break;**
21: **end if**
22: **if** $w + w_x < w_{\max}$ **or** $w_x = 0$ **then**
23: valid = **false**;
24: **end if**
25: **if** valid = **false then**
26: remove **a** from stack $\mathcal{A}$, **v** from stack $\mathcal{P}$, w from stack $\mathcal{W}$, and **f** from stack $\mathcal{F}$
27: $\forall i \in \{1, \ldots, |\mathcal{N}_T|\}$ insert a_i back at V_{i,v_i}
28: w_x = *getPotentialWeight*(**f**);
29: **end if**
30: **until** valid
31: **until not** valid

The sub-program *permuteNext* (Algorithm 4) simply increments the index vector **v** similar to a decimal counter. Each digit in a decimal counter would be represented by an vector element v_i. Here, each vector element v_i is in the range $1, \ldots, |\mathcal{N}_i|$. If the increment of one element exceeds the range of v_i, then the previous element v_{i-1} is incremented and all other elements $v_k, k = i, \ldots, |\mathcal{N}_T|$ are set to $v_k = 1$. If **v** can not be incremented any further then the function returns valid=**false**.

Algorithm 4 select next node allocation (*permuteNext*)

1: **Input:** $\mathbf{v} \in \mathbb{N}^{|\mathcal{N}_T|}$, $\mathcal{N}_i, i = 1, \ldots, |\mathcal{N}_T|$
2: **Output:** **v**, valid
3: valid = **false**;
4: $i = |\mathcal{N}_T|$;
5: **while** $i > 1$ **and** valid = **false do**
6: **if** $v_i < |\mathcal{N}_i|$ **and** $|\mathcal{N}_i| > 0$ **then**
7: valid = **true**;
8: $v_i = v_i + 1$;
9: **else**
10: $v_i = 1$;
11: $i = i - 1$;
12: **end if**
13: **end while**

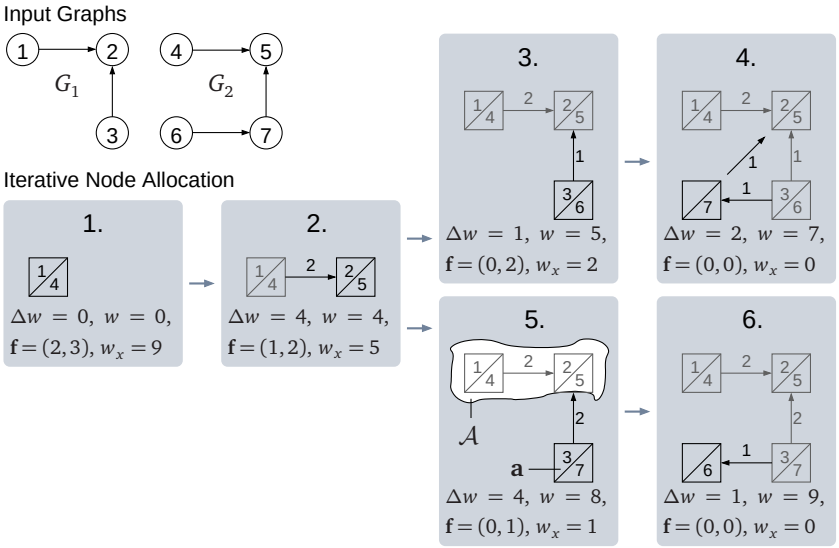


Figure 3.12: An example for the iterative node allocation. The figure shows the first six iterations (1.–6.) of the algorithm. In each iteration a new allocation **a** (black print) are added to the previous allocation $\mathcal{A}$ (gray print). For each iteration, some variables are shown, the edges are labelled with u(e).

.

14: **return** valid;

In the following example we illustrate the enumeration of allocations by Algorithm 3.

Example 3.11 *Consider the iterative allocation for input graphs G_1 and G_2 shown in Figure 3.12. At first, the nodes* 1 *and* 4 *are allocated to the same resource. Because there are unallocated nodes in the input graphs $\mathcal{N}_i$, the algorithm continues and allocates the nodes* $(2,5)$*,* $(3,6)$*, and* $(\text{-},7)$*. Now, there are no unallocated nodes available. Thus the state (2.) is restored from the stack and the iterations continue. Next, the function permuteNext yields the allocation* $(3,7)$*. The algorithm continues until all possible allocations are investigated. Figure 3.12 shows only the first six iterations of the algorithm.*

The Solution Metric for the allocation is computed successively. It is assumed that the metric w is known for an allocation $\mathcal{A}$. Now, the function *getMatchWeight* computes how the metric is increased if the new allocation **a** is added to $\mathcal{A}$.

Formally, the metric increase is calculated as follows: Assume that the allocation stack is given by $\mathcal{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_l\}$ with $\mathbf{a}_k = (a_{k,1}, \ldots, a_{k,|\mathcal{N}_\mathsf{T}|}), k = 1, \ldots, l$ and $a_{k,i} \in \mathcal{N}_i, i = 1 \ldots, |\mathcal{N}_\mathsf{T}|$. It is further assumed that all nodes $a_{k,i}$ within the same allocation vector $\mathbf{a}_k$ are allocated to the same VA node, i.e. $\mathsf{a}(a_{k,i}) = n'_k \in \mathcal{N}_\mathsf{A}$. Now, a new allocation vector $\mathbf{a}$ is added to $\mathcal{A}$ with $\mathbf{a}_{l+1} = \mathbf{a}$. The edges induced by this additional allocation are given by the set:

$$\mathcal{E}_\mathsf{A}^{(\mathcal{A},\mathbf{a})} = \{e \in \mathcal{E}_\mathsf{A} : e = (n'_k, n'_{l+1}) \vee e = (n'_{l+1}, n'_k) \vee e = (n'_{l+1}, n'_{l+1}), \forall k \in \{1, \ldots, l\}\}, \tag{3.49}$$

where the set $\mathcal{E}_\mathsf{A}$ denotes all edges in the VA. The function *getMatchWeight* computes the metric Δw, which results from the new allocation vector as follows:

$$\Delta w(\mathcal{A}, \mathbf{a}) = \sum_{e \in \mathcal{E}_\mathsf{A}^{(\mathcal{A},\mathbf{a})}} u^2(e). \tag{3.50}$$

The current metric w is saved on stack $\mathcal{W}$, then the term Δw is added to the metric w in line 7. In the algorithm, the best solution is recorded in $\mathcal{A}_\mathsf{max}$ and w_max (lines 12–15 of Algorithm 3).

Example 3.12 (continued from Example 3.11.) *Suppose* $\mathcal{A} = \{(1, 4)\}$ *and* $\mathbf{a} = (2, 5)$. *There is an edge* $(1, 2)$ *in graph* G_1 *and an edge* $(4, 5)$ *in graph* G_2 *that is allocated to the same edge* e *in the VA, i.e.* $\mathsf{u}(e) = 2$. *Both edges have one already allocated node (*1 *and* 4*) and one node in* $\mathbf{a}$ *(*2 *and* 5*). The added metric* Δw *evaluates to* $\Delta w = \mathsf{u}^2(e) = 4$.

Computing an Upper Bound for the Metric For a large number $|\mathcal{N}_\mathsf{T}|$ of tasks and for input graphs with many nodes the number of possible allocations becomes very large. Many allocations can be omitted from the computation when it can be shown that they can not yield an optimal allocation. Hence, the iterative node allocation can be truncated if the upper bound of the current metric is smaller than the current best solution w_max, cf. Algorithm 3, lines 22–24. Therefore we compute an upper bound of the metric w for the current allocation $\mathcal{A}$.

We require the computation of the upper bound to be simple, because otherwise the computation time for the upper bound may outweight the benefits of a reduced runtime for the enumeration of allocations.

Here, we use the following assumption to compute the upper bound: Consider there are f edges in each input graph that are not yet allocated to an edge in the VA because at least one node of the edge is not allocated yet. In the best possible case, there are f edges in each input graph such that $|\mathcal{N}_\mathsf{T}|$ edges are allocated to the same edge $e \in \mathcal{E}_\mathsf{A}$, i.e. $\mathsf{u}(e) = |\mathcal{N}_\mathsf{T}|$. Hence the added weight Δw for all f edges is bounded by:

$$\Delta w \leq f\,\mathsf{u}^2(e) = f\,|\mathcal{N}_\mathsf{T}|^2. \tag{3.51}$$

The idea has been extended such that the number of unallocated edges in $\mathbf{f} = (f_1, \ldots, f_{|\mathcal{N}_\mathsf{T}|})$ does not have to be equal for all elements. The algorithm *getPotentialWeight* (Algorithm 5) computes the metric that can be achieved by yet unallocated edges, cf. Algorithm 5. First, the vector elements $\mathbf{f}$ are sorted in increasing order. Thus it is known that every input graph contains at least f_1 unallocated edges, i.e. in the first iteration of the loop (line 6), $w_x = f_1|\mathcal{N}_\mathsf{T}|^2$. For f_2, only $|\mathcal{N}_\mathsf{T}| - 1$ tasks contain a certain number of unallocated edges. Here, all f_1 edges have already been taken into account, hence for $i = 2, w_x = w_x + (f_2 - f_1)(|\mathcal{N}_\mathsf{T}| - 1)^2$.

Algorithm 3 maintains a count of unallocated edges in $\mathbf{f}$ that is updated for every allocation using a function *getFreeEdges* (Algorithm 3, line 8). The function *getFreeEdges* computes the number of edges in each input graph that are not allocated to an edge in the VA by an allocation $\mathcal{A}, \mathbf{a}$. $\mathbf{f}$ is saved on stack $\mathcal{F}$ during the enumeration of all allocations. Hence, for each iteration, the term $w + w_x$ yields an upper bound of the metric.

Algorithm 5 compute the potential solution metric (*getPotentialWeight*)

```
1: Input: f
2: Output: w_x
3: w_x = 0
4: sort f in increasing order
5: f' = 0
6: for 1 ≤ i ≤ |N_T| do
7:     w_x += (f_i − f')(|N_T| − i + 1)^2
8:     f' = f_i
9: end for
10: return w_x
```

Example 3.13 (continued from Example 3.12.) *Suppose there is an allocation stack* $\mathcal{A} = \{(1,4)\}$ *and an allocation vector* $\mathbf{a} = (2,5)$. *The edge* $(3,2)$ *in* G_1 *and the edges* $(6,7)$ *and* $(7,5)$ *in* G_3 *are not allocated yet, i.e.* $\mathbf{f} = (1,2)$. *The function getPotentialWeight yields* $w_x = 4+1$ *because the edge* $(3,2)$ *could be allocated to the same edge in the VA as one of the edges in* G_2. *The other edge in* G_2 *can not be reused by* G_1.

Number of Allocations Here we want to determine the number of possible allocations for a set of input graphs. We assume that each input graph contains N nodes and that there are $|\mathcal{N}_\mathsf{T}|$ input graphs.

First, the number of *complete allocations* is calculated, i.e. the number of different allocations that result from the allocation of all nodes to the VA. In order to avoid any symmetry in the allocation, Algorithm 3 assumes a fixed allocation for one task and enumerates all permutations of node allocations for the other tasks. Hence, the number of different complete allocations C_{ca} is given by:

$$C_{\mathsf{ca}} = 1 \times (N!)^{|\mathcal{N}_\mathsf{T}|-1}. \tag{3.52}$$

However, the iterative node allocation algorithm evaluates the cost metric not only for complete allocations, but also for all intermediate partial allocations. Now we derive how many partial allocations exist.

Assume that the allocation stack $\mathcal{A}$ contains l allocations. With the rules of combinatorics, the number of different allocations that contain exactly l allocations in $\mathcal{A}$ is given by:

$$\left(\frac{N!}{(N-l)!}\right)^{|\mathcal{N}_\mathsf{T}|-1}. \tag{3.53}$$

The total number of partial and complete allocations C_{pa} that can occur during enumeration (without truncation) is given by:

$$C_{\mathsf{pa}} = \sum_{l=1}^{N} \left(\frac{N!}{(N-l)!}\right)^{|\mathcal{N}_\mathsf{T}|-1}. \tag{3.54}$$

It can be seen, the number of different allocations and thus the algorithm complexity increases over-exponentially for N and exponentially for $|\mathcal{N}_\mathsf{T}|$. Apparently, the problem can be solved for small input graphs in reasonable time with the presented algorithm. The efficiency of the truncation condition depends on the structure of the input graphs. Therefore a manageable number of allocations can not be guaranteed for general input graphs.

Direct Allocation with Simulated Annealing

In the previous section we have shown that the number of different allocations becomes very large for many tasks with many nodes. Therefore we have implemented a heuristic optimization approach that enables us to find near-optimal allocations with a limited computation time. However, the heuristic can not guarantee an optimal allocation with respect to reconfiguration cost, but in general the approach achieves high-quality results.

Here, we use a simulated annealing (SA) based optimization method. The concept has been introduced by Kirkpatrick [49]. SA is a metaheuristic that can be adapted to a range of nonlinear optimization problems. The general approach is described in Appendix A. In summary, SA works as follows: The algorithm starts with an initial, non-optimal allocation. In every iteration, the current allocation is slightly modified and the cost of the modified allocation are computed. The modified allocation may become the new, current allocation, depending on the progress of the optimization and the difference between the cost for the current allocation and the modified allocation. During the iterations, the best overall allocation is recorded and represents the near-optimal allocation found by SA.

In order to apply this general scheme to the allocation problem, we have to define three methods: 1) a method that generates an initial allocation, 2) a method

that modifies the current allocation randomly to generate a new allocation, and 3) a function to compute the cost of the new allocation.

Initial Allocation (*getInitialSolution*) Initially, the nodes of the input graphs are allocated in any order to nodes in the VA. The number of VA resources that is required for the allocation depends on the size of the largest input graphs, i.e.

$$|\mathcal{N}_\mathsf{A}| = \max_{i \in \mathcal{N}_\mathsf{T}} |\mathcal{N}_i|. \tag{3.55}$$

Generate a new Allocation (*PermuteSolution*) The SA approach requires that the current allocation is modified slightly. For the allocation problem it is straightforward to modify the allocation for a few nodes only while the allocation remains valid. Here, we choose the following method:

1. Select randomly one node $n_1 \in \mathcal{N}_i$ from any of the tasks $i \in \mathcal{N}_\mathsf{T}$. In the current solution, n_1 is allocated to $n_1' \in \mathcal{N}_\mathsf{A}$.
2. Select randomly a new allocation $n_2' \in \mathcal{N}_\mathsf{A}$ for this node, i.e. set $\mathsf{a}(n_1) = n_2'$.
3. If another node $n_2 \in \mathcal{N}_i$ is already allocated to n_2', then change the allocation of n_2 to $\mathsf{a}(n_2) = n_1'$.

In summary, the modified allocation is derived from the current allocation by the exchange of the allocation for two nodes of one task.

Note that the modification in terms of node allocation is small. However, because a node can have many related edges, the effect on the edge allocation a_e can be more profound.

Allocation Cost Function (*getSolutionCost*) The allocation of nodes from the input graphs also defines the allocation of edges. Therefore we can determine the reconfiguration cost caused by the allocation. The average reconfiguration time can be computed according to Equation 3.4 (Section 3.2.1). Alternatively, average reconfiguration time for interconnect can be computed with Equation 3.34. The average configuration size is given in Equation 3.8 (Section 3.2.2).

In this section we have shown how the allocation of nodes can be optimized such that low reconfiguration cost can be achieved. The methods described here allow a simultaneous optimization of node allocation and interconnect allocation, in contrast to the allocation of node pairs presented in Section 3.5.1.

3.5.3 Experiments

In this section we present some experimental data on randomly generated input graphs. The goal is to compare the performance of the different approaches described in Sections 3.5.1 and 3.5.2. It is also discussed how the complexity of the solution algorithms relates to the theoretical bounds in practice. Further we are

interested in the performance of the SA-based heuristic compared to the optimal solutions computed with the exact algorithms.

First, the general setup of the experiments is described. In the following we analyze the complexity of the introduced algorithms in practice. Finally, the experimental results are discussed and some general conclusions for reconfiguration cost are drawn.

Benchmark Set-Up

We generated random input graphs with N nodes each. We assume that each node has k different input ports and one output port, similar to a k-input LUT or an operation with k arguments. All input ports of each node were connected randomly to an output port of a node. We run the experiments for a different number $|\mathcal{N}_T|$ of tasks. We assumed a complete RSG in all cases. The optimization goal was set to minimize the average reconfiguration time. We do not include the reconfiguration cost for nodes here, but only the interconnect related cost.

The results shown were averaged over 10 different sets of input graphs with the same parameters for $N, |\mathcal{N}_T|$, and k. The runtime of the solution algorithms was restricted to 1800 sec. If the runtime limit was exceeded, the results were not included in the results. The experiments were run on a Pentium4, 2.4 GHz with 1 GB RAM.

Allocation of Node Pairs

At first, an optimal allocation in terms of reconfiguration cost has been computed with the allocation of node pairs. It is known that the related MWCP is hard to solve, thus we attempted to solve only MWCPs with less than 100,000 nodes and 2,000,000 edges. In this section we do not focus on the results, but on the effective MWCP complexity. The complexity depends on the parameters of the input graphs $N, |\mathcal{N}_T|$, and k.

In Section 3.5.1 we calculated a theoretical upper bound of the number of nodes in the MWCP (Equation 3.42). In the implementation we omit all nodes with $w(\mathbf{n}) = 0$ (cf. Equation 3.37) from the MWCP, because these nodes do not contribute to the overall solution. We found that the effective number of nodes in the MWCP is considerably lower than suggested by Equation 3.42. The effective MWCP complexity is shown in Table 3.4.

It appears that the effective MWCP size grows considerably if the number of nodes N or tasks $|\mathcal{N}_T|$ are increased as it is suggested by Equation 3.42.

Also, the MWCP size increases with the number of LUT inputs k. An input graph with N nodes contains kN edges. If an input graph with the same number of nodes contains more edges, it is more likely that a node pair combination yields a weight $w(\mathbf{n}) \neq 0$ and hence, the effective MWCP problem becomes more complex.

For increasing N, the effective MWCP size becomes a smaller fraction of the theoretical MWCP size. Nevertheless, the decrease is slow, while the absolute increase in terms of problem complexity is exponential. As a consequence, the MWCP can only be solved for small problems in reasonable time. The computed optimal solutions can compared to other approaches in order to evaluate their performance.

Table 3.4: Complexity of the MWCP for different parameters N, $|\mathcal{N}_\mathsf{T}|$, and k.

Input Graphs		Theoretical MWCP Size		Effective MWCP Size			
				$k=2$		$k=4$	
N	$\lvert\mathcal{N}_\mathsf{T}\rvert$	$\lvert\mathcal{N}_\mathsf{C}\rvert$	$\lvert\mathcal{E}_\mathsf{C}\rvert$	$\lvert\mathcal{N}_\mathsf{C}\rvert$	$\lvert\mathcal{E}_\mathsf{C}\rvert$	$\lvert\mathcal{N}_\mathsf{C}\rvert$	$\lvert\mathcal{E}_\mathsf{C}\rvert$
3	2	100	1 917	8.8	6.8	14.7	19.5
4	2	245	11 070	17.6	28.8	32.6	92.8
5	2	516	50 520	29.9	88.0	60.6	388.5
10	2	6 281	10 084 815	162.0	5 595.5	307.1	19 820.7
20	2	89 061	2 729 326 230	722.9	171 852.8	1 452.5	691 320.0
3	3	1 162	89 451	232.3	996.6	369.2	2 667.1
4	3	4 755	1 536 879	894.1	16 646.3	1 464.9	47 267.0
5	3	15 111	18 256 305	2 298.6	192 100.9	4 137.6	618 508.3
3	4	14 536	4 907 025	4 481.2	85 513.4	6 396.5	188 320.2

Iterative Node Allocation

The iterative node allocation is described in Section 3.5.2. Here we discuss the effective complexity of the algorithm for the input graphs compared to the theoretical complexity. As above, the parameters of the random input graphs are N, $|\mathcal{N}_\mathsf{T}|$, and k.

In Table 3.5, the number of enumerated (partial) allocations are compared. The theoretical complexity is given by the number of partial (C_pa) and complete allocations (C_ca). The effectively investigated number of partial and complete allocations are denoted as C'_pa and C'_ca. In Table 3.5, it is shown that the number of investigated allocations can be reduced to a small extend only. It seems that the computation of the upper bound is possibly based on very optimistic assumptions, or the quality of the allocation becomes apparent very late, i.e. for a large allocation stack $\mathcal{A}$. There are little differences between $k=2$ and $k=4$.

Direct Allocation with Simulated Annealing

The allocation has been optimized for minimal reconfiguration cost using the SA algorithm described in Section 3.5.2. The results are shown in Table 3.6 and Table 3.7. Via simulated annealing, it was easily possible to solve large input problems, with

up to $N = 500$ nodes per task, within the given time limit. With very few exceptions, the computational results match exactly those computed with the exact solution algorithms.

Results

As discussed before, we run the allocation algorithms for different parameters N, $vv|\mathcal{N}_T|$, and k. The results are shown in Table 3.6 and Table 3.7 for $k = 2$ and $k = 4$ respectively.

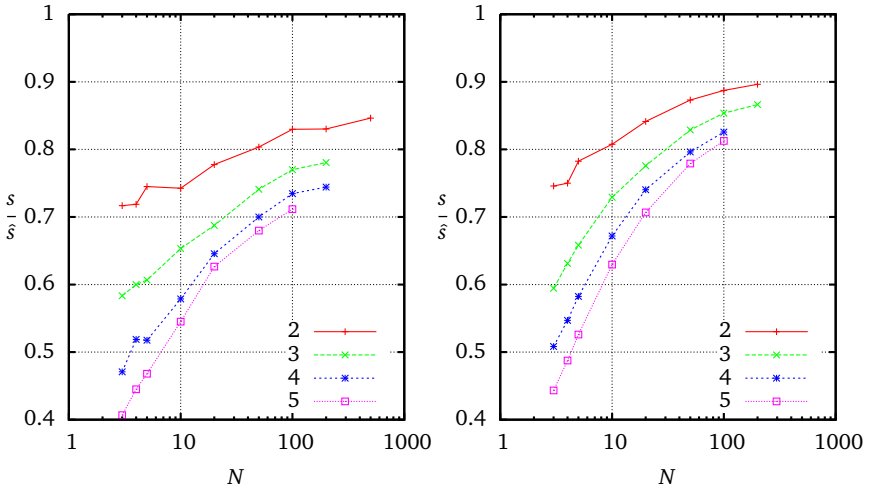
Note that only the reconfiguration cost for interconnect is considered here.

The results obtained with the allocation of node pair combinations could be confirmed with the iterative node allocation algorithm. The iterative node allocation requires only a small amount of memory during the enumeration of allocations. Further, the allocation of node pair combinations is solved as a complex MWCP problem. As a result, the iterative node allocation algorithm is able to handle some larger problems compared to the MWCP method, cf. to Tables 3.6 and 3.7. The SA-based allocation algorithm can handle much larger problems (up to 500 nodes per task), even though the quality of results can not be compared to other solutions.

Figures 3.13 and 3.14 show the results obtained by the SA algorithm compared to the upper bound of the reconfiguration cost. The configuration size for the interconnect is bounded by $\hat{s}$. If there is no reuse of interconnect in the VA, then every edge is allocated only once. There are $\hat{s} = kN|\mathcal{N}_T|$ edges in all input graphs. Now we define the relative configuration size as $\frac{s}{\hat{s}}$. Similarly, the reconfiguration time between any two input graphs is $2kN$ if we assume there are no matching edges between the graphs. There are exactly $|\mathcal{N}_T|(|\mathcal{N}_T| - 1)$ such reconfigurations. Hence, the upper bound for the total reconfiguration time is $\hat{t} = 2kN|\mathcal{N}_T|(|\mathcal{N}_T| - 1)$. Now we define the relative reconfiguration time as $\frac{t}{\hat{t}}$. Note that the relative reconfiguration costs compare the reconfiguration cost with a known similarity to the reconfiguration cost, which is caused if the interconnect is completely reconfigured.

As an example, consider the results obtained for the parameters $N = 3, k = 2$, and $|\mathcal{N}_T| = 2$. The upper bounds for the configuration size and the reconfiguration time are $\hat{s} = 12$ and $\hat{t} = 24$. However, the nodes of the input tasks can be allocated such that $s = 8$ and $t = 10$. This means the relative configuration size is now only $\frac{8}{12} \cdot 100\,\% = 66.7\,\%$ and the relative reconfiguration time is only $\frac{10}{24} \cdot 100\,\% = 41.7\,\%$ of the reconfiguration cost's upper bound.

The Figures 3.13(a) and 3.13(b) show the relative configuration size for $k = 2$ and $k = 4$ for several numbers $|\mathcal{N}_T| = \{2, 3, 4, 5\}$ of tasks, respectively. It can be observed that, for an increasing number $|\mathcal{N}_T|$ of tasks, the relative configuration size decreases. Hence, more interconnect in the VA is reused. We conclude that if the the VA must support more different tasks, then fewer additional interconnect must be provided in order to accommodate the additional tasks.



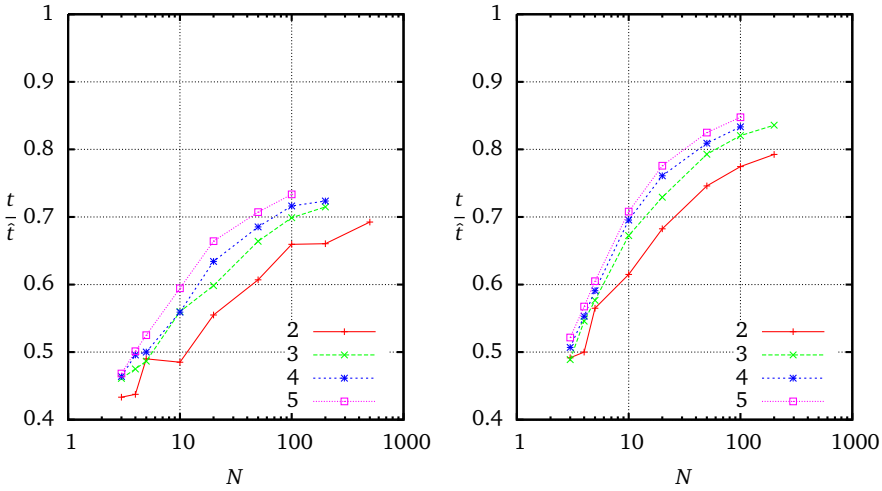
(a) Relative configuration size for interconnect, $k = 2$.

(b) Relative configuration size for interconnect, $k = 4$.

Figure 3.13: Relative configuration size for random input graphs computed with the SA algorithm for different parameters k, N. The four plots show the results for $|\mathcal{N}_T| = \{2, 3, 4, 5\}$.

The relative reconfiguration time is shown in Figures 3.14(a) and 3.14(b) for $k = 2$ and $k = 4$, respectively. In contrast to the relative configuration size, the relative reconfiguration time increases with the number of tasks $|\mathcal{N}_T|$. Hence on average, more interconnect must be reconfigured if there are more different tasks allocated to the VA, even though the complexity of a single task remains constant. This behaviour can be explained as follows: For more different tasks, the overall configuration size of the VA interconnect increases (see above). Hence, the possibility that the edges of two tasks are allocated to the same interconnect decreases, which increases reconfiguration time.

We observe that for larger N the relative reconfiguration cost increase substantially, cf. Figures 3.13 and 3.14. This can be explained by the fact that for larger N, more variations in the input graphs exist. Thus, a reuse of many interconnects in the VA becomes less likely. The effect is more prominent for $k = 4$, because in this case smaller graphs (in terms of nodes) contain already more edges that should be reused. Moreover, the reuse of interconnect is determined by the allocation of nodes. With $k = 4$, each node allocation influences the allocation of 5 edges. Thus the allocation of edges is more restricted than with $k = 2$.



(a) Relative reconfiguration time for interconnect, $k = 2$.

(b) Relative reconfiguration time for interconnect, $k = 4$.

Figure 3.14: Relative reconfiguration time for random input graphs computed with the SA algorithm for different parameters k, N. The four plots show the results for $|\mathcal{N}_T| = \{2, 3, 4, 5\}$.

While the results are very promising for tasks with a small number of nodes, the performance decreases for $N > 20$. This can be rooted in the performance of the simulated annealing method, which can not be justified with an exact solution, but it is more likely caused by the allocation problem itself. Although the experiments were run with random graphs, this behaviour can be a general trend also for real-life structural representations. However, we expect the performance in real task sets to be much better because:

a) Real-life circuits often contain common structures (macros), generated by the synthesis tools. We will deal with structural representations of circuit netlists in Chapter 4.

b) Structural representations compiled from high-level languages have fewer nodes than circuit netlists, hence it is more likely to find reconfiguration-efficient allocations. Moreover, these structural representation allow for resource sharing, i.e. several nodes of one task can be mapped to the same VA resource. High-level synthesis with the objective of low reconfiguration cost will be described in detail in Chapter 5.

3.6 Summary

In this chapter we have introduced the RSG model. It has been shown how the RSG model can be used to describe both, the configurations and the reconfiguration between configurations. We have described two cost metrics based on the RSG, the reconfiguration time and the configuration data. The RSG model has been applied to frame-based, binary configuration data in order to evaluate configuration cost. We have shown that the RSG can be applied to structural representations of tasks, too. Therefore we define an allocation of the structural representations to a VA. The allocation and the VA is used to derive a configuration that can be used in conjunction with the RSG in order to evaluate reconfiguration cost. Further we have studied the problem of finding an optimal allocation such that reconfiguration cost become minimal. Three approaches have been developed and their usefulness has been demonstrated with experiments. The experiments indicate that it can be expected to reduce reconfiguration cost substantially with an optimal allocation and the difference based reconfiguration model. The application of the methods will be described in the following chapters.

Table 3.5: Number of allocations enumerated for different input problem sizes.

Input Graphs		Theoretical Complexity		Effective Complexity			
				$k = 2$		$k = 4$	
N	$\lvert\mathcal{N}_T\rvert$	C_{pa}	C_{ca}	C'_{pa}	C'_{ca}	C'_{pa}	C'_{ca}
3	2	15	6	14.2	6.0	14.7	6.0
4	2	64	24	53.1	23.1	55.7	24.0
5	2	325	120	242.8	116.9	287.2	118.1
10	2	9 864 100	3 628 800	1 268 280.3	910 366.6	4 315 564.0	2 646 961.6
3	3	81	36	78.3	36.0	80.9	36.0
4	3	1 312	576	1 065.9	576.0	1 253.4	576.0
5	3	32 825	14 400	21 519.2	13 883.4	28 441.3	14 400.0
3	4	459	216	450.2	216.0	459.0	216.0
4	4	29 440	13 824	24 083.8	13 824.0	29 120.7	13 824.0
5	4	3 680 120	1 728 000	2 218 421.2	1 720 446.3	3 346 974.0	1 728 000.0
3	5	2 673	1 296	2 646.8	1 296.0	2 673.0	1 296.0
4	5	684 544	331 776	548 859.3	331 776.0	684 077.4	331 776.0

Table 3.6: Reconfiguration cost for interconnect with $k = 2$. The computation of the configuration size s and reconfiguration time t is given in Equations 3.3 and 3.7.

		No Reuse		SA Solution		Exact Solution	
N	$\lvert\mathcal{N}_T\rvert$	s	t	s	t	s	t
3	2	12	24	8	10	8	10
3	3	18	72	10	33	10	33
3	4	24	144	11	66	11	66
3	5	30	240	12	112	12[†]	112[†]
4	2	16	32	11	14	11	14
4	3	24	96	14	45	14	45
4	4	32	192	16	95	16[†]	94[†]
4	5	40	320	17	160	17[†]	158[†]
5	2	20	40	14	19	14	19
5	3	30	120	18	58	18	58
5	4	40	240	20	120	20[†]	119[†]
5	5	50	400	23	210		
10	2	40	80	29	38	29	37
10	3	60	240	39	134		
10	4	80	480	46	268		
10	5	100	800	54	475		
20	2	80	160	62	88		
20	3	120	480	82	287		
20	4	160	960	103	608		
20	5	200	1600	125	1062		
50	2	200	400	160	242		
50	3	300	1200	222	796		
50	4	400	2400	280	1644		
50	5	500	4000	339	2828		
100	2	400	800	331	527		
100	3	600	2400	462	1677		
100	4	800	4800	587	3438		
100	5	1000	8000	711	5867		
200	2	800	1600	664	1056		
200	3	1200	4800	936	3431		
200	4	1600	9600	1190	6946		
500	2	2000	4000	1692	2770		

[†] These results could only validated with the iterative allocation method.

Table 3.7: Reconfiguration cost for interconnect with $k = 4$. The computation of the configuration size s and reconfiguration time t is given in Equations 3.3 and 3.7.

		No Reuse		SA Solution		Exact Solution	
N	$\lvert\mathcal{N}_T\rvert$	s	t	s	t	s	t
3	2	24	48	17	23	17	23
3	3	36	144	21	70	21	70
3	4	48	288	24	146	24	146
3	5	60	480	26	250	26[†]	250[†]
4	2	32	64	24	32	24	32
4	3	48	192	30	104	30	104
4	4	64	384	35	212	35[†]	212[†]
4	5	80	640	39	363	39[†]	363[†]
5	2	40	80	31	45	31	45
5	3	60	240	39	138	39	138
5	4	80	480	46	283	46[†]	283[†]
5	5	100	800	52	484		
10	2	80	160	64	98	64	97
10	3	120	480	87	322		
10	4	160	960	107	667		
10	5	200	1600	125	1132		
20	2	160	320	134	218		
20	3	240	960	186	700		
20	4	320	1920	236	1461		
20	5	400	3200	282	2482		
50	2	400	800	349	596		
50	3	600	2400	497	1902		
50	4	800	4800	637	3884		
50	5	1000	8000	779	6599		
100	2	800	1600	709	1239		
100	3	1200	4800	1024	3937		
100	4	1600	9600	1321	7999		
100	5	2000	16000	1625	13561		
200	2	1600	3200	1434	2536		
200	3	2400	9600	2079	8024		

[†] These results could only validated with the iterative allocation method.

Chapter 4

Implementation Tools for Reconfigurable Computing

In Chapter 3 we have introduced a generic reconfiguration cost model. The model is based on an abstract representation of a reconfigurable architecture. The abstract architecture model consists of resources and the interconnect between resources. This enables a very concise formal description of the allocation of structural representations to such an architecture. However, existing FPGA architectures have features that are not reflected in the formal model. In this chapter, we describe these features and their effect. We describe two different tools that take advantage of the FPGA features, whereas the allocation problem is solved. The tools optimize the mapping of synthesized netlists to FPGAs, not the synthesis from RTL or high-level descriptions itself. High-level synthesis will be described in Chapter 5.

In an FPGA implementation flow, a digital circuit is given as a synthesized netlist. The netlist is translated to a configuration bitstream in a three-step approach: (1) the netlist is mapped to a device specific netlist. While the synthesized netlist consists of basic sequential and combinational logic, the device specific netlist consists of resource instances that are available in the device, e.g. logic elements (LEs), RAM blocks, and multipliers. Each resource instance is associated with a configuration of that instance. The configuration is derived from the functionality of the sequential and combinational logic that is mapped to the resource instance. (2) the device specific netlist is placed and routed on an FPGA device, i.e. resource instances are placed on physical resources in the device and the interconnect between resource instances is routed using physical wires and switches in the device. At this stage, the netlist is still a structural representation, but there is already a fixed association between resource instances and interconnect with the configurable elements in the device. (3) The placed and routed netlist is transcribed into a configuration bitstream. In current FPGA implementation flows, the steps 1–3 are repeated for each reconfigurable task independently. The reconfiguration costs depend on the result of the implementation flow. The differences between the configuration bitstreams of

different digital circuits define the reconfiguration costs as described in Chapter 3. Currently, the implementation of digital circuits is not optimized for low reconfiguration cost, because the digital circuits are translated independently and there has been no model to assess reconfiguration cost during the steps 1 and 2.

In this chapter we describe how our generic reconfiguration cost model can be used in the implementation flow. At first, in Section 4.1 we discuss some effects that arise from the mapping of synthesized netlists to device specific netlists in step 1. In the Sections 4.2 and 4.3 we describe two different tools: The first tool computes an allocation to the virtual architecture and outputs information on matching nodes and matching edges. The second tool computes a mapping of synthesized netlists to device specific netlists. Thereby the tool treats multiple netlists at the same time. The aim of the tool is to compute a mapping with minimal cost for the reconfiguration of interconnect.

4.1 Mapping of Netlists to FPGA Resources

The synthesized netlists are structural representations of the different tasks. The synthesized netlists can be represented as LMG as shown in Section 3.4.1. However, the mapping of elements in the synthesized netlists to resources in a real FPGA architecture is not as straightforward as the allocation of nodes in the input graphs to a resources in a VA, because often a one-to-one mapping is not possible. Whereas our cost model provides a reasonable abstraction, a tool that can be applied in a real implementation flow must take advantage of the special properties of an FPGA architecture.

We discuss the mapping on single device resources first and on the overall device afterwards.

4.1.1 Mapping to Device Resources

The properties of an FPGA architecture result in several effects that can occur during the mapping process. The analysis presented here is based on the Xilinx VirtexII/VirtexII-Pro FPGA architecture, which has been introduced in Section 2.2.1. Similar effects can be observed for other FPGA architectures, too. We concentrate the discussion on the properties of the LEs here, because these is the most often used resource type in the FPGA. The LE is called a *Slice* in the documentation provided by Xilinx [107][105]. The LEs consist of several internal resources for logic and routing. The properties of the LE architecture can be summarized as follows:

- several different functions can be mapped to the same type of resource in the LE,
- some functions can be mapped in several ways to resources in the LE,

- there are several semi-independent logic resources in one LE,
- some of the resources types are available more than once in one LE, e.g. LUTs and flip-flops,
- there exists local interconnect between the logic resources inside the LE,
- some local interconnects are fixed while others are configurable, and
- the LEs contain specialized logic for specific functionality.

We observed that the resources and interconnect in an LE can be reconfigured with a small number of reconfiguration frames (cf. Figure 2.3).

Miscellaneous digital logic functions can be realized on configurable logic in an FPGA, whereas many different logic functions are realized by using the same type of device resource, e.g. LEs. In the VirtexII architecture, an LE contains two 4-input LUTs, two flip-flops, and other logic. In this architecture, different types of registers (e.g. synchronous set or reset, asynchronous clear, with or without enable, active at the rising or falling edge of a clock signal) are realized by using the same flip-flop resource in an LE. The configuration of the resource determines the actual function of the flip-flop.

Example 4.1 *Figure 4.1 depicts different register types (FD, FDC, and FDCE) that can be present in synthesized netlists. All types can be mapped to the same flip-flop resource in the LEs. Therefore the configuration of the flip-flop resource is chosen such that it realizes the functionality required by the register type. For example, to realize the FDCE type (edge triggered register with asynchronous clear and data enable) the following configuration is chosen: enable the driver on ports CE and SR to connect the inputs to the routing matrix, set the reset type to Async for asynchronous reset, set SRLow to set the register content to '0' if SR is triggered, set Init0 to initialize the register content to '0' on power-up, and set FF to implement an edge triggered register.*

A function can be mapped such that the same functionality is realized on the same kind of resource, but with a different resource configuration. This is illustrated in the following example:

Example 4.2 *The boolean function $o = i_1 + \overline{i_2}$ can be mapped in six different ways onto a 3-input LUT. In Figure 4.2(a) the schematic that realizes the function is shown. The circuit in Figure 4.2(b) depicts the 3-input LUT. The variables b_0–b_7 represent the configurable LUT content. One of the variables is presented by the multiplexer on the output o. The select signal for the multiplexer is given by the input signals a_1, a_2, a_3.*

Table 4.1 shows all possible mapping variants of the boolean function to the LUT. For each variant, the realized function and the configuration of the variables b_0–b_7 is given. The mapping defines—in addition to the LUT configuration—a mapping of the original boolean function arguments to the input signals a_1, a_2, a_3 of the LUT. For example, in the first variant the argument i_1 is mapped to the input a_1 and the argument i_2 is mapped to the input a_2.

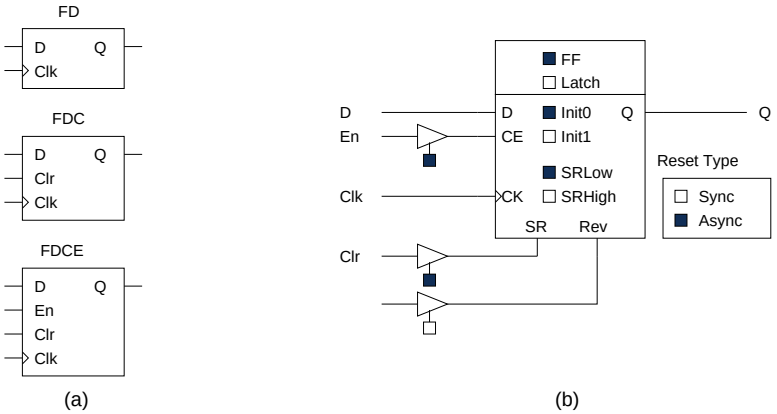


Figure 4.1: (a) Register symbols from a synthesized netlist. (b) Flip-flop resource in an LE configured as FDCE register.

Table 4.1: Mapping variants of the boolean function $o = i_1 + \overline{i_2}$ to a 3-input LUT.

LUT Function	Input Port Map		LUT Configuration
	i_1	i_2	"$b_0b_1b_2b_3\,b_4b_5b_6b_7$"
$o = a_1 + \overline{a_2}$	a_1	a_2	"1101 1101"
$o = a_1 + \overline{a_3}$	a_1	a_3	"1111 0101"
$o = a_2 + \overline{a_1}$	a_2	a_1	"1011 1011"
$o = a_2 + \overline{a_3}$	a_2	a_3	"1111 0011"
$o = a_3 + \overline{a_1}$	a_3	a_1	"1010 1111"
$o = a_3 + \overline{a_2}$	a_3	a_2	"1100 1111"

We already mentioned that a VirtexII LE contains two resources to realize 4-input LUTs (LUT G, LUT F) and registers (FF Y, FF X). Apparently, any function in the synthesized netlist that can be realized with a 4-input LUT can be mapped to LUT G or LUT F. In practice, the functions are not isolated in the netlist, but connected to other functions. Therefore, the mapping depends also on the routability (and timing requirements) of the interconnect between functions.

For example the outputs of combinational functions are often connected to registers in the synthesized netlist. Inside the LE there exists a direct, configurable connection between the LUT output and the flip-flop input, i.e. between LUT G and FF Y, and between LUT F and FF X. Therefore if a combinational function is mapped to one of the LUTs, the connected registers must be mapped to the corresponding flip-flop in order to obtain a timing-optimized implementation of the circuit. The

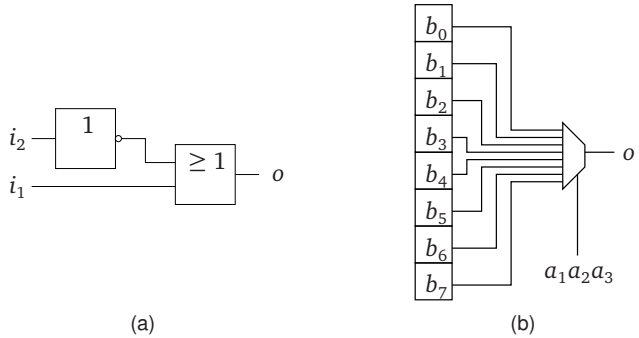


Figure 4.2: A circuit diagram of the boolean function $o = i_1 + \overline{i_2}$ (a) and a 3-input LUT the function is mapped to (b).

situation becomes even more complex if larger circuit structures such as full-adder circuits are mapped to device specific netlists.

The effects that occur during the mapping must be observed, when the circuit similarity is analyzed and during the mapping process that takes advantage of the similarity, cf. Sections 4.2 and 4.3.

4.1.2 Connectivity Transformations

In the previous sections we discussed how the mapping to device specific netlists effects the resource configuration. Now, we discuss how the connectivity is transformed by such a mapping. The interconnect between the different resources in an FPGA is realized in two different ways: There exists (configurable) local interconnect inside the LEs and configurable interconnect between the LEs, which is realized with the FPGAs routing resources.

If a synthesized netlist is mapped to a device specific netlist then the original netlist elements are mapped to resources inside the LEs. Thereby several netlist elements may be mapped into one LE. As a result, the interconnect of the synthesized netlist is transformed during the mapping, too. The following effects can be observed:

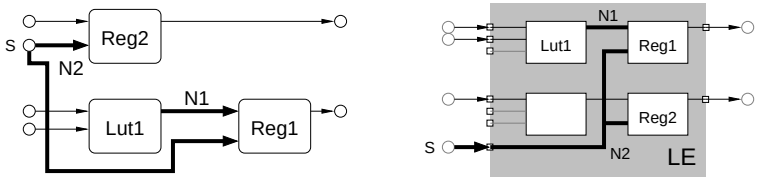
- Connections between netlist elements that are mapped into the same LE can become local interconnect. Local interconnect does not utilize the FPGA routing resources. As already mentioned local interconnect is either static or it can be reconfigured at low costs.
- The mapping of multiple netlist elements to one LE can also lead to a reduced utilization of FPGA routing resources. Consider two connections with

the same source but different drains. If the drains are mapped to one LE and the two connections are realized using one pin of the LE, then there is only one connection to be routed using the FPGA routing resources; from the source to the pin of the LE.

Both effects are illustrated on Example 4.3.

Example 4.3 *Consider the synthesized netlist illustrated in Figure 4.3(a) and the device specific netlist in Figure 4.3(b). The connection labelled N1 in the synthesized netlist is mapped to a local connection inside the LE. The connection is not routed through the FPGA routing fabric.*

The connections from the source S to Reg1 and Reg2 are mapped as follows. Because both Reg1 and Reg2 are mapped to one LE, the source S needs to be connected only once to the LE using the routing fabric. The connection to both Reg1 and Reg2 is realized inside the LE. Thus the mapping has merged two connections in the synthesized netlist to one connection in the device specific netlist.



(a) Synthesized Netlist (b) Mapped Netlist

Figure 4.3: A netlist and the mapping to a LE. The connection N1 becomes a local connection inside the LE and both connections from the source S are reduced to one connection outside the LE.

4.1.3 Mapping Variants and Reconfiguration Costs

We observed that there exist a number of mapping variants for some elements of the synthesized netlists. The mapping variants result in different transformations of the interconnect. More specifically, the connections must be realized between different pins of a LE, because a netlist element can be mapped to different resources inside the LE. Thus the resources are connected via different LE pins to the FPGA routing resources. This may have a huge effect on the similarity of the circuits and on the reconfiguration cost.

Example 4.4 *Consider the example depicted in Figure 4.4. The tasks 1 and 2 contain the elements A–D, which are mapped to different resources in the LEs. If Task 2 is mapped according to the mapping variant 2, then two interconnects (bold lines) must*

not be reconfigured between Task 1 and Task 2. This is not the case if mapping variant 1 for Task 2 is chosen instead.

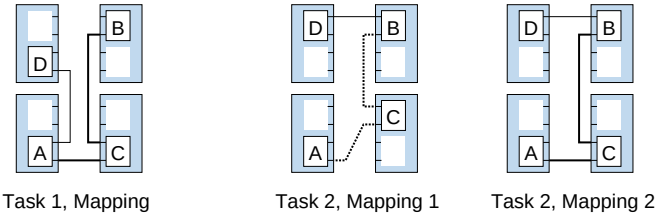


Figure 4.4: Mapped netlists of a Task 1 and two possible mapping variants for Task 2.

4.1.4 Mapping of Circuit Macros

As already mentioned the synthesized netlists often contain specialized circuitry that prefers a specific mapping of netlist elements to LE resources, e.g. in the case of LUT logic and registers. Furthermore, the synthesized netlists can contain circuitry that requires such a specific mapping, it can not be mapped in any other way. Examples are large multiplexers, shift register logic, and carry logic for adder structures.

Example 4.5 *Consider the netlist of a 4-bit full adder. The Xilinx ISE synthesis tool generates a netlist that can be mapped directly to LEs. It consists of one 2-input LUT, one 2-to-1 multiplexer, and one XOR function per bit. Whereas the LUTs in the LE have a very versatile connectivity, i.e. all inputs and outputs can be directly connected to the FPGA routing resources, the 2-to-1 multiplexer and the XOR function is only accessible by other resources in the LE.*

Figure 4.5 depicts the mapping of the 4-bit full adder two 2 LEs. The adder inputs are provided via the G1, G2, F1, and F2 pins, the adder outputs appear on the pins X and Y of each slice. The interconnect between the LUT, the 2-to-1multiplexer and the XOR function is realized internally in the LEs. The connection of the carry path (COUT to CIN) is a direct interconnect between two LEs. Hence, both LEs must be placed next to each other as shown here.

The circuit macros do not only constrain the mapping of the circuitry to LE resources, but the relative placement of the resulting LE is also constrained. A tool that analyzes the similarity between circuits has to observe these constraints. Otherwise, it is possible that the tool assumes a placement of matched LE that can not be realized in the FPGA architecture, due to special routing/placement constraints.

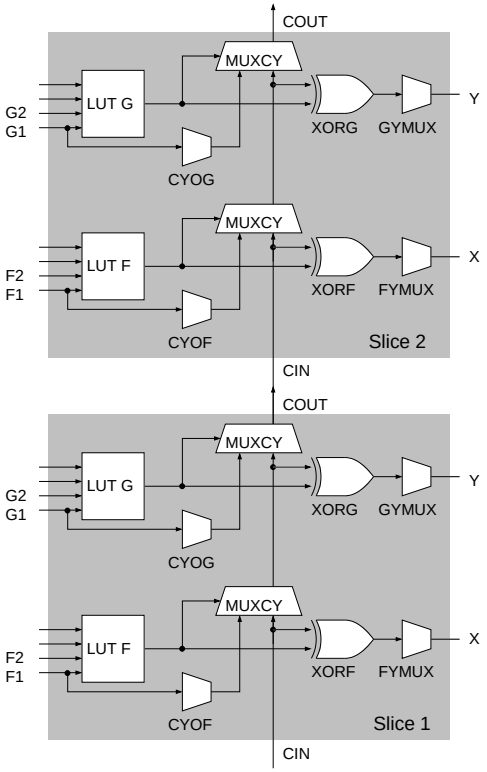


Figure 4.5: Implementation of a 4-bit full adder on two VirtexII Slices.

4.1.5 Global Interconnect

Most synthesized netlists contain a few special interconnects. For example some input pins are connected statically to logic “0”, “1”, or to clock signals. For these signals, special routing structures exist in the FPGA architecture. The clock signals are routed using a dedicated clock distribution on the FPGA. Static signals with a high fan-out are converted to multiple static signals that are placed local to the LEs. Hence both kind of interconnect does not utilize the generic routing architecture of the FPGA. Therefore such global signals can be reconfigured with minimal effort. It follows that the structural similarity must not be exploited for global signals.

4.1.6 Netlist Hierarchy

The synthesized netlists do not necessarily consist of circuit primitives only. Instead, the netlists can instantiate complete sub-circuits as a basic building block. Eventually the netlist can consist of a hierarchy of sub-circuits. For the assessment of reconfiguration cost a circuit hierarchy provides both benefits and drawbacks.

At first, if a sub-circuit is used in several reconfigurable tasks then it is known that this part of the task is identical and causes no reconfiguration overhead if it is mapped, placed and routed equally in those tasks. Hence, reconfiguration cost can be reduced by re-using the sub-circuit in several tasks.

However, if the reconfiguration cost are only analyzed at one level in the netlist hierarchy, the complex sub-modules can occlude circuit similarity.

After several effects that occur during the device mapping have been discussed, it is shown how the effect can be considered during the reconfiguration cost optimization.

4.2 Mapping Aware Allocation

The Section 3.4.1 we have shown how the allocation of nodes and edges can be performed such that minimal reconfiguration cost are achieved. Therefore we used a simplified resource model that assumes a one-to-one mapping of netlist elements to resources in a VA. Here, we outline a tool that takes into account the effects caused by the mapping to a real FPGA architecture, as discussed in Section 4.1.1.

The goal of the tool is to identify an allocation for *two* different netlists such that the reconfiguration cost for interconnect are minimal. The interconnect reconfiguration cost are calculated for the interconnect contained in the synthesized netlists, not for the interconnect of a device specific netlist. The tool accepts the netlists in a industry standard netlist format, called Electronic Design Interchange Format (EDIF) [28]. The tool provides both an exact optimization algorithm (cf. Section 3.5.2) and an algorithm based on an established optimization heuristic.

The tool represents the original netlists as input graphs. Circuit/netlist elements are represented as nodes and connections between circuit elements are represented as edges in the input graphs. Here, we assume that each circuit element—and hence, each node—is associated with a node type.

In this section we describe the features of the mapping aware allocation tool. We discuss how the mapping effects are considered in the allocation algorithms and we describe the implementation of the optimization heuristic employed in the tool. Some details of the tool were initially described in [75].

4.2.1 Generalized Node Mapping

The tool takes into account how the circuit elements are mapped to device resources. As described in Section 4.1.1, there exists a multitude of mapping variants for many circuit elements. The user of the tool can provide a constraint file that describes how the nodes and edges in the input graphs are effected by such a mapping. In the tool we do not generate a mapped circuit, but instead the reconfiguration cost optimization is based on architecture specific assumptions about the mapping.

These assumptions are specified in the constraint file. It can be described, what kind of circuit elements are realized on the same resource type in the FPGA. It can be specified how the interconnect will be transformed by the mapping. Finally, the constraint file may contain user specified allocations in order to incorporate a priori knowledge about the allocation.

The tool will only allocate nodes to the same resource that will be mapped to the same resource type in the FPGA. Because the resource type itself is not of interest, we only need to specify which node types will be mapped to the same resource type and how this effects the interconnect.

Example 4.6 *The netlist contains LUTs of different complexity, e.g. elements with the node types LUT2, LUT3, and LUT4. All those node types will be mapped to either an LUT G or an LUT F resource in a VirtexII LE. In general it is not important, which resource type is chosen, but only that the node of this type can be allocated to the same resource type.*

As we have already discussed in Section 4.1.1, there exist several mapping variants for some node types. The allocation of the edges in the input graphs to connections in the VA must be modelled accordingly. For the transformation of input graphs to image graphs, we introduced the port re-labelling functions a_s, a_d in Section 3.4.1. The edges in the input graphs describe the connection between the *ports* of the netlist elements. After allocation, which can include a re-labelling of ports, the edges in the image graph describe the connection between the ports of LEs in the FPGA. However in our tool we are only interested in the edges of the input graphs, that can be allocated to the same edge in the VA, i.e. in the edges that match between both input graphs. Therefore the re-labelling is considered in an implicit way, similar to the node types discussed above.

In the constraint file, it can be defined how the source and drain labels of nodes of a certain type are treated. Whether the source or drain labels of two edges in the input graphs are re-labelled to the same source or drain label in the image graph is specified by the user. There are several possibilities:

- Equal source or drain labels are re-labelled to equal source or drain labels.
- Any source label can be re-labelled to any source label and any drain label in can be re-labelled to any drain label.

- Labels like bus $N[x]$ that belong to an index $x = 1,\ldots,X$ of a bus $N[1:X]$ can be re-labelled to the same index of any other bus. This feature is especially useful to compute the reconfiguration cost for netlists that contain regular structured sub-circuits. E.g. it allows to match edges associated with the same index of a bus, but with different source or drain labels.
- The re-labelling can be defined explicitly for each source or drain label.

The allocation that respects the mapping of the netlist is illustrated with the Examples 4.7 and 4.8.

Example 4.7 *Consider the register types shown in Figure 4.1(a). All register types have port names that are mapped to the same port of the flip-flop resource inside a VirtexII LE. For example any edge with the drain label 'D' that is connected to a node with node type FD, FD, or FDCE will be mapped to the LE-internal port D of either the FF X or the FF Y resource.*

As mentioned before, the behaviour of the resource depends on the resource configuration and the connections with other resources.

Example 4.8 *For LUT logic, there exist several mapping variants (cf. Example 4.2). In Figure 4.6 two nodes of the node types LUT3 and LUT2 are shown. Here we assume that the interconnect sig1–sig4 with the same name should be matching edges. The mapping of the nodes to 4-input LUTs can be done in that manner, regardless of the original port labels.*

For example LUT3 can be mapped to the 4-input LUT as shown in Figure 4.6(c) and LUT2 can be mapped to the 4-input LUT as shown in Figure 4.6(d). Hence, the edge sig3 that leads to LUT3, port I2 is mapped such that it connects to 4-input LUT, port A2. Similarly, the edge sig3 that leads to LUT2, port I1 is mapped to 4-input LUT, port A2.

The tool allows to incorporate the designer's knowledge about the circuit structure into the reconfiguration cost optimization. Therefore it can be specified in the constraint file which netlist elements of the input graphs can be allocated to the same resource. The specification limits the allocation variants and therefore decreases the solution time of the algorithms. Furthermore, with this method a meaningful allocation of the netlist elements that belong to a circuit macro (cf. Section 4.1.4) can be obtained.

4.2.2 Successive Node Allocation

The tool implements an exact solution algorithm that is based on the algorithm described in Section 3.5.2 with the extensions described in Section 4.2.1. In its specialized form, the algorithm computes the allocation of two given circuits to a

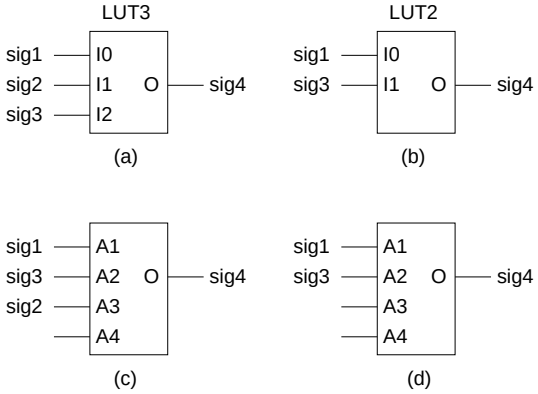


Figure 4.6: Example of a port re-labelling that maps edges with the same name to the same port of the 4-input LUT, cf. Example 4.8. The source node of the edges sig1–sig3 and the drain node of the edge sig4 is not shown here.

VA with the objective of minimal reconfiguration cost. The output of the tool consists of matching nodes and matching edges, which define the structural similarity of the circuits obtained for the allocation.

The annotation of netlist elements (i.e. nodes in the input graphs) with node types and the allocation restriction of netlist elements causes a reduction in the number of possible allocations. Thus, the allocations investigated with the successive node allocation algorithm is also reduced. Here, we compute the number of different allocations that is possible under the current assumptions. Each node in an input graph is associated with a node type t from a set $\mathcal{T}$ of node types. We assume there are $N_{1,t}$ and $N_{2,t}$ nodes of the node types $t \in \mathcal{T}$ in the input graphs G_1 and G_2. Now the number of complete allocations for the two input graphs is given by:

$$\prod_{t \in \mathcal{T}} \left(\frac{N_{i,t}!}{(N_{i,t} - N_{j,t})!} \right) \tag{4.1}$$

where $i = 1, j = 2$ if $N_{1,t} > N_{2,t}$ and $i = 2, j = 1$ otherwise.

The introduction of node types reduces the search space for an optimal solution considerably. However, for many relevant applications, a complete enumeration of the search space is not feasible with an acceptable algorithm runtime. The computation of an upper bound for the reconfiguration cost metric discussed in Section 3.5.2 can alleviate the problem to some extend.

4.2.3 Node Allocation with Ant Colony Optimization

In addition to the SA annealing method described in Section 3.5.2, we also developed a solution algorithm for the allocation problem based on ant colony optimization (ACO). The algorithm originates from the early developments of this work. Our ACO algorithm is based on the work of Stützle and Hoos [89]. The approach has been chosen, because the heuristic has a good performance and the computation of the cost function is similar to the successive node allocation. It has been demonstrated before that ACO algorithms are competitive with other heuristic methods on well known optimization problems e.g. the travelling salesman problem and quadratic assignment problem (cf. [89] for further references). Although some details have already been published in [75], the algorithm is described in this section in detail.

ACO is a biologically inspired meta heuristic that mimics swarm intelligence. We applied the algorithm to the allocation problem (cf. Definition 3.6). In ACO, many individuals (i.e. *ants*) successively construct solutions of the given problem, where each ant constructs one solution. A solution is constructed step-by-step. In each step, a partial solution is chosen randomly, based on the quality of the partial solution in the current context and the quality of the partial solution with respect to previous solutions. The partial solutions of the previous steps form the context for the new partial solution.

In the following we describe the adaptation of the ACO algorithm to the allocation problem in detail. For now, a solution may be identified by m. In each iteration t of the ACO there are M solutions generated. The allocation associated with the solution m is denoted as $\mathsf{a}^{(m)}$. The input to the algorithm are the input graphs G_1 and G_2. Next we describe how a single solution is constructed and then we describe the iterations of the ACO.

An allocation $\mathsf{a}^{(m)}$ is constructed as follows: Initially the set $\mathcal{A}$ of node pairs is empty. The set $\mathcal{A}$ contains the node pairs $\mathbf{a} = (n_1, n_2)$, where the nodes n_1 and n_2 are allocated to the same resource in the VA, i.e. $\mathsf{a}(n_1) = \mathsf{a}(n_2) = a_k \in \mathcal{N}_\mathsf{A}$. In each step, the ACO selects a node pair $\mathbf{a} = (n_1, n_2)$ with $n_1 \in \mathcal{N}_1, n_2 \in \mathcal{N}_2$, where the nodes n_1 and n_2 are not already contained in any other node pair in $\mathcal{A}$. Now the node pair is added to $\mathcal{A}$. This method is continued until there are no further un-allocated nodes.

The solutions that are generated by the algorithm depend on the selection of the node pairs. In each step there exists a variety of node pairs that can be added to $\mathcal{A}$. The probability to select one node pair depends on two parameters: the pheromone level τ resembles the contribution of this node pair to the cost metric known from the previous iterations and local heuristic weight η describes the contribution of the node pair to the cost metric in the context of $\mathcal{A}$.

The ACO is an iterative algorithm. In each iteration a number of different allocations are constructed. The allocations found during one iteration are used to

update the pheromone levels τ of the node pairs. With an increasing number of iterations, node pairs that lead to good overall solutions strengthen the weight of τ. Subsequently it becomes more likely that these node pairs are selected when a new allocation is constructed. Thus the allocations are constructed from "good" node pair allocations. Often this allocations are good local optima of the allocation problem but a global optimum can not be guaranteed.

The following steps describe an iteration t of the ACO algorithm:

1. For each solution m, construct an allocation function $\mathsf{a}^{(m)}$. The probability $p_{(n_1,n_2)}(t)$ to allocate a node pair $(n_1,n_2), n_1 \in \mathcal{N}_1, n_2 \in \mathcal{N}_2$ to the resource $n_k \in \mathcal{N}_\mathsf{A}$, i.e. $\mathsf{a}^{(m)}(n_1) = \mathsf{a}^{(m)}(n_2) = n_k \in \mathcal{N}_\mathsf{A}$, is:

$$p_{(n_1,n_2)}(t) = \frac{[\tau_{(n_1,n_2)}(t)]^\alpha \cdot [\eta_{(n_1,n_2)}]^\beta}{\sum_{\forall(n_1',n_2')}[\tau_{(n_1',n_2')}(t)]^\alpha \cdot [\eta_{(n_1',n_2')}]^\beta} \tag{4.2}$$

   where α, β are tuning parameters for the ACO algorithm. $\tau_{(n_1,n_2)}$ and $\eta_{(n_1,n_2)}$ resemble the pheromone levels and the local heuristic weight for the node pair, respectively. The local heuristic weight is given by Equation 3.50, i.e. $\eta_{(n_1,n_2)} = \Delta\mathsf{w}(\mathcal{A},(n_1,n_2))$. The sum in the denominator normalizes the probability such that the sum of the probability over all possible node pairs equals one. Here, $\forall(n_1',n_2')$ denotes all possible node pairs that consist of nodes, which have not been allocated in $\mathcal{A}$ yet.

2. After all M allocations have been constructed, the pheromone levels τ are updated based on the rules:

$$\tau_{(n_1,n_2)}(t+1) = (1-\rho)\cdot\tau_{(n_1,n_2)}(t) + \sum_{m=1}^{M}\Delta\tau^{(m)}_{(n_1,n_2)}(t), \tag{4.3}$$

$$\Delta\tau^{(m)}_{(n_1,n_2)}(t) = \begin{cases} \mathsf{w}(\mathcal{A}) & \text{if } (n_1,n_2)\in\mathcal{A} \text{ for} \\ & \text{solution } m \text{ in iteration } t \\ 0 & \text{otherwise} \end{cases}. \tag{4.4}$$

   Equation (4.3) implements a decay of the pheromone level from the previous iteration (evaporation) and an enforcement of the weight depending on the quality of the current solutions. The enforcement is proportional to the reuse of interconnect in the VAs derived for all allocations $\mathsf{a}^{(m)}$, which contain the node pair (n_1,n_2). The term $\mathsf{w}(\mathcal{A})$ denotes the metric derived from the reuse of edges in the VA, i.e. $\mathsf{w}(\mathcal{A}) = \sum_{e\in\mathcal{E}_\mathsf{A}^{(\mathcal{A})}} u^2(e)$ (cf. Equation 3.50). The evaporation is controlled by the parameter ρ.

3. Continue with the next iteration $t+1$ at step 1, until the desired number of iterations is reached.

The best allocation found in all iterations represents the (sub-)optimal solution with the maximum reuse and hence, the lowest reconfiguration cost.

The approach of the ACO algorithm is similar to the successive node allocation algorithm. In both algorithms, the allocations are constructed successively by adding additional node pairs to already investigated allocations. In the case of the successive node allocation, any combination of such allocations is investigated until the allocation is proved to be non-optimal. The ACO constructs a large number of different allocations. The allocations are investigated randomly, controlled by the information derived from previous solutions. For each selection, Equation 4.2 must be computed for every possible node pair, which can be costly if many nodes are available. In comparison with the simulated annealing (SA) algorithm presented in Section 3.5.2, the allocation of nodes is modified randomly in SA. There, the changes in the cost function arise from a few, re-allocated nodes only. Thus, the cost function in SA can be evaluated more efficiently.

4.2.4 Examples

Here we reproduce the experimental result already presented in [75] in order to compare the two approaches for mapping aware allocation presented here. The examples are small, such that it is possible to compute the optimal solution with both algorithms. The results are summarized in Table 4.2.

Table 4.2: Experimental results of the successive node allocation and the ACO algorithm taken from [75]. The reconfiguration time $t_{\mathcal{E}}$ for interconnect is given in Equation 3.24.

Test Set	Solution Time [sec]		Time/Allocation [sec]		$t_{\mathcal{E}}$	
	SNA†	ACO	SNA†	ACO	without reuse	with reuse
(A) Random $\lvert\mathcal{N}_1\rvert = 12, \lvert\mathcal{E}_1\rvert = 40$ $\lvert\mathcal{N}_2\rvert = 12, \lvert\mathcal{E}_1\rvert = 40$	278	0.92	$2.4 \cdot 10^{-6}$	$3.2 \cdot 10^{-4}$	160	60
(B) Adder/Subtractor $\lvert\mathcal{N}_1\rvert = 25, \lvert\mathcal{E}_1\rvert = 44$ $\lvert\mathcal{N}_2\rvert = 28, \lvert\mathcal{E}_2\rvert = 49$	1	0.41	$9.6 \cdot 10^{-7}$	$2.6 \cdot 10^{-4}$	186	18

† successive node allocation.

In the table it can be seen that the overall solution time of the ACO is much better in both examples. As already discussed, the computation of a single allocation for the ACO algorithm requires much more time compared to the successive node allocation. In the table we computed the reconfiguration cost ($t_{\mathcal{E}}$ w/o reuse) without any reused interconnect (i.e. $\mathsf{u}(e) = \{0, 1\}$) and $t_{\mathcal{E}}$ with reuse that results

from the optimal allocation (i.e. $u(e) = \{0, 1, 2\}$). For the above examples, the algorithms identified 25 matching edges out of 40 in (A) and 42 matching edges out of 44/49 in (B). In both examples, the reconfiguration time for the interconnect could be reduced substantially, i.e. from $t_{\mathcal{E}} = 160$ to $t_{\mathcal{E}} = 60$ in (A) and from $t_{\mathcal{E}} = 186$ to $t_{\mathcal{E}} = 18$ in (B).

4.3 Netlist Mapping with Minimized Reconfiguration Cost

In the previous section it has been shown, how the similarity can be computed for synthesized netlists. It has been mentioned that the netlists are translated into device specific netlists by a mapping tool. During this translation the mapping tool maps the synthesized netlist to a device specific netlist. The device specific netlist consists of elements that are realized using LEs and of interconnect that is realized using the FPGA's routing resources.

In most fine grain reconfigurable devices, LEs can implement several elements from the synthesized netlist. Therefore, elements from the synthesized netlist are packed to LEs. Often, there exist multiple variants to map an element to a resource inside the LE. In Section 4.2 we described a tool that solves the allocation problem where the mapping of netlist elements to resources inside the LE is anticipated.

In this section we describe our mapping tool [76] that performs the mapping of several synthesis netlists to the device specific netlists at the same time. The mapping is performed with the objective of minimal reconfiguration cost in terms of interconnect. Therefore the allocation constraints computed by the tool in Section 4.2 are observed. In addition the tool exploits the existing mapping variants to obtain optimized solutions. The mapping tool can be used for modern, complex FPGA architectures such as Xilinx VirtexII—in contrast to previous approaches that assume a simplistic LE structure [57]. However, our objectives are minimal reconfiguration cost, not timing or resource optimizations.

The result of the mapping tool are the device specific netlists. Each netlist consists of elements that are instances of LEs associated with a configuration of that instance. The LE configuration describes the behaviour of the resources in the LE and the connectivity inside the LE. The netlists also contain connections between the ports of the LE instances.

The connectivity inside a LE can be reconfigured very efficiently because it requires very few configuration data. Hence, the connectivity inside the LE is not relevant for the computation of reconfiguration cost of the device specific netlists.

In the following we describe the implementation of the mapping tool. At first it is described how the information on how to map synthesized netlist elements is obtained. Then a mapping algorithm is presented to ensure correct mapping of

several elements to the same LE. The proposed mapping algorithm provides several mapping variants for a range of elements. Finally a method is described that selects the mapping variants that are optimal in terms of reconfiguration costs.

4.3.1 Mapping Database

The mapping tool uses two device specific databases that contain information on how to map netlist elements to device resources.

One database describes the device architecture itself. In our implementation we use a database generated with the Xilinx XDL tool. The XDL tool can generate architecture descriptions for all Virtex and Spartan device families. The architecture description consists of a definition of LE types and of a definition of the device architecture, which describes where the LE types are placed and how they are connected. The mapper uses the definition of the LE types as input. An LE type definition describes the LE as a circuit, very similar to a netlist. The definition also contains the configuration possibilities of all resources inside an LE. However, the functionality of resources is not defined in the formal architecture description. This information can be obtained from the technical documentation, e.g. [107].

A second database contains the mapping rules for an architecture. The mapping rules define, how the elements of a synthesized netlist can be mapped to resources in an LE. One element can occupy more than one resource inside the LE, also there may be several possibilities to map one element to resources inside an LE. The mapping rules describe only the mapping of netlist elements, the routing inside the LE between internal resources is determined automatically by the mapping tool. The LE type definitions in the XDL generated database are used for the automatic routing.

Some netlist elements require special treatment of the device mapper that cannot be expressed easily by mapping rules. However, we tried to keep such elements minimal and implemented the mapping tool as generic as possible.

One example are LUTs. The configuration of an LUT inside the LE depends on LUT defined in the synthesized netlist and the chosen mapping variant for that LUT. Therefore the configuration of the LUT inside the LE is computed by the device mapper depending on those parameters. For an example refer to Example 4.2.

The employed mapping algorithm considers only routability inside the LEs. For a few exceptions, the interconnect between the LEs requires an additional pre-processing step. In the pre-processing, special circuit structures are extracted from the synthesized netlist. The structures exploit features in the architecture that improve the implementation of commonly used logic such as arithmetic (e.g. adders that use the fast carry chain, cf. Example 4.5), large multiplexers, and shift registers. The extracted elements are packed according to the requirements of the architecture in order to generate a routable device specific netlist.

4.3.2 Mapping and Packing of Elements into Logic Blocks

Today's FPGAs have increasingly complex LEs with support for shift registers, carry chain logic, large multiplexers etc. This often requires packing of specific netlist elements to ensure routability, since not all resources in an LE are connected to the FPGA's routing resources. In this section we present a mapping algorithm which automatically packs elements into LEs that must be connected with local routes. The algorithm is designed to avoid additional processing of special circuit structures as much as possible. Another objective is the generation of multiple mapping variants for many elements of the synthesized netlist.

Now we describe our approach for a mapping algorithm that automatically maps netlist elements to LE resources and packs the elements such that (1) routability inside the LE, and (2) the mapping of locally connected elements to a single LE is enforced. The major advantage of the algorithm is the independence from any specific target architecture. The mapping algorithm is described in Algorithm 6. It is performed independently for each synthesized netlist. The algorithm generates several mapping variants. The appropriate mapping variant that minimizes reconfiguration cost is selected as described in Section 4.3.4 and the following.

In order to describe the algorithm consistently with the notation introduced in Chapter 3, we assume that the synthesized netlist of task $i \in \mathcal{N}_\mathsf{T}$ is given as an input graph G_i. The set $\mathcal{N}_i$ of nodes n are the elements of that netlist. Finally, the synthesized netlist is mapped to the device specific netlist, i.e. in the terminology of Chapter 3 the input graph G_i is mapped to the image graph G_i' with a set $\mathcal{N}_i'$ of nodes, which are the LEs of the device specific netlist.

The algorithm generates a set $\mathcal{V}_i$ of LEs v. The set $\mathcal{N}_i'$ of nodes in the image graph G_i' will be a subset of $\mathcal{V}_i$ and contains those LEs, which resemble the chosen mapping variants. There may be several nodes n mapped to an LE v. However, it is ensured that v contains only nodes that are connected in the input graph. The algorithm ensures the routability of all nodes n mapped to v.

The set $\mathcal{R}_n$ of LE resources r_n denotes all mapping variants of a node n.

Algorithm 6 works as follows. First, for each $n \in \mathcal{N}_i$ (line 1) the mapping variants $\mathcal{R}_n$ (line 3) are investigated with the aim of producing a series of valid LEs v that contain node n. Two lists (line 4) and a queue (line 5) are introduced for controlling the following operations. The mapping procedure starts from one mapping version $r_n \in \mathcal{R}_n$ of node $n \in \mathcal{N}_i$, where n is called initial node. For this case, the edif_queue contains all nodes $n_k \in \mathcal{N}_i$ connected to the initial node n.

For each element n_k in the edif_queue, all possible mapping variants $\mathcal{R}_{n_k}$ of the node n_k to the resources inside the current LE v are investigated in sequential (line 10), until a valid mapping for each node is found.

Condition (line 11) determines which mappings $r_{n_k} \in \mathcal{R}_{n_k}$ for n_k are investigated: For the initial node n (i.e. $n_k = n$), only the mapping $r_n = r_{n_k}$ is allowed; For all other nodes (i.e. $n_k \neq n$), any mapping to the same type of LE as r_n can be

chosen. A mapping is valid, if the LE has free resources to map the node according to r_{n_k} and if the mapping leads to any new connection *inside* the LE (line 14). Condition (line 14) also skips the local route condition for the initial node n.

After this mapping procedure the LE v is generated. If the LE v is routable, it is stored temporarily in routable_list. In line 24, we add all nodes that are connected to n_k to the back of edif_queue. If one node of the edif_queue could not be mapped to resources of the current LE, it is inserted into black_list (line 26), a list of nodes that will not be inserted into edif_queue again.

If loop (line 7) is finished, the first and last element of routable_list are added to the set $\mathcal{V}_i$ of generated LEs. Hence we obtain two different LEs for each mapping variant r_n of the initial node: one LE has a minimal number of nodes mapped to and another one has a maximum number of nodes mapped to the LE, with respect to the greedy packing algorithm. The algorithm ensures that all connections between the resources inside the LE can be routed locally, connections that run between different LEs are routed later, during the place and route step.

This procedure is repeated for all elements $n \in \mathcal{N}_i$ and their corresponding mapping variants $r_n \in \mathcal{R}_n$.

The result of the mapping algorithm can be summarized in a relation $\mathcal{L}_i$ for task i, where the elements $(n, v) \in \mathcal{L}_i$ describe the mapping of element $n \in \mathcal{N}_i$ to the generated LE $v \in \mathcal{V}_i$.

Algorithm 6 Pseudo code for the mapping algorithm

```
1:  for all n ∈ N_i do
2:    obtain mapping variants R_n of node n
3:    for all r_n ∈ R_n do
4:      initialize empty black_list, routable_list
5:      initialize edif_queue with n as initial node
6:      create empty LE v
7:      while edif_queue not empty do
8:        n_k := front( edif_queue)
9:        obtain mapping variants R_{n_k} of node n_k
10:       for all r_{n_k} ∈ R_{n_k} do
11:         if (((n = n_k and r_n = r_{n_k}) or n! = n_k) and LE_type( r_n) = LE_type(
            r_{n_k})) then
12:           v' := v
13:           ok = map n_k to v' according to the rules r_{n_k}
14:           if ( ok and n_k adds new local routes) or n = n_k then
15:             v := v'
16:             if v routable then
17:               add v to routable_list
18:             end if
19:             exit loop 10
```

```
20:            end if
21:          end if
22:        end for
23:        if n_k was mapped to v then
24:          add edif instances connected with n_k to back of edif_queue
25:        else
26:          add n_k to black_list
27:        end if
28:        remove n_k from edif_queue
29:        add front( routable_list) and back( routable_list) to V_i
30:      end while
31:    end for
32: end for
```

The presented algorithm generates all mapping variants for each initial node and ensures routability of the generated LEs by packing all nodes that result in local connections. However, the logic packing is only performed in a greedy manner, hence there is only one feasible mapping variant investigated for each node connected with the initial node. This reduces the amount of LEs generated, but results still in a variety of different LEs for each node. Note that this algorithm does not perform packing of unconnected nodes, this separate problem is discussed e.g. in [57].

4.3.3 Logic Element Selection

The mapping algorithm yields a number of mapping variants for each element in the synthesized netlist. However, it is required that every element of the synthesized netlist is mapped only once to the device specific netlist. Thus there exists an allocation $\mathsf{a} : \mathcal{N}_i \mapsto \mathcal{N}_i'$ with $\mathcal{N}_i' \subset \mathcal{V}_i$. Here, we describe an integer linear program (ILP) that can be used to choose a set $\mathcal{N}_i'$ of LEs. The ILP is extended in Section 4.3.4 to select optimal LEs for reconfiguration.

The ILP formulation is straightforward. For each LE $v \in \mathcal{V}_i$ we define a binary variable S_v that is 1 if the LE v is contained in $\mathcal{N}_i'$, or 0 if not. To select exactly one LE v for each node $n \in \mathcal{N}_i$ the following constraint is used:

$$1 = \sum_{(n,v)\in\mathcal{L}_i} S_v. \tag{4.5}$$

The constraint considers all LEs that contain the node n. The sum over all S_v represents the number of selected LE. An ILP that minimizes the amount of LEs used in a task i can be written as follows:

Program 1 Select a minimal set of LEs

minimize:

$$\sum_{v \in \mathcal{V}_i} S_v \tag{4.6}$$

subject to:

$$\forall n \in \mathcal{N}_i : \quad 1 = \sum_{(n,v) \in \mathcal{L}_i} S_v \tag{4.7}$$

where:

$$S_v \in \{0, 1\}. \tag{4.8}$$

4.3.4 Logic Element Selection for Minimal Routing Reconfiguration

In this section we describe a method to select a valid set LEs in order to optimize the reconfiguration time for interconnect. The objective is to reduce the number of reconfigurable connections to be routed in the FPGA, because these are the major cost factor in fine grained architectures. At first we derive a cost function for routing reconfiguration similar to Section 3.4.3. The mapping of elements from the synthesized netlist to the device specific netlist requires some modifications to that model, because the interconnect is mapped to both LE local connections and to connections realized in FPGA's routing resources. We further analyze, how the allocation information derived with the tool presented in Section 4.2, is incorporated into the allocation of device specific netlist elements. Finally we present an ILP that selects the LEs as required.

Reconfiguration Time for Interconnect after Mapping

It has already been discussed in Section 4.1.2, how the connectivity of the original synthesized netlist is altered during the mapping process. Here, the transformation is quantified to derive a cost function for the optimization that is consistent with the reconfiguration cost model presented in Chapter 3. In the following, we distinguish connections from the synthesized netlist by *net edges* (i.e. the edges $\mathcal{E}_i$ in the input graphs) and connections from the device specific netlist as *route edges*. The net edges of a task i can occur as route edges in the device specific netlist as follows:

- as local connections inside an LE, where the number of local connections is denoted as $E_{\mathrm{L},i}$,
- as connections between LEs ($\mathcal{E}_i'$), and

- as multiple net edges that are merged into one route edge, where the number of such edges is denoted as $E_{\mathsf{M},i}$.

Hence, for a task i the number $|\mathcal{E}_i|$ of net edges in the synthesis netlist translates to:

$$|\mathcal{E}_i| = |\mathcal{E}'_i| + E_{\mathsf{L},i} + E_{\mathsf{M},i}. \tag{4.9}$$

The number $|\mathcal{E}_i|$ of net edges is constant, while the number of local connections, connections between LEs and merged connections depends on the selected LEs for task i. Both type of connections, local and external may be reconfigured at runtime. We noted that the reconfiguration external connections is much more expensive because more configuration data must be used to reconfigure the routing resources in the FPGA. Therefore, one objective of the LE selection is to reduce the reconfigurable external connections in a device specific netlist.

In the following we derive the cost for the reconfiguration of interconnect analog to Section 3.4.3. The reconfiguration cost between two tasks $i, j \in \mathcal{N}_\mathsf{T}$ are given by:

$$t_\mathcal{E}(i,j) = |\mathcal{E}'_i| + |\mathcal{E}'_j| - 2|\mathcal{E}'_i \cap \mathcal{E}'_j|. \tag{4.10}$$

In contrast to Equation 3.25 the mapping is not a one-to-one mapping here. With Equation 4.9 we receive:

$$E' = \sum_{i \in \mathcal{N}_\mathsf{T}} |\mathcal{E}'_i| = \sum_{i \in \mathcal{N}_\mathsf{T}} [|\mathcal{E}_i| - E_{\mathsf{L},i} - E_{\mathsf{M},i}] = \text{const.} \tag{4.11}$$

With the definitions above, a cost function similar to Equation 3.24 can be derived. This cost function is used to minimize the reconfiguration time between tasks that is induced by the reconfiguration of the routing configuration.

$$t_\mathcal{E} = 2(|\mathcal{N}_\mathsf{T}| - 1)E' - 2 \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j| \tag{4.12}$$

$$t_\mathcal{E} = 2(|\mathcal{N}_\mathsf{T}| - 1) \sum_{i \in \mathcal{N}_\mathsf{T}} [|\mathcal{E}_i| - E_{\mathsf{L},i} - E_{\mathsf{M},i}] - 2 \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j| \tag{4.13}$$

$$t_\mathcal{E} = 2(|\mathcal{N}_\mathsf{T}| - 1) \sum_{i \in \mathcal{N}_\mathsf{T}} |\mathcal{E}_i| - 2(|\mathcal{N}_\mathsf{T}| - 1) \sum_{i \in \mathcal{N}_\mathsf{T}} [E_{\mathsf{L},i} + E_{\mathsf{M},i}] - 2 \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j| \tag{4.14}$$

Observe that, in order to minimize routing reconfiguration time we can maximize the use of LE internal routing (i.e. $E_{\mathsf{L},i} + E_{\mathsf{M},i}$) and increase matching route edges between the tasks (i.e. $|\mathcal{E}'_i \cap \mathcal{E}'_j|$). Hence, in order to minimize the reconfiguration cost $t_\mathcal{E}$ we can maximize the term:

$$(|\mathcal{N}_\mathsf{T}| - 1) \sum_{i \in \mathcal{N}_\mathsf{T}} [E_{\mathsf{L},i} + E_{\mathsf{M},i}] + \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j|. \tag{4.15}$$

In the following we derive constraints that allow us to compute the terms of the cost function in Equation 4.15. The constraints are used to formulate an ILP that selects the LEs appropriately.

Computation of Local and Merged Interconnect

The terms $E_{\mathsf{L},i}$ and $E_{\mathsf{M},i}$ can be directly calculated from the LE selection for each task i:

$$E_{\mathsf{L},i} = \sum_{v \in \mathcal{V}_i} \mathsf{w}_{\mathsf{L}}(v) S_v \tag{4.16}$$

and

$$E_{\mathsf{M},i} = \sum_{v \in \mathcal{V}_i} \mathsf{w}_{\mathsf{M}}(v) S_v. \tag{4.17}$$

Where $\mathsf{w}_{\mathsf{L}}(v)$ is equal to the number of local connections inside an LE and $\mathsf{w}_{\mathsf{M}}(v)$ is the number of merged connections in the LE v.

Sustained Node Allocation

In Section 4.2 we have presented a tool that analyzes the synthesized netlists for similarity. The identified node allocation can be used as input to the mapping tool. However, the tool faces two problems due to the mapping to complex LEs. First, nodes that have been allocated to the same VA node may be mapped to different resources inside LEs, cf. Section 4.1.3. Second, there may be several nodes mapped to one LE (cf. Section 4.1.1), which in turn makes the similarity information ambiguous. Here we present a method that takes the similarity information from the synthesized netlists as a starting point. Using this information, the complexity of the reconfiguration cost optimization is reduced because not all possible allocations of LEs to a VA must be investigated. Here, we differentiate between the allocation derived for the synthesized netlists and the allocation of the device specific netlist to the VA model, which is referred to as *LE allocation*. Before we describe the method formally, we illustrate the situation with an example.

Example 4.9 *Figure 4.7 shows two device specific netlists. The nodes 1–9 from the synthesized netlist are mapped to 5 LEs. The nodes that should have been mapped to the same resource are connected by dashed lines. Because the LEs are generated independently of each other by Algorithm 6, the matching nodes are not necessarily mapped into a single LE in each task. I.e. in Netlist 2 the nodes 6, 9 are mapped to LE 'd', but the matching counterparts (2,3) in Netlist 1 are mapped to separate LEs 'A' and 'B'. However, after mapping the LEs can be allocated such that two LEs ('B','D') and ('C','E') in each netlist are matched. As a result one external connection matches in both tasks.*

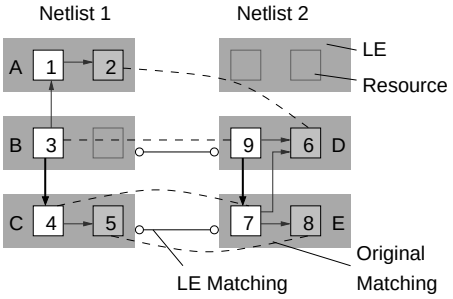


Figure 4.7: Illustration of how the matching information is effected by the mapping of nodes to LE resources.

It appears that the similarity information from the synthesis netlists can only serve as a hint in the reconfiguration cost optimization. In fact, the allocation of nodes is used to restrict the search for good candidates to of LE allocations. Instead of investigating all allocations of generated LEs, only the LEs that contain matching nodes considered. The ILP constraints that describe matching LEs are introduced in the following.

We assume that the allocation of nodes $\mathcal{N}_i$ is given as a set $\mathcal{A}$ of tuples $\mathbf{a} = (a_1, \ldots, a_{|\mathcal{N}_T|})$. Each element $a_i, i = 1, \ldots, |\mathcal{N}_T|$ of a tuple $\mathbf{a}$ is a node associated with a task $i \in \mathcal{N}_T$, i.e. $a_i \in \mathcal{N}_i$. It is assumed that all elements of $\mathbf{a}$ are allocated to the same resource in the VA in the context of the synthesized netlists. Obviously, each tuple $\mathbf{a}$ contains one node from each task, and each node belongs to only one tuple in $\mathcal{A}$. The allocation defined in $\mathcal{A}$ is used to define constraints in the ILP that can be used to compute an allocation for the LEs.

Now we define a set $\mathcal{A}'$ of possible LE allocations $\mathbf{a}' = (a'_1, \ldots, a'_{|\mathcal{N}_T|})$. Here, the element $a'_i, i = 1, \ldots, |\mathcal{N}_T|$ of $\mathbf{a}'$ is an LE associated with task i, i.e. $a'_i \in \mathcal{V}_i$. The possible allocations $\mathcal{A}'$ are induced by the relations $\mathcal{L}_i = \mathcal{A} \times \mathcal{A}'$ and the node allocations in $\mathcal{A}$. Thus for any LE $a'_i, i = 1, \ldots, |\mathcal{N}_T|$ in a possible LE allocation $\mathbf{a}' \in \mathcal{A}'$ there exists a mapping $(a_i, a'_i) \in \mathcal{L}_i$ where the nodes a_i belong to the same allocation $\mathbf{a} \in \mathcal{A}$.

Example 4.10 *This example is illustrated in Figure 4.8. Consider the allocation* $\mathcal{A} = \{(1,3,5),(2,4,6)\}$ *as an example. The elements may be mapped to LEs as follows:* $\mathcal{L}_1 = \{(1,a),(2,b)\}$, $\mathcal{L}_2 = \{(3,c),(3,d),(4,c)\}$, *and* $\mathcal{L}_3 = \{(5,e),(6,e)\}$. *The possible LE allocation* $\mathcal{A}'$ *induced by* $\mathcal{L}_1$, $\mathcal{L}_2$, *and* $\mathcal{L}_3$ *are* $\mathcal{A}' = \{(a,c,e),(a,d,e),(b,c,e)\}$.

From the set of possible allocations it is required to select a subset $\mathcal{A}''$ of feasible LE allocations, i.e. $\mathcal{A}'' \subset \mathcal{A}'$. The feasible allocations describe, which LE must be

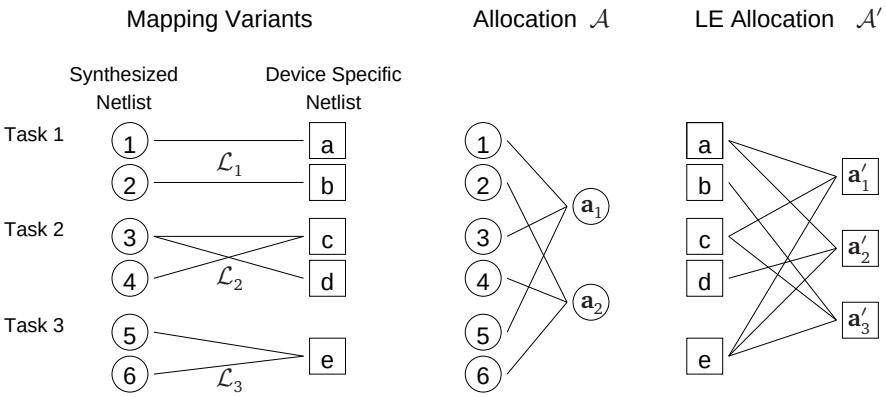


Figure 4.8: An example that illustrates mapping variants of the nodes from the synthesized netlist to device specific netlist. The example shows the possible allocations $\mathcal{A}'$ derived from the allocation $\mathcal{A}$.

placed on the same resource in the VA in order to obtain low reconfiguration cost. Therefore the subset $\mathcal{A}''$ must fullfill the same condition as $\mathcal{A}$: each LE is contained not more than once in any allocation $\mathbf{a}'' \in \mathcal{A}''$. With respect to the LE selection, each LE a''_i in an allocation $\mathbf{a}''$ must be selected, i.e. $a''_i \in \mathcal{N}'_i$. In order to formulate the conditions in an ILP, we introduce a binary variable $A_{a'}$ that is 1 if an allocation $\mathbf{a}' \in \mathcal{A}'$ is contained in $\mathcal{A}''$ and 0 if not. $A_{a'}$ can only be 1 if all related LE are selected:

$$A_{a'} \leq \bigwedge_{v=a'_1}^{a'_{|\mathcal{N}_\mathsf{T}|}} S_v \quad \text{where } \mathbf{a}' = (a'_1, \ldots, a'_{|\mathcal{N}_\mathsf{T}|}). \tag{4.18}$$

Additionally, every LE $v \in \mathcal{V}_i$ of a task i can only be contained in one LE allocation $\mathbf{a}'$, which is ensured by the constraint:

$$1 \geq \sum_{\forall \mathbf{a}' : a'_i = v} A_{a'}. \tag{4.19}$$

Matching of Route Edges

Finally we compute the term $|\mathcal{E}'_i \cap \mathcal{E}'_j|$ contained in Equation 4.15. The term describes the amount of matching edges for two tasks i and j. The matching edges are defined by the selected LE allocation $\mathcal{A}''$, but the contribution to the cost function is not yet computed explicitly. In the following, the computation of matching edges from node pair combinations is adopted.

Suppose there are two allocations $\mathbf{a}'_1 = (a'_{1,1}, \ldots, a'_{1,|\mathcal{N}_\mathsf{T}|})$ and $\mathbf{a}'_2 = (a'_{2,1}, \ldots, a'_{2,|\mathcal{N}_\mathsf{T}|})$. A node pair combination $\mathbf{n}$ is induced by the allocations. It is denoted as $\mathbf{n} = ((a'_{1,1}, a'_{2,1}), \ldots, (a'_{1,|\mathcal{N}_\mathsf{T}|}, a'_{2,|\mathcal{N}_\mathsf{T}|}))$. For such a node pair combination the reuse $\mathsf{w}(\mathbf{n})$ is given by Equation 3.37.

In Section 3.4.3 we have shown that the number E_U of matching edges between all tasks is equivalent to the reuse of edges in the VA:

$$E_\mathsf{U} = \sum_{i \in \mathcal{N}_\mathsf{T}} \sum_{\substack{j \in \mathcal{N}_\mathsf{T} \\ i \neq j}} |\mathcal{E}'_i \cap \mathcal{E}'_j| = \sum_{e \in \mathcal{E}_\mathsf{A}} \mathsf{u}^2(e). \tag{4.20}$$

Therefore we can conclude that the number E_U of matching edges between all tasks is equal to the reuse of edges induced by the node pair combinations. The reuse of edges has been calculated by assuming a small VA for each node pair combination. The overall reuse if given by the sum over all node pair combinations.

In the ILP the node pair combination does only contribute to the reuse if both associated allocations are selected, i.e. if $A_{a'_1} = 1$ and $A_{a'_2} = 1$. This condition is checked by the binary variable B_{a_1,a_2}, which is introduced for any node pair combination $\mathbf{a}'_1, \mathbf{a}'_2 \in \mathcal{A}'$:

$$B_{a_1,a_2} = A_{a'_1} \wedge A_{a'_2}. \tag{4.21}$$

Now the number of matching edges is given by:

$$E_\mathsf{U} = \sum_{\mathbf{a}'_1 \in \mathcal{A}'} \sum_{\mathbf{a}'_2 \in \mathcal{A}'} \mathsf{w}(\mathbf{n}) B_{a_1,a_2}. \tag{4.22}$$

The above equation includes all reused edges defined by the node pair combination $\mathbf{a}'_1, \mathbf{a}'_2$, which is enabled by the binary variable B_{a_1,a_2}.

The complete ILP is shown in Program 2. The ILP describes the selection of LE and the selection of LE allocations simultaneously. The objective is the minimization of the total reconfiguration time for the routing between LEs.

Instead of minimizing the term $t_\mathcal{E}$ directly, the ILP maximizes Equation 4.15. The logical *and* ($\wedge$) of the binary variables can be reformulated as linear constraints, which is not done here to maintain the readability of the ILP.

Program 2 Minimal Routing Reconfiguration Time

maximize:

$$(|\mathcal{N}_\mathsf{T}| - 1) \sum_{i \in \mathcal{N}_\mathsf{T}} [E_{\mathsf{L},i} + E_{\mathsf{M},i}] + E_\mathsf{U} \tag{4.23}$$

subject to:

$$\forall n \in \mathcal{N}_i: \qquad 1 = \sum_{\substack{n=n' \wedge \\ (n',v)\in\mathcal{L}_i}} S_v \tag{4.24}$$

$$\forall i \in \mathcal{N}_\mathsf{T}: \qquad E_{\mathsf{L},i} = \sum_{v\in\mathcal{V}_i} \mathsf{w}_\mathsf{L}(v) S_v \tag{4.25}$$

$$\forall i \in \mathcal{N}_\mathsf{T}: \qquad E_{\mathsf{M},i} = \sum_{v\in\mathcal{V}_i} \mathsf{w}_\mathsf{M}(v) S_v \tag{4.26}$$

$$\forall \mathbf{a}' \in \mathcal{A}': \qquad A_{a'} \leq \bigwedge_{v=a'_1}^{a'_{|\mathcal{N}_\mathsf{T}|}} S_v \quad \text{where } \mathbf{a}' = (a'_1, \ldots, a'_{|\mathcal{N}_\mathsf{T}|}) \tag{4.27}$$

$$\forall \mathbf{a}' \in \mathcal{A}': \qquad 1 \geq \sum_{\forall \mathbf{a}': a'_i = v} A_{a'} \tag{4.28}$$

$$\forall \mathbf{a}'_1, \mathbf{a}'_2 \in \mathcal{A}': \qquad B_{a_1,a_2} = A_{a'_1} \wedge A_{a'_2} \tag{4.29}$$

$$E_\mathsf{U} = \sum_{\mathbf{a}'_1\in\mathcal{A}'} \sum_{\mathbf{a}'_2\in\mathcal{A}'} \mathsf{w}(\mathbf{n}) B_{a_1,a_2} \tag{4.30}$$

$$\tag{4.31}$$

where:

$$E_{\mathsf{L},i}, E_{\mathsf{M},i}, E_\mathsf{U} \in \mathbb{N} \tag{4.32}$$

$$S_v, A_{a'}, B_{a_1,a_2} \in \{0,1\} \tag{4.33}$$

4.3.5 Experiments

We have conducted a number of experiments in order show the feasibility of our mapping approach. We run our mapping tool on a number of example tasks, cf. Tables 4.3 and 4.4. The tasks add8, sub8 contain an 8 bit add and subtract circuit with registered outputs, respectively. Tasks opb_add and opb_sub are reconfigurable IP cores of an RSOC, see [17]. The tasks int_trafo and motion_est are reconfigurable IP cores that perform integer transform and motion estimation for a video compression algorithm, cf. [74]. The tasks have been analyzed with our matching tool presented in Section 4.2.

At first, we discuss the data obtained for our mapping algorithm. In Table 4.3, the results show the number of nodes in the synthesis netlists and the number of generated LEs. On average, there are two valid mapping variants for each node. From the generated LEs a subset of LEs is chosen that are included in the final

Table 4.3: Number of generated ($|\mathcal{V}_i|$) and selected ($|\mathcal{N}_i'|$) LEs compared to the number of nodes ($|\mathcal{N}_i|$) in the synthesized netlists.

Task	$\lvert\mathcal{N}_i\rvert$	$\lvert\mathcal{V}_i\rvert$	$\lvert\mathcal{N}_i'\rvert$
add8	58	71	32
sub8	60	74	33
opb_add	300	614	262
opb_sub	300	614	262
int_trafo	932	1 629	555
motion_est	985	1 801	578

device specific netlist. The results show that about one third of the generated LEs is chosen by mapping tool.

Now, we describe the results of the LE selection obtained from the ILP solution. In Table 4.4 we show the similarity of the reconfigurable circuits. In this table, both the size of the synthesized netlists and the size of the device specific netlists are shown. The size of the synthesized netlists is determined as number of nodes (i.e. the number of resources used) and as the number of edges between nodes (i.e. the interconnects between resources). Similarly for the device specific netlist the number of nodes represents the number of LEs. It can be observed that the size of the device specific netlist is about 30 % smaller than the synthesized netlist.

The benchmark tasks presented here belong to the applications (1)–(3) as shown in Table 4.4. For each application we computed the reconfiguration cost $t_{\mathcal{E}}$ for interconnect for both, the synthesized netlists and the device specific netlists. For the synthesized netlists we first computed an optimal allocation in order to reduce reconfiguration cost. The results are given in Table 4.4. Here, we compare the results to the case, were no reuse of interconnect is assumed. It can be seen that the reconfiguration cost computed for the synthesized netlists can be reduced to 1.8 %, 27.7 %, and 23.1 % for the applications (1)–(3), respectively.

The allocation information was used by the mapping tool for an optimal selection of LEs and the computation of a cost optimized allocation for the device specific netlist. It can be observed that the interconnect complexity is reduced substantially. About one third of the connections from the synthesized netlist are realized as local connections inside the LEs, because there are several nodes assigned to most LEs. This reduces the requirements for interconnect reconfiguration in general. However, our mapping tool reduces the interconnect reconfiguration cost additionally by computing an optimized allocation during the selection of LEs. In the example, the reconfiguration cost for the device specific netlists are reduced to 4.4 %, 7.7 %, and 16.6 % for the applications (1)–(3), compared to the reconfiguration cost if no reuse of interconnect is assumed. It can be observed that the advantage of the opti-

Table 4.4: Results of the mapping tool compared to the similarity information from the synthesized netlists.

Application	Task	Synthesized Netlist $\lvert\mathcal{N}_i\rvert$	$\lvert\mathcal{E}_i\rvert$	$\lvert\mathcal{E}_1 \cap \mathcal{E}_2\rvert$	$t_\mathcal{E}$	Device Specific Netlist $\lvert\mathcal{N}'_i\rvert$	$\lvert\mathcal{E}'_i\rvert$	$\lvert\mathcal{E}'_1 \cap \mathcal{E}'_2\rvert$	$t_\mathcal{E}$
(1)	G_1: add8	57	109			32	46		
	G_2: sub8	58	111			33	46		
				108	8			44	8
				0	440†			0	184†
(2)	G_1: opb_add	300	969			262	687		
	G_2: opb_sub	300	969			262	687		
				701	1 072			634	212
				0	3 876†			0	2 748†
(3)	G_1: int_trafo	932	3 564			555	2 224		
	G_2: motion_est	985	3 715			578	2 224		
				2 799	3 362			1 854	1 480
				0	14 558†			0	8 896†

†The entries show the reconfiguration cost if no reuse of edges is assumed.

mized allocation is increased by the mapping tool. For example in Application (2), the reconfiguration cost are 27.7 % before and only 7.7 % after the mapping has been performed. This may serve as evidence that the mapping has a fundamental impact on the reconfiguration cost optimization.

Our experiments have shown that the mapping tool indeed shows the intended behaviour for real life synthesized netlists. The tool generates a number of mapping variants with Algorithm 6. The mapping variants include the LEs where several nodes are mapped into. The optimal LE selection described in ILP 2 retains the similarity information and provides an optimized allocation of LEs within the VA model. Furthermore we have shown that the similarity can be increased substantially by the mapping tool.

4.4 Summary

In this section we have applied the reconfiguration cost model and the virtual architecture model that have been introduced in Chapter 3 to the implementation flow for FPGAs. The starting point has been a set of synthesized netlists that should be mapped to an FPGA architecture with the aim of low interconnect reconfiguration cost. At the outset, we presented a detailed analysis of circuit transformations that occur during the mapping step. Based on this analysis we have implemented a tool that computes an optimized allocation for the nodes and edges specified in

the synthesized netlists. In the computation, the circuit transformations have been considered. Finally we have implemented a mapping tool that takes the allocation information as an input in order to perform an optimized mapping of the synthesized netlists to device specific netlists. The mapping tool maps all reconfigurable circuits of an application at once. Thus it is possible to select the best solution from several possible mapping variants and to compute the according allocation information that is required later in the implementation process.

Chapter 5

High-Level Synthesis for Reconfigurable Computing

Today's economic requirements of short development cycles together with the increasing complexity of electronic systems call for efficient design methods. *High-level synthesis*[1] (HLS) is an automated design process to generate digital systems from behavioural descriptions [91][60]. The behavioural descriptions are provided for example as SystemC or ANSI-C/C++ source code. HLS tools compile the behavioural descriptions into *register transfer level* (RTL) code that is implemented with traditional logic synthesis tools.

In general, RTL code describes the dataflow between registers and logical operations and the control of the dataflow. RTL descriptions describe a timed behaviour, i.e. it is specified for each control step, which operations from the behavioural descriptions are executed. The allocation of data to registers and of operations to computational resources is fixed. However, resources can be shared by different operations.

In contrast to RTL descriptions, a behavioural description does not model the circuit behaviour at this level of detail. Instead, behavioural descriptions model the algorithm as an untimed sequence of operations. An HLS tool translates the behavioural description into a digital circuit that performs exactly the function specified. The HLS tool decides, based on user constrains, in which cycle and on which resource the operations are computed, in which memory elements the results are stored, and how the control of the dataflow is realized. Apparently, the HLS tool operates on a large design space from where the tool aims to choose the best feasible solution. The design space consists of different resource types on which operations are computed, the available device area to instantiate those resources, and the order in which the operations can be executed. Different solutions in the design space can result in implementations with different structures and different control sequences.

[1] Also: C-synthesis, electronic system level (ESL) synthesis, algorithmic synthesis, or behavioural synthesis

The motivation for using HLS for RSOC design arises from the large design space. In Chapter 4 we considered fixed, synthesized netlists of each configuration. The netlists were produced by an RTL synthesis tool, which usually optimizes the synthesis result only in terms of resource usage and circuit delay. Hence, the principal similarity of the netlists and the resulting reconfiguration costs could only be influenced by the designer, but not by the synthesis tools. Therefore the designer is required to write reconfiguration-optimized code in order to enhance the netlist similarity.

With HLS the situation is completely different: the HLS tool has the possibility to choose solutions from the design space that form the desired trade off between resource usage, execution latency, circuit delay, and reconfiguration costs. This is possible because the untimed behavioural description is transformed fundamentally into a corresponding datapath during HLS synthesis.

In this chapter we describe new methods for HLS that are used to generate implementations, which are optimized for dynamic reconfiguration. We have developed a complete HLS tool that implements these methods [78]. Additionally we introduce several other approaches that exploit the versatile behavioural descriptions to realize runtime reconfiguration of tasks with reduced overhead.

This chapter is structured as follows: In Section 5.1 we describe the HLS tool flow that is used to generate the reconfigurable modules. Next, the architecture of the reconfigurable modules and the execution of a task in such a module is explained. This information is fundamental to the introduction of new concepts for runtime reconfiguration in Section 5.2. In these concepts we illustrate how hardware tasks can be realized more efficiently in terms of resource usage and reconfiguration cost. We propose the implementation of multiple tasks in one reconfigurable module in Section 5.2.1 and the use of dynamic reconfiguration for sub-blocks of the reconfigurable module in Section 5.2.2. With these concepts reconfiguration is considered at system design level. In Section 5.3, we briefly review established HLS models. We employ established methods to model the resource binding and scheduling of a task. However, the models are extended such that our virtual architecture model is incorporated in order to assess the reconfiguration cost during resource binding. We describe how the reconfiguration cost and the implementation cost are computed within the virtual architecture/HLS model. In Section 5.4 we propose several optimization strategies for implementation of datapaths that are optimized in terms of runtime reconfiguration cost. The optimization methods are based on the methods introduced Chapter 3. Finally we present a series of experiments in Section 5.5, which demonstrate the performance of our methods. Based on the results we discuss the performance compared to traditional approaches and the implications of the proposed reconfiguration concepts presented in Section 5.2.

5.1 Introduction to HLS

In this section we present an overview of the synthesis steps that are performed by our HLS tool. We further describe the resulting architecture of the reconfigurable modules and the execution model of the hardware tasks on such a module..

5.1.1 HLS Tool Flow

The HLS steps of our tool are depicted in Figure 5.1. The tool flow shows the processing steps that are applied to the behavioural description given as C-Code. The result of the processing are the reconfigurable modules.

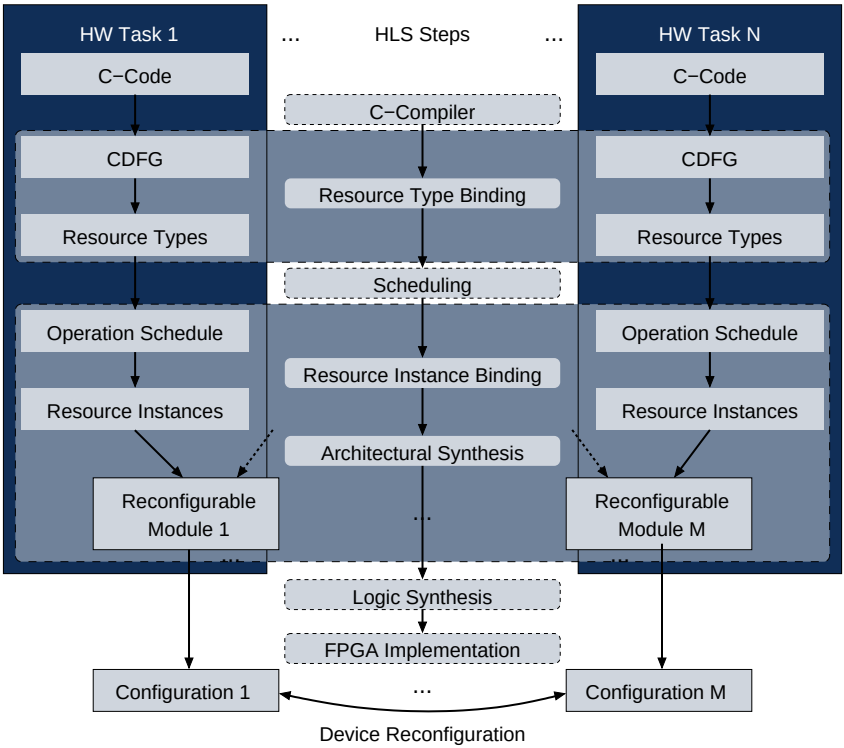


Figure 5.1: The HLS tool flow.

Here we assume that the designer has selected the hardware tasks (HW Task 1...N) for the implementation as reconfigurable modules in the RSOC. For our tool

flow we have extended the lcc ANSI-C compiler [31][30] such that it compiles a C-function into a control dataflow graph (CDFG). The compiler has been modified such that instruction level parallelism can be exploited by the HLS tool. Each HW task is compiled separately.

In the next processing steps, the CDFG is processed until the reconfigurable modules can be created. Therefore information on how to execute the operations contained in the CDFG is gathered.

At first, a resource type is selected for each operation in the CDFGs. The resource types are provided in a library. The library contains several resource types for arithmetic and logical operations, and for special hardware functions. The resource types are annotated with resource requirements and timing information. The resource types are selected for all CDFGs in one step in order to allow an optimization of reconfiguration cost, cf. Section 5.4.2. With the timing information and resource requirements, a scheduling of operations in the CDFG is performed, for each task independently. Now, the operations are allocated to resource instances. This is performed for all hardware tasks at once, because there exists a huge optimization potential in terms of the resulting similarity in the implementation. The architectural synthesis step translates CDFG according to the collected information into an RTL description. The RTL description contains a datapath to realize the CDFG operations and a unit that controls the datapath.

The RTL description represents the starting point for existing logic synthesis tools and the FPGA implementation tools to generate a reconfigurable module and the configuration that implements the reconfigurable module. In Section 5.2 we discuss several concepts how the reconfigurable modules $1 \ldots M$ can be designed such that efficient runtime reconfiguration can be achieved.

As indicated in Figure 5.1, the reconfigurable modules can contain a datapath and control logic that enables the execution of multiple hardware tasks on that module. Therefore, architectural synthesis is performed such that the reconfigurable module contains all necessary resource instances and control functions.

In the following we describe the architecture and the operation of such a reconfigurable module.

5.1.2 Realization of the Hardware Tasks

The HLS determines all parameters that are required to describe the execution of a hardware task. However, in order to build an RTL description that can actually execute the hardware task an *architecture template* is needed. The architecture template is a concept how the reconfigurable module is built from the information provided by the HLS. The formal parameters that result from HLS describe the structure of the datapath and the control sequence that realizes the task control. The architecture template defines how the datapath is connected to the control unit and to the

other components in the RSOC, and how the control sequence is implemented in a state machine. The architecture template defines also how the RSOC can initiate the hardware task execution in the reconfigurable module and how to communicate with the hardware task. The architecture template further provides several possibilities to perform partial reconfiguration of sub-modules inside the reconfigurable module. Therefore, the device configuration can be adapted by reconfiguring parts of a reconfigurable module in order to adapt the functionality for execution other hardware tasks. The partial reconfiguration of the sub-modules is explained in Section 5.2.

In the following, we describe the architecture template and the execution scheme of a hardware task on reconfigurable module, which is based on the architecture template.

Architecture Template

The architecture template consists of a datapath unit, a control unit, a bus interface and optional i/o interfaces. The general structure is shown in Figure 5.2.

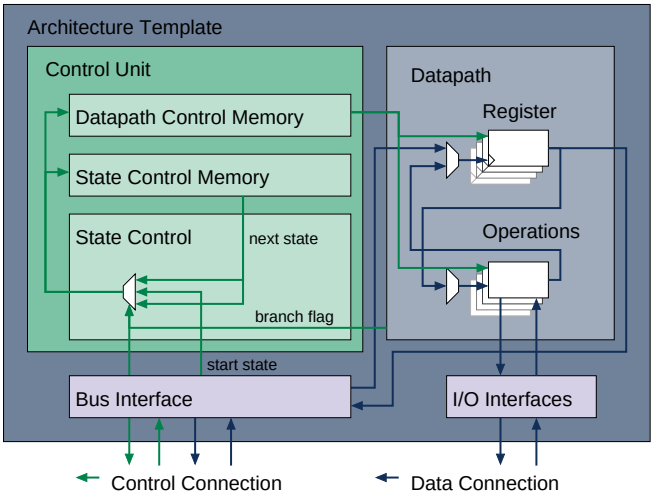


Figure 5.2: Architecture template for HLS

The control unit consists of a state control memory and a datapath control memory. In the control unit, a multiplexer selects the next state of the state machine according to the contents of the state control memory. The memory contains two alternative next states, one of those states is selected depending on the branch flag.

Alternatively, a required next state can be requested via the bus interface. Depending on the next state, the datapath control signals are driven from the datapath control memory.

The sequence stored in the state control memory can contain the control sequence for several tasks simultaneously. In addition, the state control memory contains special states that are used to exchange data between the RSOC and the datapath. Different tasks can be executed by forcing the control unit to the respective initial state of that task.

The state and datapath control memories are parametrized according to the requirements of the datapath and the control sequence. The parameters include the memory width, i.e. the size of a data word in the memory, and the memory depth, i.e. the number of data words stored in the memory.

The datapath unit contains registers as storage elements for variables, operations to compute data, and multiplexers to control the dataflow over the datapath connections. The control signals of these units are driven from the datapath control memory in the control unit. 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. Operations can also set the branch flag transferred to the control unit. Thus, a data dependent operation of the control unit is obtained.

The architecture template provides several different possibilities to adapt the module to a new task. The control memories and the datapath can be reconfigured as separate entities by dynamic reconfiguration. The memory contents could also be adapted by simple memory transfers from the system bus. The datapath is adapted at clock speed by the control signals from the datapath control memory. A more elaborate description of these methods is given in Section 5.2.

During architectural synthesis, the RTL description of a reconfigurable module is generated according to the presented architecture template. The datapath consists of resource instances from a macro library and the interconnect between the instances. The control memories are parametrized according to the requirements and filled with the control data. The reconfigurable module can be integrated easily into current FPGA SOC design tools, like e.g. Xilinx EDK.

Hardware Task Execution Scheme

In this section we describe the execution of a hardware task on a reconfigurable module that is based on the architecture template described above.

At first, the configuration associated with the reconfigurable module is loaded into the FPGA. Before the actual task is started, the initial parameters of the task are written over the system bus into the datapath registers. These parameters include the constant values and input variables used in the task. In order to initialize a

parameter the bus address selects a state from the state control memory. For this state, a datapath control word is selected. The control word activates the write function of the register where the parameter should be written to. The data of the parameter is supplied from the system data bus directly to the register input. Immediately after the write, the state control returns to an idle state.

After the initialization is performed, the initial state of the task is selected over the system address bus. The initial state is the entry point to the control sequence that controls the whole task execution. After the initial state is activated, the control of the task is performed by the state control, independently from the system bus. The task is finished when the state control returns to the idle state. The reconfigurable module contains a special register that holds a flag wether the task is running or idle. The system can monitor this register to determine when the task has finished.

The return values of the task are read through the system bus as well. The bus sets the desired read address and the state control selects the output of a register according to that address. The data is then transferred from that register to the data bus. Note that the state control has direct control over a multiplexer that connects several datapath registers to the data bus; the data read is not realized with special control states.

5.2 New Concepts for Task-based Reconfiguration

The high abstraction level of the behavioural description of a task provides a clear separation between the functionality and the realization of a task. While the functionality is specified by the designer, the realization is determined by the HLS tool. Therefore the HLS tool has the ability to take advantage of new concepts that are based on dynamic reconfiguration, without putting additional burden on the designer.

Here we introduce new concepts that allow a more efficient realization of tasks with reconfigurable modules. Efficiency here means high quality trade offs between resource usage of the reconfigurable modules and runtime reconfiguration cost. First, we propose the implementation of different tasks in one reconfigurable module, cf. Section 5.2.1. Second, we propose the use of dynamic reconfiguration for sub-blocks of the reconfigurable module (Section 5.2.2). Third, we want to highlight that partial reconfiguration and datapath control should be considered altogether when hardware tasks are realized.

The concepts extend the way dynamic reconfiguration is understood and used today.

5.2.1 Multiple Hardware Tasks in one Reconfigurable Module

Traditionally, dynamic reconfiguration is used to implement separate tasks as separate reconfigurable modules and hence, configurations. As already discussed, this can be inefficient in both resource usage and reconfiguration costs. In most reconfigurable systems, the amount of reconfigurable resources is fixed at design time. Naturally the reconfigurable resources must allow the largest among the tasks to fit into the reconfigurable area. The other tasks may leave some resources unused, which leads to internal fragmentation.

Later on in this chapter we will demonstrate that the tasks can re-use a large amount of datapath resources. We show that it is feasible to realize several tasks in one reconfigurable module, often without excessive resource overhead. The reason is that resources can be shared between tasks at the expense of a small amount of extra resources to control the dataflow.

This method is perfect to use up previously unoccupied resources. Hence we can realize as many hardware tasks in a reconfigurable module until the resources are exhausted. At the same time, the number of different reconfigurable modules in an RSOC is reduced, which in turn reduces the number of reconfigurations. Thus we are able to use available resources more efficiently and reduce the number of reconfigurable modules.

The realization of tasks in one reconfigurable module is effected by knowledge on the expected execution order of the tasks. Mainly those tasks should be merged into one module, that are most frequently reconfigured. In this case the frequency and time required for dynamic reconfiguration can be reduced.

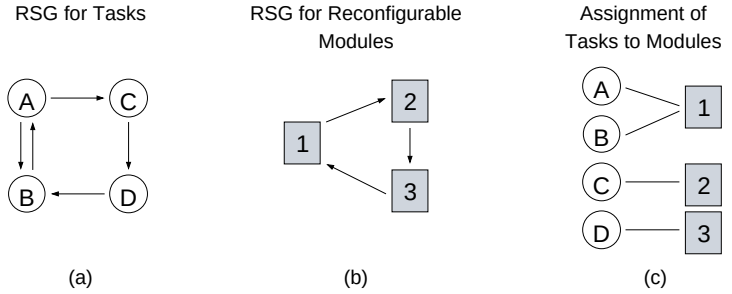


Figure 5.3: Realization of multiple hardware tasks in reconfigurable modules.

Example 5.1 *Consider the tasks shown in Figure 5.3. The RSG for the tasks A–D is shown in Figure 5.3(a). The RSG for the reconfigurable modules 1–3 is shown in*

Figure 5.3(b). The mapping of tasks to reconfigurable modules is illustrated as bipartite graph in Figure 5.3(c).

There are five possible transitions between tasks. It is assumed that the Tasks A and B can be realized in the same reconfigurable module (1) without the violation of resource constraints. Hence there is no dynamic reconfiguration necessary if either task is executed after the other.

Still dynamic reconfiguration must be used when e.g. Task C (realized in Module 2) is executed after Task D (realized in Module 3). For realization b), the number of dynamic reconfigurations is three, compared to five dynamic reconfigurations if each task would be mapped to an individual module.

5.2.2 Multi-Level Reconfiguration

Here we present a holistic view of reconfiguration. There is much dispute over the definition of the term *reconfiguration*. For FPGAs the term is often used only for techniques that change the configuration of the FPGA device. This view totally neglects the purpose of reconfiguration, i.e. the runtime adaptation of a digital circuit to the required data processing. In this section we want to discuss several techniques to adapt a reconfigurable module at runtime under the notion of *multi-level reconfiguration*.

The techniques considered for multi-level reconfiguration include:

- dynamic reconfiguration of LE resources (e.g. LUTs),
- dynamic reconfiguration of sub-modules (e.g. datapath and control memories), and
- runtime control of the datapath.

With our HLS approach we are able to generate reconfigurable modules that use all aforementioned reconfiguration techniques. In this chapter we want to investigate, which techniques are efficient in which scenarios. We expect that several techniques must be applied in order to obtain an efficient RSOC implementation on FPGAs.

In the following we discuss the reconfiguration techniques in detail. At first we show how the dynamic reconfiguration of LE resources can be exploited to implement resource efficient datapaths that can be adapted with little reconfiguration data. Second, the dynamic reconfiguration of sub-modules of a reconfigurable module is explained. We point out that the control memory contents can be adapted by dynamic reconfiguration or via the RSOC system bus. Third, we consider the runtime control of the datapath as 'reconfiguration' of the datapath functionality. At last, we compare the differences between the dynamic reconfiguration of LE resources and the runtime control on an example datapath operation.

Dynamic Reconfiguration of LE Resources

We have already shown in previous chapters, that for general reconfigurable circuits, parts of the logic and parts of the connectivity can be reused in several configurations. Because the reconfiguration of connectivity is very expensive we want to reconfigure parts of the logic only, i.e. individual LE resources.

In many island style FPGAs the logic is structured such that it can be configured for several different functions, while the connectivity over the routing resources remains the same. We have explored this manually e.g. in [81]. If word level ALU operations are considered, it becomes obvious that only a fraction of the reconfigurability of the logic and routing resources are needed. In fact, many ALU operations can be realized by a reconfiguration of the LE only. The connectivity to other logic in the design remains static.

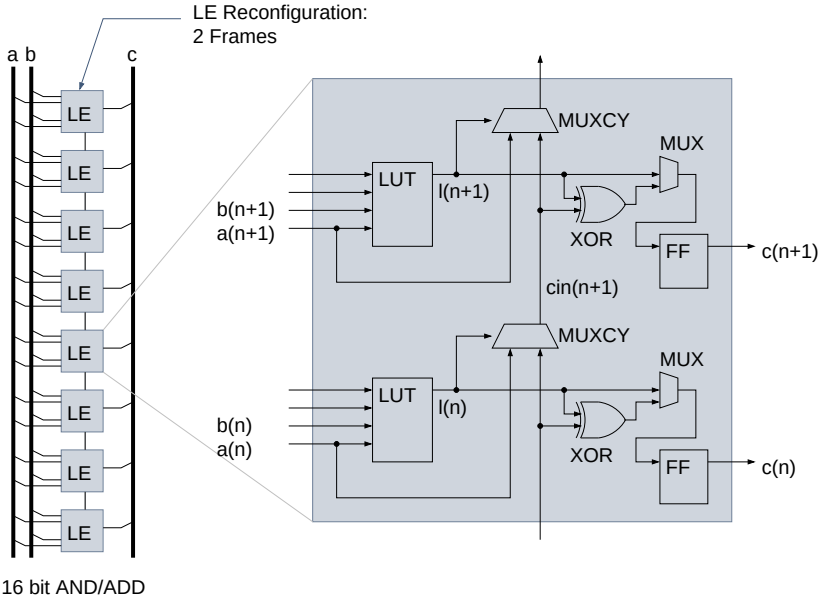


Figure 5.4: Realization of a reconfigurable ADD/AND operation with VirtexII LEs .

Example 5.2 *In this example we illustrate how the reconfiguration of the LE resources can change the ALU operation that is realized by the LEs. Suppose the resource instance implemented on the LEs must be reconfigured between an AND ($c = a$ and b) and an ADD ($c = a + b$) operation. The logic block in Figure 5.4 shows the relevant logic functions.*

Table 5.1: LE configuration to realize either an AND or an ADD operation.

LE Element	AND	ADD	Minor Frame Address†
LUT	$a(n)$ and $b(n)$	$a(n)$ xor $b(n)$	1
MUX	LUT out	XOR out	0

† if the logic blocks are placed on even numbered slice columns.

The AND operation is composed of the AND operation of the individual bits n of a, b, hence each LUT in the LEs realizes $c(n) = a(n)$ and $b(n)$. *The LUT output is propagated through the MUX element directly to the output flipflop FF.*

Conversely, the ADD operation employs the carry chain to propagate the carry bit from one partial result to the next. A 1 bit full adder is realized as follows: the LUT computes $l(n) = a(n)$ xor $b(n)$. *The MUXCY select signal is controlled by* $l(n)$ *and propagates either the carry input* ($cin(n)$) *to the next bit or generates a carry bit if necessary from* $a(n)$. *The adder result is computed by the XOR resource:* $c(n) = l(n)$xor$cin(n)$.

The difference in the LE configuration is summarized in Table 5.1. Note that the use of the carry chain enforces that only logic blocks in the same column are allocated by resource instance. Hence, the whole operation can be reconfigured with two configuration frames only. The connection to the routing structures and the placement of the ADD and AND operation remains static and does not require reconfiguration.

In the macro library of the HLS tool, a reconfigurable resource type may be introduced that is capable to realize many ALU functions at the same time. At runtime, the instances of this resource type are frequently reconfigured to adapt the associated LEs to the required functions in a certain control state.

Dynamic Reconfiguration of Sub-Modules

Using partial dynamic reconfiguration, the implemented digital logic can be changed completely. However, the reconfiguration costs are high, because the logic as well as the connectivity of the circuit is reconfigured. The reconfigurable modules contains the control until and the datapath as separate sub-modules, hence they can be reconfigured independently.

The control memories can be reconfigured in two domains: (1) The memory content may be reconfigured to accommodate the control data for different tasks. Thus, the control data must not be present for all tasks at once, which saves valuable on-chip memory. In dynamically reconfigurable architectures, the memory contents can be updated with partial dynamic reconfiguration or via a bus that is connected to the RSOC system bus. (2) In the case of very tight resource restrictions it might also be necessary to adapt the memory layout, i.e. the width and the depth of the

control memories. The depth of the state control memory depends on the number of states that are required in the control sequence. The memory width depends on the address width of both control memories. The depth of the datapath control memory depends on the number of different datapath control words. The data width depends on the number of control signals that are used for a datapath. It becomes apparent that both memory layouts are very application dependent and dynamic reconfiguration of the memory layout can provide benefits in terms of task dependent resource utilization. An alternative implementation of the datapath control is described in [80].

The tasks may be assigned to reconfigurable modules based on their similarity in the datapath. However, if a reconfigurable module is configured on the FPGA, then the datapath of that module can be exchanged by reconfiguration of the datapath sub-module. The dynamic reconfiguration of the datapath has the advantage that less unused datapath and control flow resources are needed to implement different tasks. Despite the differences in the datapaths there are resources that can be re-used. The HLS tools provide exact information, which operations, registers, datapath multiplexers and data connections can remain static and which must be reconfigured. If dynamic reconfiguration at these fine grain level can not be realized, the datapath can be reconfigured altogether.

Runtime Control of the Datapath

As already mentioned, the control of the datapath is usually not considered as reconfiguration in FPGAs, but here we interpret any change in the functionality of the datapath as reconfiguration, because it enables a consistent description of runtime reconfiguration over many levels and granularity. The main argument for using the datapath control to implement different tasks arises from the fact that in many datapaths, the control is required for a single task anyway. The datapath control of a single task results from the task control flow and from resource sharing.

Different sequences of dataflow control can be used to realize several tasks on the same datapath, if the datapath contains all necessary control and logic resources. The data processing is adapted by control data only. In this work we show that this technique re-uses large parts of dataflow control that is required for the execution of a single task anyway, hence the introduced overhead is small.

Note the fundamental difference between the dynamic reconfiguration of LE and the reconfiguration by dataflow control: the first uses the device reconfiguration mechanism to adapt parts of the circuit; the second adapts the circuit by control logic that uses FPGA resources. Dynamic reconfiguration uses the reconfiguration mechanism in the FPGA which has a limited reconfiguration speed; the reconfiguration by datapath control can be executed in a single clock cycle.

The architecture template provides a very simple mechanism to execute different tasks on the same datapath. The state control memory and the datapath control

memory is programmed to contain the data for multiple tasks to be executed. The RSOC initiates the tasks simply by putting the control unit in the correct initial state.

Dynamic reconfiguration and reconfiguration by dataflow control can be combined to achieve the most efficient implementation of an application.

Comparison Between Datapath Control and Reconfiguration of LEs

It has already been discussed that the function of a datapath resource can be controlled by control signals generated in the control unit or by dynamic reconfiguration of the associated LE resources. Control signals can only be used for LE resources where the control is connected to the routing resources. However, many multiplexers inside the LEs are only controlled by the device configuration. Therefore, datapath resources that employ control signals are implemented less resource efficient, compared to device reconfigurable datapath resources. This is illustrated in Example 5.3. However, device reconfigurable datapath resources can not be adapted in a single clock cycle.

Example 5.3 *Here we compare the implementation of two reconfigurable resources. In one resource, the operation is selected by a control signal and in the other resource the operation is adapted by device reconfiguration. Both resources perform two types of operations: an AND and an ADD operation. For the ADD operation, the fast carry path in the VirtexII LE shall be used. The LE configuration for both resources is shown in Figure 5.5. For illustration, only the logic for the partial result* $c(n) = \mathrm{f}(a(n), b(n))$ *is shown.*

The configurations of the device reconfigurable resource are shown in Figures 5.5(a) and 5.5(b). Figure 5.5(a) depicts the LE configuration for the ADD operation. In the LE, the LUT is used to implement and XOR function. The MUXCY and XOR elements are used to realize a carry chain. The MUX is configured to pass the XOR output to the result $c(n)$*. The configuration of the LE for the AND operation is shown in Figure 5.5(b). The LE configuration differs in the configuration of the LUT that realizes the logical AND of* $a(n)$ *and* $b(n)$*, and in the configuration of the MUX that passes the LUT output to the result* $c(n)$*. The configuration of the LUT and the MUX can not be controlled from the FPGA fabric, instead 2 configuration frames must be written in order to reconfigure the functionality, cf. Example 5.2.*

As an alternative, both functions can be implemented in a datapath resource, where the function is selected by a control signal d*, cf. Figure 5.5(c). The ADD functionality is realized in the left LE, similar to Figure 5.5(a). In the right logic block, the LUT is used to realize the AND function of* $a(n)$ *and* $b(n)$ *and to pass through the result* $c'(n)$ *of the adder, depending on the control signal* d*. Hence, the LUT function is e.g.* $FCN = c'(n)\overline{d} + a(n)b(n)d$*. Observe that the control over the type of operation is achieved by LUT logic now, not by a reconfiguration of the MUX element.*

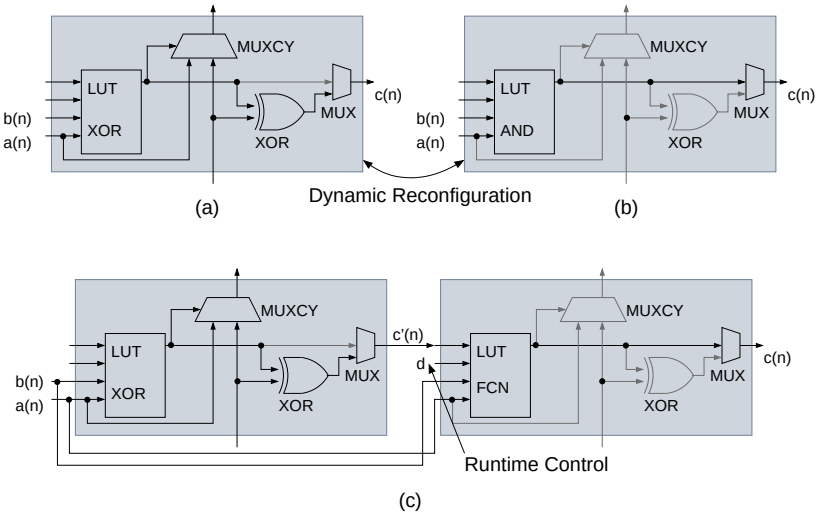


Figure 5.5: VirtexII LE configuration (simplified) for a single bit of operation ADD (a), AND (b), and selectable ADD/AND (c). Active LE resources are drawn black.

The example illustrates the trade-off involved when using reconfigurable resources. The implemention requires only half the resources of a resource with a control signal and also has a shorter propagation delay, because no LUT cascading is used. On the other hand, the resource may not be allocated by both types of operations within the same task, because dynamic reconfiguration within the execution of a task is not feasible.

5.2.3 Resource Sharing

Resource sharing is a technique that is frequently applied when DFGs are realized on a datapath. Resource sharing means that a resource instance in a datapath is used to execute more than one DFG node on this resource. As a result, less resource instances are required in the datapath and hence, the implementation is more resource efficient.

In this work we can distinguish different kinds of resource sharing. In *intra-task* resource sharing, several nodes of one DFG are executed on one resource instances. This is the classic application of resource sharing. If multiple tasks are realized in one reconfigurable module, then *inter-task* resource sharing can be exploited: the nodes of different DFGs are mapped to the same resource instances in such a datapath. Hence, the number of resource instances can be reduced further.

The resource instances that are shared by DFG nodes, where the DFGs are realized in different reconfigurable modules, are not considered as shared resources here. Instead, these resource instances are called *reused* resource instances, according to our RSG model.

5.3 Datapath Synthesis

In this section we review established models that are used in HLS. The established models are extended in an appropriate way in order to incorporate our reconfiguration cost model. We describe a graph-based model for the tasks in Section 5.3.1 and a model for the datapath resources in Section 5.3.2. The task model and the resource model are borrowed from earlier work on HLS, cf. Teich [91] and de Micheli [60]. In Section 5.3.3 we first review the established definition of resource binding. We further demonstrate how our virtual architecture (VA) model that has been derived in Section 3.4.2 (p. 62 et seq.) can be employed to describe the resource binding. With the definition of resource binding and the VA model we are able to introduce our implementation cost and a reconfiguration cost definition for HLS. The scheduling of the tasks is reproduced from previous works [91][60] in Section 5.3.4. Similarly, the constraints for scheduling and resource binding are summarized from [91][60] in Section 5.3.5. The scheduling and binding constraints are included for completeness and in order to motivate our method for resource binding presented in Section 5.4.3.

5.3.1 Task Model

Here we assume that each hardware task is specified as a graph. Similar to other work, we employ control dataflow graphs (CDFGs) here. A CDFG is a heterogeneous model that consists of a control flow graph (CFG) and a dataflow graph (DFG). The CFG models the control flow of a task, i.e. jumps and the conditional execution of *basic blocks* of computation. The DFG models the operations, data dependencies and precedence constraints inside a basic block. In our work, a modified ANSI-C compiler (lcc) is used to compile tasks specified as C-code into these CDFGs. In the following, more a elaborate description of the graphs is given.

The CFG describes the execution order of the basic blocks of a task. Since a formal definition of a CFG is not required in the remainder of this work, it will be omitted here. Instead, we describe the execution of a task using a CFG. The CFG consists of a begin node, an end node, and a node for each basic block. The nodes are connected by directed edges that indicate the execution order of the basic blocks associated with the nodes.

The task execution starts at the begin node of a CFG and follows the edge to the

first basic block. After all computations of the basic block are finished, the execution is passed to the next basic block, which is chosen according to the outgoing edge. The exit condition of a basic block (if one exists) determines, which basic block is executed next depending on the label of the outgoing edge. The task execution is finished if the control is passed from the last basic block to the end node.

The DFG is an labelled multidigraph (LMG), which has been introduced in Section 3.4.1 (Definition 3.3, p. 57). However, in our DFG the set $\mathcal{N}$ of nodes can be divided into two exclusive sets: a set $\mathcal{N}_O$ of operations and set $\mathcal{N}_V$ of variables. Similarly, the set $\mathcal{E}$ of edges can be divided into two exclusive sets: a set $\mathcal{E}_D$ of data dependencies and set $\mathcal{E}_C$ of precedence constraints.

An operation $n \in \mathcal{N}_O$ performs a computation with the input data supplied by incoming data dependencies. A variable $n' \in \mathcal{N}_V$ is used to provide and to store data.

The definition of an edge in an LMG has been illustrated in Figure 3.4, p. 58. A data dependency $e \in \mathcal{E}_D$ indicates a data transfer from the source node $\mathsf{s}(e)$ to the drain node $\mathsf{d}(e)$. The source label $\mathsf{l}_\mathsf{s}(e)$ indicates the result (esp. for multi-valued operations) that is transferred to the drain node. The drain label $\mathsf{l}_\mathsf{d}(e)$ indicates the argument of the operation referred to by the drain node. A precedence constraint $e' \in \mathcal{E}_C$ defines that the source operation $\mathsf{s}(e')$ must be finished before the drain operation $\mathsf{d}(e)$ can be started. Precedence constraints are defined in addition to data dependencies in order to realize a partial ordering of read and write operations on variables.

A basic block is described by a DFG. The purpose of the basic blocks is to separate control flow from dataflow. I.e. all operations in the DFG of a basic block are performed, if the basic block is executed. In addition to regular operations or variables, the DFG of a basic block contain one node that serves as entry point and another node that serves as exit point of the basic block.

The CFG defines a sequential execution order of the basic blocks. Thus, we can treat the operations in a basic block independently from other basic blocks. The only inter-dependency between the basic blocks is given by common variables. Hence, we have to ensure that common variables are mapped to the same resource instance for all basic blocks.

The CDFG model is illustrated in Example 5.4.

Example 5.4 *The CDFG model is illustrated on the CDFG specification of the Fibonacci algorithm, cf. Figure 5.6. The specification has been compiled with the modified lcc compiler. The source code is compiled into 3 basic blocks, labelled 1 to 3 in the figure. Block 2 represents the operations executed in the do-while loop. The block is always entered at node 6 and the block is finished when node 12 is executed. The subtraction at node 10 reads the variables 'n' and '1' as input and sends the output to variable 'n'. Additionally, there exists a data dependency between node 10 and node 11. The result computed at node 11 is passed to exit node 12. The result is used as a branch flag by*

the control unit to decide, whether the next basic block is started at node 6 (to continue the loop) or whether the next block is started at node 13. In block 2, the assignment of 'f2' must take place before the assignment of 'f'. Hence, the corresponding operations at nodes 8 and 9 are connected by a precedence constraint.

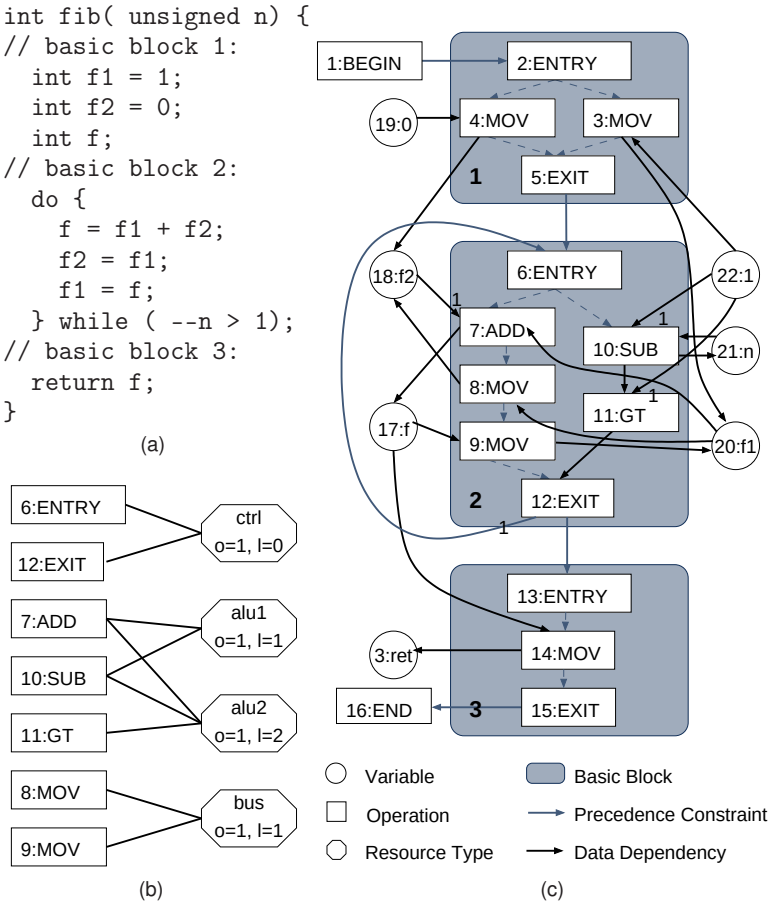


Figure 5.6: C-Source (a) and CDFG specification (c) of the Fibonacci algorithm. The resource graph of the CDFG, basic block 2 is shown in (b).

5.3.2 Resource Model

Resources can execute operations, store variables or provide a mechanism to access i/o interfaces. Operations include arithmetic and logic operations and i/o interfaces provide memory access or bus transfers. The resources are used to realize the functionality defined by the DFG in the datapath.

We differentiate between resource types $r_T \in \mathcal{R}_T$ and resource instances $r \in \mathcal{R}$. *Resource types* specify general characteristics of resources, e.g. the execution delay, the use of FPGA resources, or the DFG operations that can be executed. Many resource instances of the same resource type can exist. A *resource instance* is an entity that is realized as part of the datapath on FPGA resources. The number of resource instances may be restricted by the available FPGA resources, i.e. by the LEs in an FPGA. We continue to discuss properties that are associated with the resource types. The resource instances inherit the properties of the associated resource type.

A basic property of a resource type r_T is the execution *latency* l, where $\mathsf{l} : \mathcal{R}_T \mapsto \mathbb{N}$. For each resource type we assume an operation is started at clock cycle t and the execution of this operation is finished at clock cycle $t + \mathsf{l}$. When the execution is finished, the results of the operation (if any) are available. The results must be transferred to other operations depending on the data dependencies, or stored in an intermediate register. While the latency describes time delay of an operation, the *offset* o, where $\mathsf{o} : \mathcal{R}_T \mapsto \mathbb{N}$ describes the occupation of the resource itself. if the previous operation has been started at clock cycle t, then the resource can be used by the next operation at clock cycle $t + \mathsf{o}$. A resource is called a pipelined resource if $\mathsf{o} < \mathsf{l}$, i.e. new operations can be started before the result of the previous operation is available. For simplicity, the offset and latency is given globally for each resource type here.

Further parameters of resources are: the required device resources w_{LE}, the number of control signals w_{S}. The cost for dynamic reconfiguration a resource is assumed to be proportional to the required device resources.

5.3.3 Resource Binding

In this section we define how the nodes in the DFG are associated with datapath resources. We differentiate between the selection of a resource type that is used execute the operation and the selection of a resource instance on which the operation is actually executed. The selected resource type provides vital information for the scheduling of the operation. The mapping of operations and variables to resource instances in conjunction with the data dependencies and precedence constraints of the DFG yields the description of the complete datapath architecture.

The focus of this section is on resource binding for datapaths in reconfigurable modules. We describe the resource binding with respect to our VA model. This enables us to evaluate the reuse of resources between different reconfigurable modules

at a high abstraction level. The use of shared resources requires the introduction of dataflow multiplexers. We show how the size of the dataflow multiplexers can be derived from the VA. Based on our VA model we define the implementation cost and the reconfiguration cost of a datapath in a reconfigurable module.

Resource Type Binding

The resource type binding is a HLS step that assigns each node from the DFG to a resource type. The assignment is described by an allocation function a_T, where $\mathsf{a}_\mathsf{T} : \mathcal{N} \mapsto \mathcal{R}_\mathsf{T}$ assigns each node $n \in \mathcal{N}$ to a resource type $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$. Resource type binding imposes some (weak) constraints on the resulting datapath architecture. The binding defines, which operations are mapped to the same resource type, but not necessarily to the same resource instance.

The nodes of a DFG may be executed on different resources types from the HLS library. Thus, the resource type binding is not a one-to-one mapping. In addition, there exist different variants to map a node to the same resource type, if the node describes a commutative operation. The allocation a_T realizes only one mapping from several possibilities. In order to express all possibilities to map operations to resource types, a resource graph is defined as follows (cf. Teich [91], p. 84):

Definition 5.1 *For a given set $\mathcal{N}$ of DFG nodes and a set $\mathcal{R}_\mathsf{T}$ of resource types, the* resource graph *$\mathcal{G}_\mathsf{R}(\mathcal{N}, \mathcal{R}_\mathsf{T}, \mathcal{E}_\mathsf{R})$ is defined as a bipartite graph with a set $\mathcal{E}_\mathsf{R}$ of edges. The resource graph is a multigraph because there exist different possibilities k to map an operation to a resource type. An operation (or variable) $n \in \mathcal{N}$ can be executed (or stored) on an instance of the resource type $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$, if an edge $(n, r_\mathsf{T})_k \in \mathcal{E}_\mathsf{R}$ exists.*

The resource graph is illustrated in Example 5.5.

Example 5.5 *In Figure 5.6(b) a resource graph is shown. The operations 6, 12 can be executed on resource type ctrl and operation 8, 9 on type bus. The operations 7, 10 can be bound to either resource type alu1 or alu2 and Node 11 only to resource type alu2 . alu2 is a pipelined resource type: a resource instance can be allocated at every clock cycle ($o = 1$), but the results take 2 clock cycles to be computed ($l = 2$).*

Resource Instance Binding

Resource instance binding is performed by mapping each node $n \in \mathcal{N}$ from the DFG to a resource instance $r \in \mathcal{R}$. The allocation $\mathsf{a} : \mathcal{N} \mapsto \mathcal{R}$ describes such a mapping (cf. Section 3.4.1, p. 59). With the allocation, the actual datapath architecture is derived. The allocation of nodes also describes the mapping of data dependencies to connections in the datapath. The data transfers indicated by the data dependency are realized on the connections.

In many applications, the number of resource instances of a resource type is constrained. The restriction often reflects the memory or logic constraints or system design considerations. The number of available resource instances given by the function $\mathsf{b} : \mathcal{R}_\mathsf{T} \mapsto \mathbb{N}$, which assigns to each resource type $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$ the number of available resource instances. The binding to resource instances is restricted by the function b. There are only $\mathsf{b}(\mathsf{a}_\mathsf{T}(n))$ different allocations possible for each node $n \in \mathcal{N}$.

Embracing the VA Model into Resource Binding

In the following we show how the resource binding is used to map the DFGs to datapaths. Therefore the DFGs are considered as input graphs (cf. Section 3.4.1, p. 59). The DFG of a task $i \in \mathcal{N}_\mathsf{T}$ is denoted as an LMG $G_i(\mathcal{N}_i, \mathcal{E}_i, \ldots)$, which is called an input graph in the following. The input graph G_i of a task $i \in \mathcal{N}_\mathsf{T}$ is mapped to an LMG $G'_i(\mathcal{N}'_i, \mathcal{E}'_i, \ldots)$, which is called the image graph of G_i.

The resource type binding and the resource instance binding represent two different mappings of input graphs to image graphs. The mappings are highly interrelated. The mapping realized by the resource type binding yields image graphs that consist of types only. In contrast, the mapping realized by the resource instance binding yields image graphs that consist of resource instances and interconnect. This kind of image graph is a suitable model for a datapath in a reconfigurable module.

For the resource type binding, the input graph is mapped to an image graph G'_i, where the set $\mathcal{N}'_i$ of nodes contains all resource types that are used by the DFG. The employed resource types are a subset of all resource types $\mathcal{R}_\mathsf{T}$ provided in a synthesis library, i.e. $\mathcal{N}'_i \subset \mathcal{R}_\mathsf{T}$. The mapping of $\mathcal{N}_i$ to $\mathcal{N}'_i$ is given by the allocation a_T. The allocation of nodes also results in an allocation of edges (cf. Section 3.4.1, p. 60). Here, the function $\mathsf{a}_\mathsf{e} : \mathcal{E}_i \mapsto \mathcal{E}'_i$ allocates the set $\mathcal{E}_i$ of edges to the set $\mathcal{E}'_i$ of edges. The an edge $e' \in \mathcal{E}'_i$ denotes an *interconnect type*. The interconnect type describes, which ports of which resource types are connected with each other in the datapath, independently of the resource instance binding. For instance an edge $e' = (\mathrm{add}, \mathrm{sub}, \mathrm{o}, \mathrm{arg1})$ indicates an interconnect type, which describes that there exists a connection in the datapath between an instance of the resource type 'add' and an instance of the resource type 'sub'. The connection is established between the ports 'o' and 'arg1' of these resource instances.

The resource instance binding is modelled similarly. The input graph $G_i(\mathcal{N}_i, \mathcal{E}_i, \ldots)$ is mapped to an image graph $G'_i(\mathcal{N}'_i, \mathcal{E}'_i, \ldots)$. The allocation function $\mathsf{a} : \mathcal{N}_i \mapsto \mathcal{N}'_i$ maps the nodes $n \in \mathcal{N}_i$ of the DFG to the nodes $n' \in \mathcal{N}'_i$ (cf. Section 3.4.1, p. 59). Now, the set $\mathcal{N}'_i$ of nodes contains all resource instances that are used to realize the DFG on a datapath. The used resource instances are a subset of all provided resource instances $\mathcal{R}$, i.e. $\mathcal{N}'_i \subset \mathcal{R}$. The edges $\mathcal{E}'_i$ in the image graph comprise the

interconnect in the datapath that is required to realize the DFG. Again, the edge allocation a_e is performed according to the definition in Section 3.4.1, p. 60.

The mapping results of the resource type binding and of the resource instance binding differ in both, the nodes and the edges of the image graph. However, the relabelling of the port labels is the same in both cases, because the port labels depend only on the resource type, but not on the allocated resource instance.

Both, resource type binding and resource instance binding map the input graphs to image graphs. Here we employ our virtual architecture (VA) model presented in Section 3.4.2, p. 63. The VA model is used for two objectives: First, the reuse of resources and interconnect can be analyzed for different allocations. Second, it provides an efficient method to merge the resources and interconnect required for different tasks into a common datapath.

The VA graph $G_\mathsf{A}(\mathcal{N}_\mathsf{A}, \mathcal{E}_\mathsf{A}, \ldots)$ consists of a set $\mathcal{N}_\mathsf{A}$ of nodes and a set $\mathcal{E}_\mathsf{A}$ of edges amongst others. Here we discuss the resource instance binding in the context of the VA as an example. In this case, the nodes of the image graphs (and the nodes in the VA) represent resource instances in a datapath. The edges in the image graphs (and the edges in the VA) represent the interconnect in a datapath. For all tasks, the sets $\mathcal{N}_i, i \in \mathcal{N}_\mathsf{T}$ of nodes are mapped to the sets $\mathcal{N}'_i$ of resource instances. Similarly, the sets $\mathcal{E}_i, i \in \mathcal{N}_\mathsf{T}$ of edges are mapped to the sets $\mathcal{E}'_i$ of interconnect edges.

A valid VA $G_\mathsf{A}(\mathcal{N}_\mathsf{A}, \mathcal{E}_\mathsf{A}, \ldots)$ that contains the resource instances and the interconnect required by all tasks $i \in \mathcal{N}_\mathsf{T}$ is given in Equations 3.21 and 3.22 as follows:

$$\mathcal{N}_\mathsf{A} := \bigcup_{i \in \mathcal{N}_\mathsf{T}} \mathcal{N}'_i, \tag{5.1}$$

$$\mathcal{E}_\mathsf{A} := \bigcup_{i \in \mathcal{N}_\mathsf{T}} \mathcal{E}'_i. \tag{5.2}$$

The VA represents a super-datapath that is capable to realize DFGs of all tasks $i \in \mathcal{N}_\mathsf{T}$ without being reconfigured. The set $\mathcal{N}_\mathsf{A}$ of nodes in the VA model is equal to the set $\mathcal{R}$ of resource instances introduced in Section 5.3.2.

With the VA model, the reuse of resource instances and interconnect can be computed in order to evaluate the reconfiguration cost for reconfigurable modules. Here, we take into account the implementation of multiple tasks in one reconfigurable module. We define the set $\mathcal{N}_{\mathsf{T},m}$ of tasks, where all tasks $i \in \mathcal{N}_{\mathsf{T},m}$ are realized in the same reconfigurable module m. Each reconfigurable module realizes a subset of all hardware tasks in an application, i.e. $\mathcal{N}_{\mathsf{T},m} \subset \mathcal{N}_\mathsf{T}$. The datapath is only reconfigured if the task i to be executed is not realized in the reconfigurable module m that is currently active in the device, i.e. if $i \notin \mathcal{N}_{\mathsf{T},m}$.

For the computation of reconfiguration cost, the partial reconfiguration of between different reconfigurable modules, but not between different tasks, is considered. Hence, if a resource instance or interconnect is used in the datapaths of two

reconfigurable modules, then the resource instance or interconnect is not reconfigured between those two modules.

The requirements for the datapath of a reconfigurable module m are derived from the datapath requirements of all task i that are realized on the same reconfigurable module. All resource instances and interconnect defined in the image graphs G_i of the tasks $i \in \mathcal{N}_{\mathsf{T},m}$ must be contained in the datapath of the reconfigurable module m:

$$\mathcal{N}_{\mathsf{A},m} := \bigcup_{i \in \mathcal{N}_{\mathsf{T},m}} \mathcal{N}'_i, \tag{5.3}$$

$$\mathcal{E}_{\mathsf{A},m} := \bigcup_{i \in \mathcal{N}_{\mathsf{T},m}} \mathcal{E}'_i. \tag{5.4}$$

Where the set $\mathcal{N}_{\mathsf{A},m}$ of nodes represents all resource instances and the set $\mathcal{E}_{\mathsf{A},m}$ of edges represents all interconnect in the datapath of the reconfigurable module m.

The reuse of resource instances and interconnect is computed on the basis of the reconfigurable modules. Hence, if the same resource instance $n \in \mathcal{N}_{\mathsf{A}}$ is contained in y different reconfigurable modules, then the reuse $\mathsf{u}(n) := y$. For the partial reconfiguration of the datapath it does not matter, how often the resource instance n is used by any DFG that is realized in a reconfigurable module.

Dataflow Control and Interconnect

In the DFGs, a data dependency indicates the transfer of a single date between the operations. However in the datapath the data dependencies are realized as permanent connections between resource instances. The allocation function allows the mapping of several data dependencies onto the same connection. Although there may be several different resource instances that are connected to an input port of the same resource instance. In this case a dataflow multiplexer is required. With the multiplexer, the control unit can activate only the connection that is required in a specified control state.

We show how the number of inputs of the dataflow multiplexer can be determined from the graph representation of the datapath. Again, consider the set $\mathcal{E}_{\mathsf{A},m}$ of edges that represents the interconnect in the reconfigurable module m. A dataflow multiplexer has to control the data transfer over all edges $e \in \mathcal{E}_{\mathsf{A},m}$ that lead to the same input port $p := \mathsf{l_d}(e)$ of a resource instance $n := \mathsf{d}(e)$. In each reconfigurable module m, the number x of inputs of the dataflow multiplexer assigned to the input port p of resource instance n is given by:

$$\mathsf{x}(m,n,p) = |\{e \in \mathcal{E}_{\mathsf{A},m} : \mathsf{d}(e) = n \wedge \mathsf{l_d}(e) = p\}|. \tag{5.5}$$

Equation 5.5 allows to calculate the complexity of the dataflow multiplexers that are required for each reconfigurable module m. For each reconfigurable module m

we define the set $\mathcal{I}_m$, which contains pairs (n, p). In each pair, $n \in \mathcal{N}_{\mathsf{A},m}$ denotes a resource instance and p denotes an input port. The set $\mathcal{I}_m$ contains all possible pairs (n, p) that can be constructed for the set $\mathcal{N}_{\mathsf{A},m}$ of resource instances of a reconfigurable module m.

Implementation Costs

Here, we define the implementation cost of a datapath for a reconfigurable module m. The implementation cost describe, how many device resources are required to realize the datapath in the FPGA. The implementation cost are the combined requirements of LEs for both resource instances and dataflow multiplexers, and of interconnect resources. The implementation cost are calculated from the datapath definition $\mathcal{N}_{\mathsf{A},m}$, $\mathcal{E}_{\mathsf{A},m}$ (Equations 5.3 and 5.4) and the resulting dataflow multiplexers described by x in Equation 5.5.

During HLS only an estimation of the implementation cost is possible, because the datapath implementation costs depend on several optimization steps during logic synthesis and device mapping. The parameters used by the HLS estimation are based on a pre-synthesized resource library. Nevertheless, those estimates are necessary to optimize the resource instance binding in terms of implementation cost.

The implementation cost $C_{\mathsf{dp},m}$ for a datapath of the reconfigurable module m with the sets $\mathcal{N}_{\mathsf{A},m}$ and $\mathcal{E}_{\mathsf{A},m}$ is given by:

$$C_{\mathsf{dp},m} = C_{\mathsf{res},m} + C_{\mathsf{mux},m} + C_{\mathsf{wire},m}, \tag{5.6}$$

where $C_{\mathsf{res},m}$ represents the cost for resource instances, $C_{\mathsf{mux},m}$ represents the cost for dataflow multiplexers, and $C_{\mathsf{wire},m}$ represents the cost for the interconnect.

The cost $C_{\mathsf{res},m}$ and $C_{\mathsf{mux},m}$ determine the device resources used on the FPGA by the resource instances and dataflow multiplexers. The cost terms $C_{\mathsf{res},m}$ and $C_{\mathsf{mux},m}$ are calculated as follows:

$$C_{\mathsf{res},m} = \sum_{n \in \mathcal{N}_{\mathsf{A},m}} [f_1 + f_2 \mathsf{w}_{\mathsf{LE}}(n) + f_3 \mathsf{w}_{\mathsf{S}}(n)], \tag{5.7}$$

$$C_{\mathsf{mux},m} = \sum_{(n,p) \in \mathcal{I}_m} [f_1 + f_2 \mathsf{w}_{\mathsf{LE}}(\mathsf{x}(m,n,p)) + f_3 \mathsf{w}_{\mathsf{S}}(\mathsf{x}(m,n,p))]. \tag{5.8}$$

The cost terms $C_{\mathsf{res},m}$ and $C_{\mathsf{mux},m}$ are a weighted sum over all used resource instances or multiplexers. The factors f_1, f_2,and f_3 respresent a cost offset, a weight for the size in terms of device resources, and a factor for the control overhead, respectively. The term $\mathsf{w}_{\mathsf{LE}}(\mathsf{x}(m,n,p))$ yields the resource requirements of a multiplexer with $\mathsf{x}(m,n,p)$ inputs and the term $\mathsf{w}_{\mathsf{S}}(\mathsf{x}(m,n,p))$ the associated control overhead.

The cost $C_{\mathsf{wire},m}$ for the datapath interconnect is specified as:

$$C_{\mathsf{wire},m} = \sum_{e \in \mathcal{E}_{\mathsf{A},m}} [f_4 + f_5 \mathsf{w}_{\mathsf{W}}(e)], \tag{5.9}$$

where the function $\mathsf{w}_\mathsf{W}(e)$ yields the word-width of an interconnect e. The factors f_4 and f_5 represent a cost offset for the use of the interconnect and a weight for the word length of an interconnect.

The definition of the implementation cost $C_{\mathsf{dp},m}$ provides a very flexible cost function that allows us to weight the contributions of different cost parameters, depending on the overall objective of the optimization.

Reconfiguration Costs

In this section we apply the RSG model to compute the reconfiguration cost for a set of reconfigurable modules. Therefore, we refine the definitions from Section 3.4.2 (p. 63) for the VA model used here.

Consider a set $\mathcal{M}$ of reconfigurable modules $m \in \mathcal{M}$, where each module is associated with a datapath defined by the set $\mathcal{N}_{\mathsf{A},m}$ of nodes and the set $\mathcal{E}_{\mathsf{A},m}$ of edges. The VA that implements all reconfigurable modules is given by the set $\mathcal{N}_\mathsf{A}$ of nodes and the set $\mathcal{E}_\mathsf{A}$ of edges.

In order to compute the reconfiguration cost, we need to determine, which resource instances and which interconnect can be reused. The use of these datapath elements is described by the device configuration $\mathbf{d}(m) = (\mathsf{d}(m)_1, \ldots, \mathsf{d}(m)_K)$, where an element $\mathsf{d}(m)_k, k = 1, \ldots, K$ denotes the usage of the datapath element k in a reconfigurable module $m \in \mathcal{M}$. Here we do not consider different configurations for the resource instances. It follows that the elements $\mathsf{d}(m)_k, k = 1, \ldots, K$ indicate whether a datapath element k is used by a reconfigurable module m, i.e. $\mathsf{d}(m)_k = 1$, or whether a datapath element k is not used, i.e. $\mathsf{d}(m)_k = 0$.

Similar to Section 3.4.2, the elements $\mathsf{d}(m)_k, k = 1, \ldots, K$ are associated with the nodes $\mathcal{N}_\mathsf{A}$ and the edges $\mathcal{E}_\mathsf{A}$ of the VA (cf. p. 63).

From the device configurations $\mathbf{d}(m)$ of all reconfigurable modules $m \in \mathcal{M}$, the reconfiguration bitmap $\mathbf{r}$ can be derived. In the HLS context, we refine the definition of the reconfiguration bitmap, cf. Equation 3.1 p. 51. Now, the reconfiguration bitmap $\mathbf{r}((m_1, m_2))$ indicates all reconfigurable elements that are used in reconfigurable module m_2 and that have not been used in reconfigurable module m_1:

$$\mathsf{r}((m_1, m_2))_k = \begin{cases} 1 & \text{if } \mathsf{d}(m_2)_k = 1 \wedge \mathsf{d}(m_1)_k = 0 \\ 0 & \text{otherwise} \end{cases}, \; k = 1, \ldots, K. \tag{5.10}$$

The definition is based on the assumption, that any resource instance or interconnect, which is contained in the datapath of module m_1, but not in module m_2 will be overwritten by another resource instance or interconnect that is only present in m_2, but not in m_1.

We define the reconfiguration cost of the datapath according to the average

reconfiguration time defined in Equation 3.4, p. 52:

$$\overline{t} = \frac{1}{|\mathcal{E}_\mathsf{T}|} \sum_{e \in \mathcal{E}_\mathsf{T}} \sum_{k=1}^{K} \mathsf{w}_\mathsf{t}(k) \mathsf{r}(e)_k. \tag{5.11}$$

Now, the set $\mathcal{E}_\mathsf{T}$ represents all reconfigurations between the reconfigurable modules defined in $\mathcal{M}$.

The cost $\mathsf{w}_\mathsf{t}(k)$ to reconfigure an element $k = 1, \ldots, K$ of the device configuration $\mathbf{d}(m) = (\mathsf{d}(m)_1, \ldots, \mathsf{d}(m)_K)$ is now defined as follows (cf. p. 63): The reconfigurable elements $k = 1, \ldots, |\mathcal{N}_\mathsf{A}|$ are associated with the nodes $n_k \in \mathcal{N}_\mathsf{A}, n_k = n_1, \ldots, n_{|\mathcal{N}_\mathsf{A}|}$ and the reconfigurable elements $k = |\mathcal{N}_\mathsf{A}| + 1, \ldots, K$ are associated with the edges $e_k \in \mathcal{E}_\mathsf{A}, e_k = n_1, \ldots, e_{|\mathcal{E}_\mathsf{A}|}$. Now the weight function $\mathsf{w}_\mathsf{t}(k)$ is given by:

$$\mathsf{w}_\mathsf{t}(k) = \begin{cases} f_6 + f_7 \mathsf{w}_\mathsf{LE}(n_k) + f_8 \mathsf{w}_\mathsf{S}(n_k) & \text{if } k = 1, \ldots, |\mathcal{N}_\mathsf{A}| \\ f_9 + f_{10} \mathsf{w}_\mathsf{W}(e_{(k-|\mathcal{N}_\mathsf{A}|)}) & \text{if } k = |\mathcal{N}_\mathsf{A}| + 1, \ldots, K \end{cases} \tag{5.12}$$

Similar to the cost factors f_1–f_5, the factors f_6, f_7, and f_8 respresent a cost offset, a weight for the size in terms of device resources, and a factor for the control overhead, respectively. The factors f_9 and f_{10} represent a cost offset for the use of the interconnect and a weight for the word-width of an interconnect.

Through the factors f_1–f_{10}, the contributions can be weighted differently for the computation of reconfiguration cost and the computation of implementation cost. The factors f_1–f_{10} are summarized in Table 5.2.

Table 5.2: Factors for the implementation and reconfiguration cost weights.

Cost Category	Datapath Element	Description	Factor
Implementation Cost	multiplexer or resource instance	cost offset for usage	f_1
		weight for device resources	f_2
		weight for control overhead	f_3
	interconnect	cost offset for usage	f_4
		weight for word length	f_5
Reconfiguration Cost	multiplexer or resource instance	cost offset for usage	f_6
		weight for device resources	f_7
		weight for control overhead	f_8
	interconnect	cost offset for usage	f_9
		weight for word length	f_{10}

5.3.4 Scheduling

Scheduling assigns an execution time to each operation in a DFG. The execution time t is counted in clock cycles, i.e. $t \in \mathbb{N}$. The execution time determines, when

the operation is started and hence when the input data must be available. Moreover, with a known resource type binding, it is known when the operation is finished, i.e. at time $t + \mathsf{l}$. Similarly, the execution time of an operation determines at what time the associated resource instance is allocated. The resource instance can be used to execute another operation at time $t + \mathsf{o}$, cf. Section 5.3.2. The schedule function c, where $\mathsf{c} : \mathcal{N}_\mathsf{O} \mapsto \mathbb{N}$ assigns to each operation $n \in \mathcal{N}_\mathsf{O}$ an execution time $t \in \mathbb{N}$.

The offset and latency definition is only useful for resource types that execute operations. Variables $v \in \mathcal{N}_\mathsf{V}$ are allocated to resources as long as it is required by the schedule of the operations. The first cycle when the variable v is allocated is denoted as $\mathsf{t}_{\mathsf{min}}(v)$ and the last cycle when the variable v is required is denoted as $\mathsf{t}_{\mathsf{max}}(v)$. Both, $\mathsf{t}_{\mathsf{min}}(v)$ and $\mathsf{t}_{\mathsf{max}}(v)$ are computed as follows: Suppose there is a variable $v \in \mathcal{N}_\mathsf{V}$ and two sets $\mathcal{E}_\mathsf{I}, \mathcal{E}_\mathsf{O} \subset \mathcal{E}_\mathsf{D}$ of data dependencies. The source node of a data dependency e is denoted by $\mathsf{s}(e)$ and the drain node by $\mathsf{d}(e)$ (cf. Definition 3.3, p. 57). The set $\mathcal{E}_\mathsf{I} = \{e \in \mathcal{E}_\mathsf{D} : \mathsf{d}(e) = v\}$ denotes the data dependencies, which cause a write to the variable v. The set $\mathcal{E}_\mathsf{O} = \{e \in \mathcal{E}_\mathsf{D} : \mathsf{s}(e) = v\}$ denotes the data dependencies, which cause a read from the variable v. The time span $\mathsf{t}_{\mathsf{min}}(v), \ldots, \mathsf{t}_{\mathsf{max}}(v)$ where the variable v is allocated is given by:

$$\mathsf{t}_{\mathsf{min}}(v) = \min_{e \in \mathcal{E}_\mathsf{I}} \mathsf{c}(\mathsf{s}(e)) + \mathsf{l}(\mathsf{a}_\mathsf{T}(\mathsf{s}(e))), \tag{5.13}$$

$$\mathsf{t}_{\mathsf{max}}(v) = \max_{e \in \mathcal{E}_\mathsf{O}} \mathsf{c}(\mathsf{d}(e)). \tag{5.14}$$

The function a_T assigns to each node a resource type (cf. Section 5.3.3, p. 143) and the function l yields the latency of that resource type (cf. Section 5.3.2, p. 142). The time $\mathsf{t}_{\mathsf{min}}(v)$ is given by the minimal completion time of any operation $\mathsf{s}(e)$, which writes data to the variable v. The completion time of the operation $\mathsf{s}(e)$ is given by the operation start time $\mathsf{c}(\mathsf{s}(e))$ plus the latency $\mathsf{l}(\mathsf{a}_\mathsf{T}(\mathsf{s}(e)))$ of the operation defined by the associated resource type $\mathsf{a}_\mathsf{T}(\mathsf{s}(e))$. The time $\mathsf{t}_{\mathsf{max}}(v)$ is given by the latest time of any operation $\mathsf{d}(e)$, which reads data from the variable v.

In addition to the variables specified in the DFG it can be necessary to introduce intermediate storage for direct operation-to-operation data dependencies. In our resource model we assume that data is present on a resource output port at cycle $t + \mathsf{l}$ only. If any operation that depends on this data output is scheduled after this cycle then the data must be stored in an intermediate register. Hence, there is an intermediate variable insertion needed after scheduling for all such data dependencies. For any data dependency $e \in \mathcal{E}_\mathsf{D}$ there is an intermediate variable required if:

$$\mathsf{c}(\mathsf{s}(e)) + \mathsf{l}(\mathsf{a}_\mathsf{T}(\mathsf{s}(e))) < \mathsf{c}(\mathsf{d}(e)). \tag{5.15}$$

The schedule function c and the resource type binding a_T determines the total execution time of a DFG. The execution starts at the earliest scheduled node and finishes after the execution of the latest scheduled node has been completed. The time span between those two cycles is called the *total execution latency* L. The total

execution latency L of a DFG $G(\mathcal{N}, \mathcal{E}, \ldots)$ with the set $\mathcal{N}_\mathsf{O} \subset \mathcal{N}$ of operations is given by (cf. Teich [91], p. 85):

$$L = \max_{n \in \mathcal{N}_\mathsf{O}} [\mathsf{c}(n) + \mathsf{l}(\mathsf{a}_\mathsf{T}(n))] - \min_{n \in \mathcal{N}_\mathsf{O}} [\mathsf{c}(n)]. \tag{5.16}$$

In the following, we describe the relationship between scheduling and resource binding.

5.3.5 Constraints for Scheduling and Resource Binding

From the previous discussions, it is apparent that scheduling and binding interact with each other. For instance the schedule function allows or disallows the allocation of operations to the same resource instance. The resource type binding sets constraints for the schedule function and the number of resource instances influences possible schedules, too. This inter-dependency results in several constraints that must be observed during scheduling and resource binding [91, Chapter 3].

The data dependencies and precedence constraints restrict the schedule of operations. A valid schedule function must fullfill the following condition (cf. Teich [91], p. 85):

$$\mathsf{c}(\mathsf{d}(e)) \geq \mathsf{c}(\mathsf{s}(e)) + \mathsf{l}(\mathsf{a}_\mathsf{T}(\mathsf{s}(e))) \quad \forall e \in \mathcal{E}. \tag{5.17}$$

Hence for any edge $e \in \mathcal{E}$ a schedule is valid if the execution time $\mathsf{c}(\mathsf{d}(e))$ of the depending operation $\mathsf{d}(e)$ is greater or equal to the finishing time $\mathsf{c}(\mathsf{s}(e)) + \mathsf{l}(\mathsf{a}_\mathsf{T}(\mathsf{s}(e)))$ of the operation $\mathsf{s}(e)$. If the data dependency originates from a variable, then the data is available one cycle after it has been written by the previous operation.

Resource conflicts arise when a resource instance is allocated more than once in any cycle. A resource instance $n' \in \mathcal{N}'$ is allocated by an operation $n \in \mathcal{N}$ in the interval $\mathsf{c}(n), \ldots, \mathsf{c}(n) + \mathsf{o}(\mathsf{a}_\mathsf{T}(n))$. The allocation offset o is given by the resource type $\mathsf{a}_\mathsf{T}(n)$ associated with operation n (cf. Section 5.3.2, p. 142). Hence, the allocations $\mathsf{a}(n_i) := n'$ and $\mathsf{a}(n_j) := n'$ of any two nodes $n_i, n_j \in \mathcal{N}$, which are allocated to the same resource instance $n' \in \mathcal{N}'$, are conflict free if the execution intervals $\mathsf{c}(n_i), \ldots, \mathsf{c}(n_i) + \mathsf{o}(\mathsf{a}_\mathsf{T}(n_i))$ and $\mathsf{c}(n_j), \ldots, \mathsf{c}(n_j) + \mathsf{o}(\mathsf{a}_\mathsf{T}(n_j))$ do not overlap. Thus, two nodes $n_i, n_j \in \mathcal{N}$ can be allocated to the same resource instance if the following condition holds (cf. Teich [91], p. 168):

$$\mathsf{c}(n_i) + \mathsf{o}(\mathsf{a}_\mathsf{T}(n_i)) \leq \mathsf{c}(n_j) \ \vee \ \mathsf{c}(n_j) + \mathsf{o}(\mathsf{a}_\mathsf{T}(n_j)) \leq \mathsf{c}(n_i)$$
$$\forall\, n_i, n_j \in \mathcal{N}, n_i \neq n_j, \mathsf{a}(n_i) = \mathsf{a}(n_j). \tag{5.18}$$

In addition to resource conflicts, the binding and scheduling must obey any resource constraints. The number of resource instances of a resource type $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$ is constrained by $\mathsf{b}(r_\mathsf{T})$ (cf. Section 5.3.3, p. 144). Thus at any time t, there may be no more instances of the resource type r_T in use than given by $\mathsf{b}(r_\mathsf{T})$. Otherwise it

is not possible to realize a conflict free allocation a. More formally, the following resource constraint must be satisfied for all nodes $n \in \mathcal{N}$ in order to find a valid allocation a for a DFG (cf. Teich [91], p. 89) with a given schedule c and allocation a_T to resource types:

$$|\{n \in \mathcal{N} : \mathsf{a}_\mathsf{T}(n) = r_\mathsf{T} \wedge \mathsf{c}(n) \leq t \leq \mathsf{c}(n) + \mathsf{o}(\mathsf{a}_\mathsf{T}(n))\}| \leq \mathsf{b}(r_\mathsf{T}) \\ \forall\, r_\mathsf{T} \in \mathcal{R}_\mathsf{T}, \forall\, t \in [\min_{n \in \mathcal{N}_\mathsf{O}}(\mathsf{c}(n)), \max_{n \in \mathcal{N}_\mathsf{O}}(\mathsf{c}(n) + \mathsf{l}(\mathsf{a}_\mathsf{T}(n)))]. \quad (5.19)$$

The set described by the first term in Equation 5.19 contains all nodes $n \in \mathcal{N}$ that are bound to the same resource type r_T at cycle t. The size of the set must be less or equal to the number $\mathsf{b}(r_\mathsf{T})$ of instances of the resource type r_T. The condition must hold for any cycle $t = \min_{n \in \mathcal{N}_\mathsf{O}}(\mathsf{c}(n)), \max_{n \in \mathcal{N}_\mathsf{O}}(\mathsf{c}(n) + \mathsf{l}(\mathsf{a}_\mathsf{T}(n)))$ within the schedule of the DFG. The resource constraints are not violated if the condition holds for all resource types $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$.

In this section it has been discussed, which parameters are important to describe the execution of a DFG on a datapath. It is a non-trivial problem to find optimal parameters for resource binding and scheduling. These parameters include the allocation a_T to resource types, the allocation a to resource instances, and the schedule function c. Overall, the execution of the DFG should have a low latency, low resource use and, especially for reconfigurable modules, low reconfiguration cost. Any parameter set is a trade-off between any of these objectives.

The problem of finding good solutions for these parameters has been studied extensively in the past decades. Previously, the allocation and scheduling has been optimized for execution latency, resource utilization and power consumption. In this section we have introduced a cost function that describes the reconfiguration cost. The cost function is based on our virtual architecture model.

Finally, we summarize our method to compute parameters for binding and scheduling.

Even though the scheduling and binding problems are interacting, it is possible to divide them into successive processing steps. In this work we have adopted a three-step approach to determine these parameters. At first, we perform resource type binding. This step yields the resource types as well as constraints on the resource instances. It also provides vital parameters for the next step. Next, in the scheduling step, an execution time is assigned to operations. The schedule function is constrained by the data dependencies and precedence constaints, by the number of available resource instances, and the resource types chosen. With a known schedule function, intermediate variables are inserted in the original DFG for all data dependencies that fullfill the condition in Equation 5.15. Finally, the resource instance binding is performed. Now, all operations and variables are mapped to resource instances. The resource instance binding is restricted by the parameters from the resource type binding and the scheduling. The three-step approach guarantees a feasible implementation of the DFGs, if no constraints on registers resources exist.

5.4 Reconfiguration Optimized Datapath Implementation

High-level synthesis provides a large design space for scheduling and binding. Previously, the primary goal of HLS has been to achieve optimal implementations in terms of resource usage, execution latency, delay, and power consumption. In this work our primary aim is an implementation with minimal reconfiguration costs. In the following we will discuss the effect the resource binding and scheduling on reconfiguration costs in Section 5.4.1. Several resource binding methods have been developed in this work. We present methods to perform resource type binding in Section 5.4.2 and an instance binding method in Section 5.4.3.

Both, implementation cost and reconfiguration cost have been described in the context of HLS in Section 5.3.3. In summary, the implementation cost are determined by the device resources occupied by the datapath of each reconfigurable module and the reconfiguration cost are determined by the differences in terms of resource instances and interconnect.

5.4.1 Effects of Scheduling and Binding on Reconfiguration Costs

The scheduling and binding steps determine the datapath implementations for the DFGs and hence, the implementation costs and reconfiguration costs. Here, we discuss how different decisions made at these steps can influence these costs. This section serves a motivation for the strategies employed at the resource type binding and resource instance binding steps.

The chosen resource type binding determines the kind of resources that are used in each reconfigurable module and hence, which resource types are available to exploit reuse. Moreover, the chosen resource types also control the kind of interconnect that is used to propagate data in the datapath. The resource type binding decides, which operations may share the same resource instance and which data dependencies may be mapped to the same interconnect. Thus, resource type binding can have large impact on the resource instance binding. The effects can be put into three categories:

Type binding variants Many operations can be bound to many different resource types, which results in different solutions for the execution time, resource sharing and datapath connectivity.

Port re-labelling In the datapath, each data dependency is mapped to a connection between resource instances. Therefore a re-labelling of the port labels associated with the data dependencies to the port labels associated with the datapath interconnect is performed. The re-labelling occurs during resource type binding, because

here the allocation of DFG nodes to resource types is chosen.

Commutative operations For many resource types that implement arithmetic or logical operations the re-labelling of ports is flexible, because the operations are commutative. Hence, the datapath structure depends not only on the type of resource chosen, but also on the specific mapping.

The resource type binding obviously effects the scheduling and hence, the number of resource instances as well as the intermediate variables that must be introduced after scheduling.

The resource type binding defines constraints for the scheduling and for the resource instance binding. The constraints either allow or disallow the reuse of resources. The actual resource reuse and the resulting reconfiguration costs are finally determined by the resource instance binding step. The reuse is directly determined by the allocation a of nodes in the DFGs to the set $\mathcal{N}_\mathsf{A}$ of resource instances in the virtual architecture $G_\mathsf{A}(\mathcal{N}_\mathsf{A}, \mathcal{E}_\mathsf{A}, \ldots)$ (cf, Section 3.4.2, p. 62 and Section 5.3.3, p. 145). If a resource or a connection is present in two reconfigurable modules, then it can be reused and must not be reconfigured. The effect of the resource instance binding on reuse can be measured instantly in the virtual architecture model.

5.4.2 Strategies for Resource Type Binding

It has been discussed in the previous section that the resource type binding serves as an enabler to both intra- and inter-task sharing of resources and datapath interconnect. Operations that belong to different reconfigurable modules can reuse a resource instance, if the operations have been mapped to the same resource type.

Furthermore, the chosen type binding presumes the interconnect that can potentially be reused. Therefore the data dependencies $e \in \mathcal{E}$ in a DFG $G(\mathcal{N}, \mathcal{E}, \ldots)$ are allocated to connection types $e' \in \mathcal{E}'$ in an image graph $G'(\mathcal{N}', \mathcal{E}', \ldots)$ by the allocation function a_T (cf. Section 5.3.3, p. 143) and the port re-labelling functions a_s and a_d (cf. Section 3.4.1, p. 59) as follows: $e' = (\mathsf{a}_\mathsf{T}(\mathsf{s}(e)), \mathsf{a}_\mathsf{T}(\mathsf{d}(e)), \mathsf{a}_\mathsf{s}(e), \mathsf{a}_\mathsf{d}(e))$. Two data dependencies that are mapped to the same connection type, can possibly mapped to the same interconnect in the VA. The realization of data dependencies on the same interconnect is useful inside each reconfigurable module, because the interconnect complexity is reduced. Also, this interconnect may be reused between different reconfigurable modules, which reduces reconfiguration costs.

In this work we evaluate two different strategies for resource type binding. In one approach we minimize the number of allocated resource types over all DFGs. In the other approach we minimize the number of connection types..

Minimum Number of Resource Types

The number of different resource types influences the intra-task as well as the inter-task resource sharing possibilities. For example, an ALU offers more potential to be shared by different operations than a simple resource type that can only perform an addition operation. I.e. if resource types are chosen more general then there are more operations that can be bound to such a type, hence the possibilities for resource sharing are increased. This also increases the probability that interconnect between resources can be re-used. In contrast, more general resource types tend to require more device resources and a more complex control compared to specialized types.

The resource type binding of the sets $\mathcal{N}_i$ of nodes from the DFGs $G_i(\mathcal{N}_i, \mathcal{E}_i, \ldots)$, $i \in \mathcal{N}_\mathsf{T}$ to a minimal number of resource types can be formulated as an ILP, which can be solved to optimality. The ILP is shown in Program 3.

Program 3 Minimum Number of Resource Types

minimize:

$$\sum_{r_\mathsf{T} \in \mathcal{R}_\mathsf{T}} (f_1 + f_2 \mathsf{w}_{\mathsf{LE}}(r_\mathsf{T}) + f_3 \mathsf{w}_\mathsf{S}(r_\mathsf{T})) S_{r_\mathsf{T}} \tag{5.20}$$

subject to:

$$\forall n \in \mathcal{N} : 1 = \sum_{\forall r_\mathsf{T} : (n, r_\mathsf{T}) \in \mathcal{E}_\mathsf{R}} S_{n, r_\mathsf{T}} \tag{5.21}$$

$$\forall r_\mathsf{T} \in \mathcal{R}_\mathsf{T} : S_{r_\mathsf{T}} = \bigvee_{\forall n : (n, r_\mathsf{T}) \in \mathcal{E}_\mathsf{R}} S_{n, r_\mathsf{T}} \tag{5.22}$$

where:

$$S_{r_\mathsf{T}}, S_{n, r_\mathsf{T}} \in \{0, 1\} \tag{5.23}$$

The starting point to formulate the ILP is the resource graph $G_\mathsf{R}(\mathcal{N}, \mathcal{R}_\mathsf{T}, \mathcal{E}_\mathsf{R})$ defined in Definition 5.1, p. 143. We model the binding to resource types as follows: We introduce a binary variable $S_{n, r_\mathsf{T}}, n \in \mathcal{N}, r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$ for each edge $(n, r_\mathsf{T}) \in \mathcal{E}_\mathsf{R}$. Equation 5.21 ensures that each node n is bound to exactly one resource type r_T. Whether a resource type r_T is used in the datapath or not is determined by the binary variable S_{r_T}. A resource type r_T will be used in any datapath if at least one operation is bound to that type and thus, S_{r_T} is set to 1. Finally the number of used resource types minimized in the objective function.

Minimum Number of Connection Types

In an alternative approach, we seek to minimize the number of connection types. The aim is to enable as much interconnect sharing as possible. Note that, if two data

dependencies are allocated to equal connection types, then those data dependencies can possibly share an interconnect in the datapath. In particular we minimize the connection types that are introduced by the resource type binding. Again, this problem is described as ILP.

Program 4 Minimum Number of Connection Types

minimize:

$$\sum_{e' \in \mathcal{E}'} (f_4 + f_5 \mathsf{w}_\mathsf{W}(e')) S_{e'} \tag{5.24}$$

subject to:

$$\forall n \in \mathcal{N} : 1 = \sum_{\forall r_\mathsf{T} \in \mathcal{R}_\mathsf{T} : (n, r_\mathsf{T}) \in \mathcal{E}_\mathsf{R}} S_{n, r_\mathsf{T}} \tag{5.25}$$

$$\forall e' = (r_{\mathsf{T},1}, r_{\mathsf{T},2}, p_1, p_2) \in \mathcal{E}' : S_{e'} = \bigvee_{\substack{e \in \mathcal{E}, \\ (n_1, r_{\mathsf{T},1}), (n_2, r_{\mathsf{T},2}) \in \mathcal{E}_\mathsf{R}: \\ p_1 = \mathsf{a_s}(e) \wedge p_2 = \mathsf{a_d}(e)}} S_{n_1, r_{\mathsf{T},1}} \wedge S_{n_2, r_{\mathsf{T},2}} \tag{5.26}$$

where:

$$S_{n, r_\mathsf{T}}, S_{e'} \in \{0, 1\} \tag{5.27}$$

The objective of the ILP aims to minimize the number of different connection types. The Equation 5.25 ensures that all nodes are bound to exactly one resource type. We define a binary variable $S_{e'}$ for any possible connection type $e' \in \mathcal{E}'$.

The constraint in Equation 5.26 assigns a 1 to the binary variable $S_{e'}$ if corresponding connection type e' is present in the solution and 0 if not. A connection type e' is chosen, if any edge $e \in \mathcal{E}$ in $G(\mathcal{N}, \mathcal{E}, \ldots)$ is allocated to the edge $e' \in \mathcal{E}'$ in $G'(\mathcal{N}', \mathcal{E}', \ldots)$. Hence, the or-operator ($\bigvee$) iterates over all edges $e \in \mathcal{E}$ and all possible allocations $(n_1, r_{\mathsf{T},1}) \in \mathcal{E}_\mathsf{R}$, $(n_2, r_{\mathsf{T},2}) \in \mathcal{E}_\mathsf{R}$ of the source node $n_1 = \mathsf{s}(e)$ and the drain node $n_2 = \mathsf{d}(e)$. If both, the allocation of the source node and of the drain node are chosen, which is indicated by the binary variables $S_{n_1, r_{\mathsf{T},1}}$ and $S_{n_2, r_{\mathsf{T},2}}$, then the connection type is chosen by setting the $S_{e'} = 1$.

The Program 4 is much more complex than Program 3 in terms of generated constraints. The number of resource types results only from the mapping of individual nodes to possible resource types. In contrast, the number of connection types results from the mapping of data dependencies, which results from the combination of the mapping of the individual nodes. In the experiments in Section 5.5 we will determine, which method results in lower reconfiguration costs for the reconfigurable modules.

5.4.3 Strategies for Resource Instance Binding

The type binding step aims to provide reasonable constraints to the scheduling and instance binding steps. Type binding is based only on strategies that were designed to enable resource and interconnect reuse between configurations. In contrast, the resource instance binding determines the structure of the datapath in detail.

The number of required resource instances to realize each DFG is determined by the scheduling step. However in this section we discuss several options how the number of resource instances in the VA can be defined. The resource instance binding may benefit from more resource instances, because the number of resource instances has a severe impact on the allocation of DFG nodes and data dependencies. This is mainly due to the required dataflow multiplexers.

Next, we introduce a model that describes the constraints presented in Section 5.3.5. Finally we introduce a simulated annealing (SA) based algorithm to actually perform the resource instance binding with the given constraints and the cost functions defined in Section 5.3.3.

Resource Instances in the VA

In our approach the minimum number of required resource instances b are known after the scheduling step. However, the scheduling defines this number for a single DFG only. Moreover it can be useful to apply different strategies on how many resource instances are available in the virtual architecture (VA) that is used for the resource instance binding. Before those strategies are explained in detail, we reconsider the function b after scheduling: For any DFG $G_i(\mathcal{N}_i, \mathcal{E}_i, \ldots)$ and a resource type $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$ the number of required resource instances is given by $\mathsf{b}_i(r_\mathsf{T})$.

Now we compute how many resource instances are required in the VA $G_\mathsf{A}(\mathcal{N}_\mathsf{A}, \mathcal{E}_\mathsf{A}, \ldots)$ for a set $\mathcal{N}_\mathsf{T}$ of tasks. Following that, we compute how many resource instances are required if no intra-task resource sharing is applied.

Resource instances based on scheduling constraints During scheduling, the minimum number of required resource instances for the DFG of each task $i \in \mathcal{N}_\mathsf{T}$ is determined. Now we compute how many resource instances of each type are required for each reconfigurable module and for the VA in order to perform the resource instance binding.

Again, we imply a set $\mathcal{M}$ of reconfigurable modules $m \in \mathcal{M}$. The set $\mathcal{N}_{\mathsf{T},m}$ denotes all tasks that are realized on the reconfigurable module m.

In each reconfigurable module the dominant resource requirement above all tasks in that module must be satisfied. It follows that for each reconfigurable module, there must be as many resource instances provided of each resource type, as there are needed by the maximum requirement of a task. Thus the number $\mathsf{b}_m(r_\mathsf{T})$ of instances of the resource type r_T that is required by the reconfigurable module m

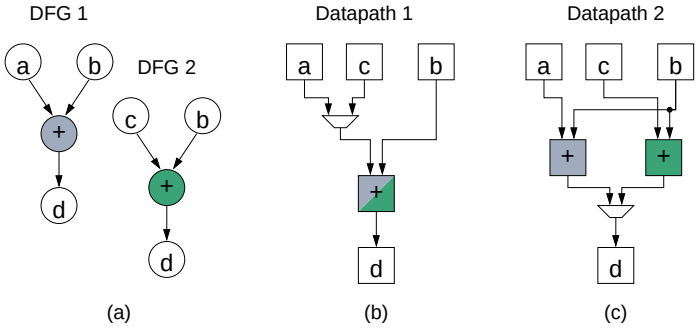


Figure 5.7: Dataflow control and resource sharing, cf. Example 5.6. (a) shows an excerpt from a DFG and (b),(c) depict two possible realizations on a datapath.

is given by:

$$\mathsf{b}_m(r_\mathsf{T}) = \max_{i \in \mathcal{N}_{\mathsf{T},\mathsf{m}}} \mathsf{b}_i(r_\mathsf{T}). \tag{5.28}$$

Note that a reconfigurable module may use occupy FPGA resources for resource instances that are used in some, but not all tasks that are realized in this module.

Similarly, the VA must provide enough resource instances of each type such that the requirement of the largest reconfigurable module can be satisfied. The number of resource instances of required any resource type $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$ in the VA are given by:

$$\mathsf{b}_{\mathsf{VA}}(r_\mathsf{T}) = \max_{m \in \mathcal{M}} \mathsf{b}_m(r_\mathsf{T}) = \max_{i \in \mathcal{N}_\mathsf{T}} \mathsf{b}_i(r_\mathsf{T}). \tag{5.29}$$

The allocation of only minimal resource instances in each reconfigurable module often requires that resources are shared between nodes in the DFG. This results frequently in additional multiplexers for the dataflow control. In FPGAs, large multiplexers require more resources than basic ALU functions. Hence the total number of FPGA resources required by a reconfigurable module can be less, when more computational resources are allowed.

Resource instances with no intra-task resource sharing The problem of dataflow control can be alleviated if more resource instances are provided for resource instance binding than absolutely necessary. Then the binding algorithm can decide whether to apply resource sharing or not.

The problem of dataflow control overhead can not be completely resolved in all cases by avoiding resource sharing, as explained in Example 5.6. Hence, the binding algorithm must be able to find a balanced solution, given that there are sufficient resource instances available.

Example 5.6 *This example illustrates that dataflow multiplexers can be necessary, even though resource sharing is avoided.*

Consider the DFG excerpts in Figure 5.7(a). The operation nodes '+' in both DFGs use the variables a–c as input and write the output both to variable 'd'.

Figure 5.7(b) shows a datapath realization with resource sharing. Both operations are mapped to one resource '+'. As a benefit, computational resources are saved as well as the dataflow control at the register 'd'. The input data on one port of resource '+' is selected by the dataflow control.

Alternatively, Figure 5.7(c) avoids resource sharing by mapping each operation to an individual resource instance. However, the result of both operations is written to the same register 'd', which makes it necessary to implement a dataflow multiplexer again.

The number of instances of a resource type $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$ that are required to realize the DFG $G_i(\mathcal{N}_i, \mathcal{E}_i, \ldots)$ of a task $i \in \mathcal{N}_\mathsf{T}$ with no resource sharing is given by:

$$\mathsf{b}_i(r_\mathsf{T}) = |\{n \in \mathcal{N}_i : \mathsf{a}_\mathsf{T}(n) = r_\mathsf{T}\}|. \tag{5.30}$$

Again, the number of resource instances in the reconfigurable modules and in the VA are given by the Equations 5.28 and 5.29.

We have discussed different possibilities of how many resource instances shall be included in the VA. The availability of these instances is only a prerequisite for different binding strategies. The number of resource instances that are available at the binding step define the size of the VA. During instance binding it will be calculated how many resources are actually used in each configuration; the VA is not necessarily utilized completely by any reconfigurable module. Moreover, the resource instances in the VA define the space for optimizations at the resource instance binding step.

Allocation Constraints

The scheduling restricts the resource sharing. The restrictions are expressed as resource conflicts in Equation 5.18, p. 151. Resource conflicts can be represented in either a conflict graph or the inverse representation, the compatibility graph [91].

Here, we define a conflict graph $G_\mathsf{C}(\mathcal{N}, \mathcal{E}_\mathsf{C})$ where the nodes $n \in \mathcal{N}$ are connected by an edge $e \in \mathcal{E}_\mathsf{C}$ if they must not be bound to the same resource instance. The edges in the conflict graphs are given by:

$$\mathcal{E}_\mathsf{C} = \{(n_i, n_j), n_i, n_j \in \mathcal{N} : n_i \neq n_j \wedge \mathsf{c}(n_i) \leq \mathsf{c}(n_j) \leq \mathsf{c}(n_i) + \mathsf{o}(\mathsf{a}_\mathsf{T}(n_i))\}. \tag{5.31}$$

Hence, any combination of nodes (n_i, n_j), where the execution time of node n_j falls into the execution time interval of node n_i, results in an edge in the conflict graph. Resource conflicts occur only between the nodes of one DFG, because our execution model assumes that tasks are executed sequentially on the reconfigurable modules.

Note that the use of resource instances in a configuration is secondary objective. Primarily we focus on the overall hardware resource demand and reconfiguration costs, when the resource instance binding is done.

Resource Instance Binding with Simulated Annealing

In our HLS, the resource instance binding is performed with simulated annealing (SA). In Section 3.5.2 we have already described how to use an SA based algorithm to optimize the allocation of nodes for low reconfiguration cost. Therefore in this section, we describe only the key aspects of the SA algorithm. First we describe how the initial solution is obtained. Second, we explain the permutation of the solution such that resource conflicts are avoided. Third, the employed cost function is given.

The inputs to the SA algorithm are:

- the input graphs G_i for each task $i \in \mathcal{N}_\mathsf{T}$,
- the set $\mathcal{M}$ of reconfigurable modules,
- the tasks $i \in \mathcal{N}_{\mathsf{T},m}$ that are assigned to a reconfigurable module $m \in \mathcal{M}$,
- the allocation a_T of nodes $n \in \mathcal{N}_i$ to resource types $r_\mathsf{T} \in \mathcal{R}_\mathsf{T}$,
- the conflict graph $G_\mathsf{C}(\mathcal{N}, \mathcal{E}_\mathsf{C})$, and
- the set $\mathcal{N}_\mathsf{A}$ of resource instances of the VA graph $G_\mathsf{A}(\mathcal{N}_\mathsf{A}, \mathcal{E}_\mathsf{A}, \ldots)$.

The SA algorithm computes the allocation a of nodes to resource instances and the interconnect $\mathcal{E}_\mathsf{A}$ of the VA. Thereby, the SA algorithm aims to achieve minimal implementation cost and reconfiguration cost.

The port re-labelling is defined by the resource type binding (cf. Section 5.3.3, p. 143). The edge allocation a_e is derived from the allocation a and the port re-labelling, cf. Section 3.4.1, p. 60.

Initial Solution The initial solution of the allocation a is computed e.g. with the Left-Edge algorithm [37][91].

However, if resource sharing prohibited, then Algorithm 7 can be used to gain an initial solution. Assume there is a set $\mathcal{N}''$ that contains all nodes that do not share resource instances. The set $\mathcal{N}''$ contains e.g. all nodes $n \in \mathcal{N}_i$ of one task $i\mathcal{N}_\mathsf{T}$ or all nodes $n \in \bigcup_{i \in \mathcal{N}_{\mathsf{T},m}} \mathcal{N}_i$ of all tasks implemented in a reconfigurable module m. The nodes of the set $\mathcal{N}''$ are successively assigned to a resource instance taken from a set $\mathcal{P}$ of resource instances. The set $\mathcal{P}$ of resource instances initially contains all instances $\mathcal{N}_\mathsf{A}$ that are available. If a resource instance r is allocated by a node n, this instance is removed from the set $\mathcal{P}$.

Algorithm 7 Resource Instance Binding without Resource Sharing

1: $\mathcal{P} = \mathcal{N}_\mathsf{A}$; // initialize the pool of resources
2: **for** $\forall n \in \mathcal{N}''$ **do**
3: $\quad \mathrm{r} = \mathit{select}(\mathcal{P}, \mathsf{a}_\mathsf{T}(n))$;
4: $\quad \mathsf{a}(n) = r$;
5: $\quad \mathcal{P} = \mathcal{P} \setminus r$;
6: **end for**

Permutation of the Solution The current solution is defined by the allocation a. The current solution is permutated by changing the allocation of DFG nodes to resource instances. This permutation must be performed consistently with the conflict graph.

In order to compute a permutation of the allocation, a node $n \in \mathcal{N}_i$ is selected randomly from any task $i \in \mathcal{N}_\mathsf{T}$. In the current solution, the node n is allocated to the resource instance $r = \mathsf{a}(n)$ with the resource type $r_\mathsf{T} = \mathsf{a}_\mathsf{T}(n)$. For the node n, a new allocation $\mathsf{a}'(n) := r'$ is chosen from the set $\mathcal{R}_n \setminus \{r\}$ of available resource instances $r' \in \mathcal{R}_n \setminus \{r\}$, where $\mathcal{R}_n$ contains all resource instances of the resource type given by $\mathsf{a}_\mathsf{T}(n)$.

Now, other nodes may require a new allocation if resource conflicts arise from the allocation $\mathsf{a}'(n) := r'$. We observe that the permutation effects only nodes that are allocated to either r or r'. The nodes that are allocated to r are given by the set $\mathcal{N}_1 = \{n \in \mathcal{N}_i : \mathsf{a}(n) = r\}$ and the nodes that are allocated to r' are given by the set $\mathcal{N}'_1 = \{n \in \mathcal{N}_i : \mathsf{a}(n) = r'\}$. We define a set $\mathcal{E}'_\mathsf{C} \subset \mathcal{E}_\mathsf{C}$ that contains any edge between the nodes $\mathcal{N}_1$ and $\mathcal{N}'_1$.

The nodes that must be re-allocated for the chosen permutation are now given by the nodes $\mathcal{N}_2 \subset \mathcal{N}_1$ and $\mathcal{N}'_2 \subset \mathcal{N}'_1$. The sets $\mathcal{N}_2, \mathcal{N}'_2$ contain all nodes that can be reached by any path contained in the graph $G(\mathcal{N}_1 \cup \mathcal{N}'_1, \mathcal{E}'_\mathsf{C})$ that originates in n.

A conflict free permutation from $\mathsf{a}(n) = r$ to $\mathsf{a}'(n) := r'$ involves the permutation of all nodes $\mathcal{N}_2, \mathcal{N}'_2$:

$$\forall n \in \mathcal{N}_2 \quad : \quad \mathsf{a}'(n) := r' \tag{5.32}$$

$$\forall n \in \mathcal{N}'_2 \quad : \quad \mathsf{a}'(n) := r \tag{5.33}$$

All other nodes are not affected by this permutation. Example 5.7 illustrates the permutation of node allocations.

Example 5.7 *In Figure 5.8(a) a conflict graph for the nodes A–I is shown. The current allocation to the resource instances r and r' is shown in Figure 5.8(b).*

The binding of node F shall be changed from r to r', without inferring resource conflicts. Therefore all nodes allocated to r are denotes as $\mathcal{N}_1$ and all nodes allocated to r' are denotes as $\mathcal{N}'_1$. Only the nodes $\mathcal{N}_2 = \{D, F\}$ and the nodes $\mathcal{N}'_2 = \{G\}$ are connected to node F. Thus in Figure 5.8(c) the nodes in $\mathcal{N}_2$ are now allocated to r' and the node in $\mathcal{N}'_2$ is now allocated to r.

As it can be seen, the new allocation a' in Figure 5.8(c) is also conflict free. The nodes A,B,C,E,H, and I are not effected by the permutation.

The permutation of nodes is more complex for resource instance binding than the method described in Section 3.5.2. The method described earlier assumed that any nodes that belong to the same task, must be permutated. However the method described now considers the conflict graph.

Note that the resource binding of the nodes is gently modified by this permutation. The effect on the connections can be more severe, because each node infers multiple connections in the datapath.

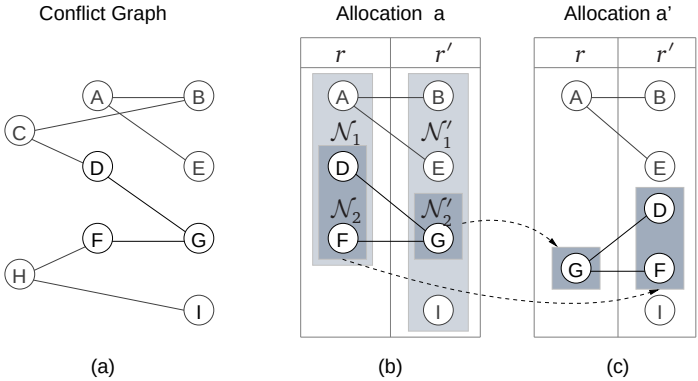


Figure 5.8: Example for conflict free permutation.

Calculation of the Cost Function The cost function is calculated from the allocation a, which represents the current solution in the SA algorithm. We use a cost function C that combines implementation costs C_{dp} defined in Equation 5.6 and reconfiguration costs $\overline{t}$ defined in Equation 5.11:

$$C = \overline{t} + \frac{1}{|\mathcal{M}|} \sum_{\mathcal{M}} C_{\text{dp}}. \tag{5.34}$$

The factors f_1–f_{10} allow a tuning of the cost function for different objectives. Hence, the SA algorithm can optimize the allocation a for implementation costs, reconfiguration costs or both.

Now we describe how the cost function is computed from the allocation in more detail. The allocation a describes, which nodes $n \in \mathcal{N}_i$ of the tasks $i \in \mathcal{N}_\text{T}$ are allocated to which nodes $n' \in \mathcal{N}_\text{A}$ in the virtual architecture $G_\text{A}(\mathcal{N}_\text{A}, \mathcal{E}_\text{A}, \ldots)$. Thus, from the allocation we can directly compute the implementation costs C_{res} for the resource instances (cf. Equation 5.7, p. 147).

The connectivity and the required dataflow multiplexers are a secondary result of the allocation. The interconnect edges $e \in \mathcal{E}_\text{A}$ in the VA are derived from the allocation a. The implementation costs C_{wire} (Equation 5.9, p. 147) for the interconnect are derived from the set $\mathcal{E}_{\text{A},m} \subset \mathcal{E}_\text{A}$ of edges that are used by a reconfigurable module m. The implementation costs C_{mux} (Equation 5.8, p. 147) for the dataflow multiplexers are derived similarly.

The reconfiguration costs for resource instances and interconnect are computed

from the differences in the device configurations $\mathbf{d}(m_1)$ and $\mathbf{d}(m_2)$ of any two reconfigurable modules $m_1, m_2 \in \mathcal{M}$ (cf. Section 5.3.3, p. 148). The device configurations are given by the utilization of the VA by the reconfigurable modules.

The SA algorithm can be implemented straightforward. The conflict graph ensures that only valid solutions are generated, because the initial solution is valid and each permutation ensures that the new solution is also valid. The changes in the interconnect that result from the permutation of the resource instance allocation result in a complex update of the VA interconnect. Hence the computation of the cost function becomes expensive compared to the simple permutation of the current solution. Nevertheless the SA implementation provides quality results for the computed datapaths as we will see on a series of examples in Section 5.5.

5.5 Experiments

In the previous section we have developed a range of methods to optimize the implementation of tasks in terms of implementation and reconfiguration cost. In this section we present a series of experiments, where our HLS tool has been used to generate reconfigurable modules for different applications. The results are presented in this section together with an in-depth analysis of the implementation and reconfiguration cost. The experimental results allow a direct comparison of the cost between established solutions and our method. The experiments clearly demonstrate the advantages of our methodology compared to previous methods. We reveal how the results of our HLS tool can be applied in order to realize the novel reconfiguration methods proposed in Section 5.2. Further we show which methods for type binding and instance binding are to be preferred.

This section is structured as follows: First we summarize the binding methods that have been employed and determine the necessary cost factors for the cost function in the optimization in Sections 5.5.1 and 5.5.2. Second we formulate four implementation scenarios that describe to use of our tool according to the proposed reconfiguration methods in Section 5.5.3. Third, we briefly describe the characteristics of the benchmark applications (Section 5.5.4). In Section 5.5.5 we present the results that have been obtained for the implementation scenarios. Finally in Section 5.5.6 we discuss the results and extrapolate the results towards more complex applications.

5.5.1 Summary of Binding Methods and Tool Setup

In the Sections 5.4.2 and 5.4.3 we described several methods for resource type binding and resource instance binding. In our experiments we used our HLS to implement several benchmark set. Each benchmark set has been implemented with

different combinations of type binding and instance binding methods. This allows us to analyze the performance of each combination in terms of implementation costs and reconfiguration costs. With our results it is possible to identify the best combination of resource type binding and resource instance binding depending on the system requirements.

Resource Type Binding

Resource type binding can be done with one of the methods described in Section 5.4.2. The methods choose the resource types such that either the cost for resource types (Equation 5.20) or the cost for connect types (Equation 5.24) is reduced. In addition, we generated results with a method that simply assigns the module type with minimal cost to each operation.

The cost optimization has been applied to a either single task or to all tasks at the same time. Hence we optimized the types used in an individual task or the types used over all tasks. When considering the tasks individually, intra-task resource sharing is improved but inter-task resource sharing of the types is neglected. If all tasks are considered at the same time, then an overall optimization is achieved. The investigated resource type binding methods are listed below:

1. Choose the module type with minimal cost for each operation in a task.
2. Minimize the cost for module types for each task individually.
3. Minimize the cost for both, module types and connect types for each task individually.
4. Minimize the cost for module types over all tasks.
5. Minimize the cost for both, module types and connect types over all tasks.

Resource Instance Binding

The implementation and reconfiguration cost are finalized with the resource instance binding. We investigated several optimization targets during this step that were combined with the different type binding methods. The results that have been obtained depend on the combination of both steps. The different objectives for the resource instance binding step are:

1. Minimize the implementation cost for module instances and interconnect individually for each task.
2. Minimize the implementation cost for module instances and interconnect for all tasks merged into one datapath.
3. Minimize the reconfiguration cost for module instances.
4. Minimize the reconfiguration cost for module instances and interconnect.

The methods are selected such that the results will illustrate the range of possible realizations with different implementation and reconfiguration cost trade-offs.

5.5.2 Cost Factors

The modules and interconnect used in a datapath configuration cause different kind of cost, which makes a global optimization of all parameters difficult. Common solutions to this problem are the use of multiobjective optimization or the use of a weighted cost function that describes a global optimization target. In our experiments we choose the second option to simplify the optimization. The general cost function that has been used in the SA algorithm for the resource instance binding is given in Equation 5.34.

The different binding methods are realized by using different factors f_1–f_{10} (cf. Table 5.2) in cost function, in order to perform the optimization with different objectives. The weights of the cost functions are given in Table 5.3. The weighting of the resource and interconnect cost has been chosen for the following reasons: The use of a resource type or instantiation introduces a penalty $f_1 = f_4 = f_6 = f_9 = 1$ on the implementation or reconfiguration of that resource. For resources we emphasize use of device resources by setting the $f_2 = f_7 = 2$. The factor for the control wires is set such that the cost of an interconnect is equal to the cost of a resource with the same word length, i.e. $f_5 = f_{10} = 1$.

Table 5.3: Setup of the factors f_1–f_{10} used in the computation of the implementation and reconfiguration cost for different binding methods in Section 5.5.1. The symbol '–' means that the factor is not applied in this type of optimization.

	resource instance and multiplexer† cost			interconnect cost		resource instance reconfiguration cost			interconnect reconfiguration cost	
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
Type Binding 1	1	2	1	0	0	–	–	–	–	–
Type Binding 2	1	2	1	0	0	–	–	–	–	–
Type Binding 3	1	2	1	1	1	–	–	–	–	–
Type Binding 4	1	2	1	0	0	–	–	–	–	–
Type Binding 5	1	2	1	1	1	–	–	–	–	–
Instance Binding 1	1	2	1	1	1	0	0	0	0	0
Instance Binding 2	1	2	1	1	1	–	–	–	–	–
Instance Binding 3	0	0	0	0	0	1	2	1	0	0
Instance Binding 4	0	0	0	0	0	1	2	1	1	1

† Multiplexer cost are not considered during type binding.

5.5.3 Implementation Scenarios

In Section 5.2 we have described new methods that take advantage of HLS in order to realize tasks on reconfigurable modules. In order to demonstrate the effect of our HLS tool, we specify four different implementation scenarios A–D based on these concepts. The scenarios include static as well as dynamically reconfigurable solutions. With the implementation scenarios it is possible to compare traditional implementation methods with the new methods developed in this work.

The implementation and reconfiguration cost presented here reflect the cost that appear when the datapaths of the reconfigurable modules are implemented as described in the scenarios. We will describe how these cost are computed for each scenario.

A: Static, Parallel Implementation

Scenario A represents the classical way to implement the HW tasks on an FPGA. All tasks are realized concurrently on the device. Each task occupies it's own set of resources including logic resources, memory or bus interface logic. This scenario is used if the tasks need to run concurrently or if no dynamic reconfiguration is available in the device. In this scenario the utilization of occupied device resources is low if the tasks are not required to run at the same time. Hence, this method lead to inefficient implementations.

The implementation cost C_{A} are given by the sum of the costs of the different modules, cf. Equation 5.6. We assume that each task $i \in \mathcal{N}_{\mathsf{T}}$ is assigned to an individual module m, i.e. $\mathcal{N}_{\mathsf{T},m} = \{i\}, m = 1, \ldots, |\mathcal{N}_{\mathsf{T}}|, i \in \mathcal{N}_{\mathsf{T}}$ (cf. p. 145):

$$C_{\mathsf{A}} = \sum_{m \in \mathcal{M}} C_{\mathsf{dp},m}. \tag{5.35}$$

Obviously the cost for dynamic reconfiguration are zero in a static implementation.

For a static parallel solution the tasks are usually optimized for size individually, independent of the other tasks.

B: Static, Sequential Implementation

In Section 5.2.1 we proposed to implement multiple tasks in one reconfigurable module. Accordingly, in scenario B we assume that all tasks of one benchmark application are implemented in one datapath. Thus no dynamic reconfiguration is required. Instead, the control unit executes the different tasks on the static datapath. Since all tasks are executed on the same datapath resources, they can not run concurrently but only sequentially. This scenario may require considerable less resources and is more efficient if the tasks are not required to run concurrently. The

control memory may hold the control data for all tasks at once. Alternatively the control data can be loaded dynamically for each task before the task is executed.

The implementation cost C_B are the cost for the single static datapath that can realize all tasks. Hence, all tasks are considered to belong to one (static) module, i.e. $\mathcal{N}_{T,1} = \mathcal{N}_T$. The reconfiguration cost are now:

$$C_B = C_{dp,1}. \tag{5.36}$$

Again a static implementation causes no dynamic reconfiguration costs.

C: Reconfiguration without Reuse of Resources

Scenario C assume a full partial reconfiguration between reconfigurable modules. It reflects the established use of dynamic reconfiguration, whereas the datapath is considered as a single unit which is reconfigured completely. The reuse of resource instances and interconnect between reconfigurable modules is not considered. The tasks can be executed only sequentially and must be configured on the device beforehand. The reconfigurable scenario C can require considerable less resources than the scenarios A and B, because each datapath contains only resources that are needed by a single task. The downside are the configuration costs.

The implementation cost C_C are computed as the average implementation cost over all reconfigurable modules:

$$C_C = \frac{1}{|\mathcal{M}|} \sum_{m \in \mathcal{M}} C_{dp,m}. \tag{5.37}$$

In scenario C, there is assumed that the datapath is completely reconfigured. Therefore, the average reconfiguration cost R_C are equal to the average implementation cost of a reconfigurable modules:

$$R_C = C_C. \tag{5.38}$$

D: Reconfiguration with Reuse of Resources

In order to reduce the reconfiguration costs we propose the reuse of resources between different reconfigurable modules. The potential of this technique is analyzed in scenario D. This scenario is the same as scenario C except that the datapath is not considered as a single unit. Here we assume that resources from the previous reconfigurable modules are reused by the new reconfigurable module. With dynamic reconfiguration, the only new resource instances and interconnect are added to the datapath. This overwrites the configuration of now unused resources from the previous reconfigurable module.

The implementation cost are computed as in scenario C:

$$C_{\mathsf{D}} = \frac{1}{|\mathcal{M}|} \sum_{m \in \mathcal{M}} C_{\mathsf{dp},m}. \tag{5.39}$$

Now, the average reconfiguration cost are computed according to the reconfiguration cost model (cf. Equation 3.4):

$$R_{\mathsf{D}} = \overline{t}. \tag{5.40}$$

As expected, the focus of implementation cost and reconfiguration cost is different for each scenario. Hence we would like to identify the preferred methods of type binding and instance binding for each scenario.

The scenarios A–C may lead to the least cost if the datapath is optimized in terms of logic resources used for operations and multiplexer logic. It is assumed that the datapath implementation is not limited by interconnect resources. In scenario A the tasks can be optimized independent of each other, because inter-task reuse of resources is not applicable. Scenario B is likely to benefit from intra-task as well as inter-task reuse. In scenario C the reconfiguration costs are of primary concern, but they are equivalent to the implementation costs of the tasks. Hence the independent optimization of the tasks serves both objectives at the same time. The preferred binding method for scenario A and C will be the same.

In scenario D the major aim is to reduce reconfiguration cost. Hence, the best implementation may be achieved if the intra-task and the inter-task resource reuse is considered in the binding method. Depending on the target architecture, the reconfiguration cost for either the resource instances or for the interconnect are dominant.

The scenarios A and C represent the established methods to realize HW tasks and static or reconfigurable modules. The scenarios B and D are available through our new methods.

The basic properties of the different scenarios are summarized in Table 5.4. The table also shows, which binding methods have been employed for each scenario in our experiments.

5.5.4 Benchmark Characteristics

Currently, there are no well established benchmarks designed for reconfigurable computing systems. Similarly to the area of HLS, the benchmarks are often taken from popular benchmark sets from the embedded systems and multimedia domain, e.g. the MediaBench suite [51]. The results that can be found in the literature often refer to some extracted, computing extensive kernels of those benchmarks. The exact definition of those kernels is usually not given. Moreover there is no common

Table 5.4: Properties of the scenarios A–D and the use of the binding methods.

Scenario		A	B	C	D
Concurrent Execution of Tasks		✓	–	–	–
Dynamic Control Memory possible		–	✓	✓	✓
Dynamic Datapath Reconfiguration		–	–	✓	✓
Reuse of Resources between Tasks		–	✓	–	✓
Type Binding Method	1	✓	✓	✓	✓
	2	✓	✓	✓	✓
	3	✓	✓	✓	✓
	4	–	✓	–	✓
	5	–	✓	–	✓
Instance Binding Method	1	✓	✓	✓	✓
	2	–	✓	–	–
	3	–	–	–	✓
	4	–	–	–	✓

method to measure of resource usage, timing, and reconfiguration cost. Hence, it is difficult to compare results to other work.

Instead we generate benchmark results with our HLS tool such that the results resemble the outcomes of established implementation methods (scenarios A,C). These results can be then rightfully compared to the results obtained with our new methodology (scenarios B,D).

The benchmark 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. In Table 5.5 the characteristics of the tasks are given in more detail. The table contains information on how many tasks are present in each task set and about the complexity of the CDFG of each task. These kind of tasks can be found in many similar work on HLS.

The benchmark ADPCM contains an ADPCM encoder and decoder from the MediaBench suite. EDGE contains three different Sobel edge detection filters: a combined horizontal and vertical filter, a horizontal only, and a vertical only filter. JPEG_DCT 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 CDFG. Finally the RGB_YUV describes a colour conversion from RGB colour space to the YUV colour space and vice versa, this function is used in many image and video coding applications.

In general, the all tasks are implemented such that one task is realized in one reconfigurable module, except for scenario B.

Table 5.5: Characteristics of the tasks.

Task Set	Task	Basic Blocks	Operations	Variables	Data Dependencies	Control States
ADPCM	adpcm_encode	35	126	24	199	101
	adpcm_decode	29	105	24	161	89
EDGE	sobel_hv	9	113	36	204	38
	sobel_h	9	100	33	177	38
	sobel_v	9	100	33	175	38
JPEG_DCT	jpeg_dct	6	178	43	378	120
	jpeg_idct	9	267	59	563	198
RGB_YUV	ycrcb2rgb	3	26	16	32	22
	rgb2ycrcb	3	24	13	28	24

5.5.5 Benchmark Results

Our HLS tool has been run on all task sets using the aforementioned binding methods. From the immediate results we computed the implementation and reconfiguration cost for each scenario. In this section we present the results in detail and provide an in-depth analysis. In order to prove the efficiency of our approaches we will answer the following questions regarding the proposed implementation scenarios and binding methods:

- What is the improvement achieved with the proposed binding methods?
- What are the benefits of the new implementation scenarios compared to previous approaches?
- What is the trade-off between resource requirements and reconfiguration cost in a reconfigurable implementation?

We will also investigate, which combination of binding methods yields the best quality of results and discuss the implications on the runtime of our prototype HLS tools.

The type binding methods have been developed with assumptions of the intra-task and inter-task reuse of resources. The benchmark data provide hints to what extend the assumed reuse is actually exploited during instance binding.

Analysis of Results

The benchmark results are analyzed for each scenario separately here in order to understand which binding method achieves the best results. Again, for the scenarios A and B only the implementation cost are relevant. In the scenarios C and

D the reconfiguration cost in terms of resources and interconnect are the primary optimization objective.

In the following figures, all results are plotted with the determined cost (C_A, C_B, C_C, C_D, R_C, R_D) at the y-axis. The x-axis is labelled with the pair of the used combination of *resource type binding, resource instance binding* method as it is given in Section 5.5.1.

Scenario A In scenario A all datapaths are implemented concurrently. We expected that if the DFG operations are bound to common module types, then intra-task reuse would lead to the most resource efficient datapath. For the selected benchmarks this could not be validated. Instead, type binding method 1 leads to the best overall results (cf. Figure 5.9). It can be seen that the differences in the results achieved with different binding methods are very small. It may be possible that the advantage of smaller and more specialized resource types selected by resource type method 1 outweighs the potential resource sharing that is enabled by the other resource type binding methods 2 and 3 with more general resource types.

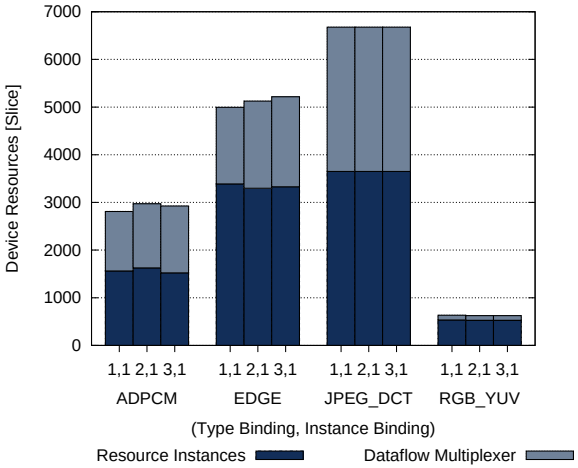


Figure 5.9: Resource requirements for both resource instances and dataflow multiplexers for the datapaths in scenario A.

Scenario B In scenario B all tasks are implemented in a single datapath. The results for the resource use in this scenario is shown in Figure 5.10. It can be observed that the resource use of the combined datapath is considerably reduced by using instance binding method 2, which takes advantage of the inter-task resource sharing.

The reduction is achieved to a large extend by reducing the datapath multiplexers. Hence, the method reuses the module instances such that the interconnect is reduced as well.

It seems that the impact of the chosen type binding method is relatively small, as already observed in scenario A.

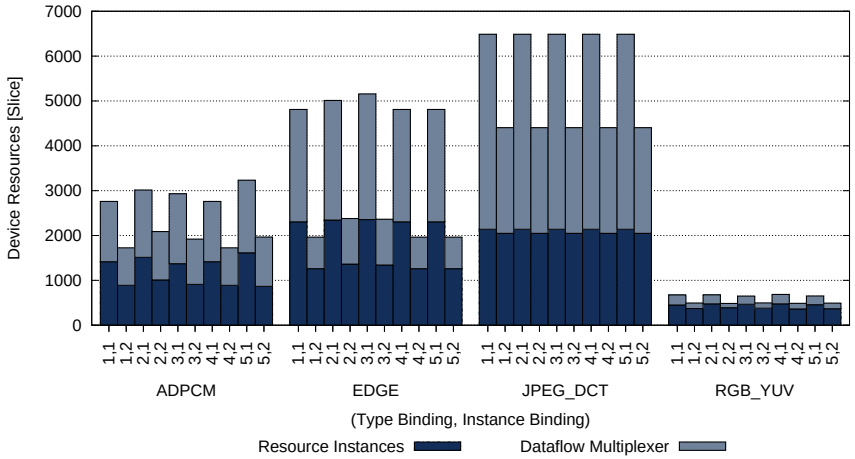


Figure 5.10: Resource requirements for both resource instances and dataflow multiplexers for the datapaths in scenario B.

Scenario C Full reconfiguration of the datapath is assumed in scenario C. In this scenario, only one reconfigurable module is configured in the FPGA at a time. This reduces the average resource use dramatically (cf. Figure 5.11(a)). In Figure 5.11(b) the number of reconfigurable Slices of the datapath is shown. Note that we included only the Slices associated with the resource instances here, for better comparison with scenario D. The amount multiplexer logic can be seen in Figure 5.11(a) as well.

Similar to scenario A, the chosen type binding method has very little influence on the overall results.

Scenario D In scenario D the datapath implementation is intended to minimize the reconfiguration cost. According to scenario C the instance binding method 1 achieves the lowest resource use of the datapath, but this is not the objective in this scenario. It serves as a starting point to evaluate the efficiency of our new instance binding methods.

The instance binding methods 1, 3, and 4 yield to very different results in both resource use and resource reconfiguration: Method 1 achieves the best results in

terms of resource use. With this method, there is already some unintended reuse of datapath resources, because the synthesis tool uses a common set of resource instances. This can be seen by comparing the number of Slices used for resource instances (Figure 5.12(a)) and the number of Slices that are reconfigurable (Figure 5.12(b)).

The number of slices for the reconfigurable modules is dramatically reduced when reconfiguration costs are the primary objective during instance binding (cf. method 3 and 4). Our results clearly show that only a small amount of Slices must be reconfigured when the datapath is adapted to different tasks. The improvement is more the 50 % compared to the unintended reuse of datapath resources, for the benchmarks ADPCM and EDGE it is even higher. If we compare the reconfigurable Slices to the overall use of Slices for the resource instances (cf. Figure 5.16(b)), it can be seen that only a small fraction of Slices (typically below 10 %) must be actually reconfigured in order to realize a new reconfigurable module on the device.

The differences in the results obtained for the instance binding methods 3 and 4 show a clear advantage for method 4. With instance binding method 4 we achieve better results in terms of device resource usage and reconfiguration of resources. In scenario D the type binding method has some influence on the results. Type binding method 5 achieves the best results in terms of reconfiguration costs whereas method 1 yields the lowest use of device resources.

The results are confirmed when the interconnect use and interconnect reconfiguration is considered (cf. Figure 5.13). The results clearly show the advantage of instance binding method 4. With this method a very low amount of reconfigurable interconnect with little overhead in the datapath wiring is obtained.

Datapath Control

Another measure to assess the quality of results is the length of the datapath control word, that is needed to control the resource instances and dataflow multiplexers. Thus, the control word length allows a relative comparison of the required memory in the control unit.

We observe that the control word length is directly correlated with the occupied datapath resources. Larger datapath implementations require wider control words. The results shown in Figure 5.14 support our statement. Note that the cost for datapath control has been included in the cost function, cf. Table 5.3. The number of control states is equal for all realizations.

Datapath Delay

In our experiments we used a commercial synthesis tool (Xilinx XST) to estimate the minimum delay of the datapath (cf. Figure 5.15). The delay estimation of the synthesized netlist is usually not very accurate, because the placement and routing

is not considered by the synthesis tool. The expected error in the estimation is about 10 % compared to the final implementation.

Our results show a similar signal delay for most combinations of the binding methods. The results are in the expected range of datapath delay for a Xilinx VirtexII architecture. It can be seen that the most complex application (JPEG_DCT) has the largest datapath delay (typically below 12 ns) and the less complex application (RGB_YUV) has a very low datapath delay (typically below 5 ns).

5.5.6 Discussion

For now we have studied the results obtained for each scenario in detail. Here we take the opportunity to summarize our findings and to answer the three major questions raised in the beginning of Section 5.5.5. In Figure 5.16(a–f), a comparison of results for all scenarios is shown.

In scenario B we have shown that a global optimization of the merged datapath can lead to a reduction of the datapath logic resources of up to 50 % compared to a naive reuse approach. The reduction is achieved by reusing operation modules as well as steering logic for the datapath control. As a result, the static datapath becomes much smaller compared to the parallel implementation of tasks at no cost for dynamic reconfiguration.

In scenario C the resource requirements are reduced compared to scenario A by using dynamic partial reconfiguration of the whole datapath. It allows to reduce the resource requirements of the datapath at the cost of a full reconfiguration of all involved logic and interconnect. However, in our examples the advantage in terms of resource requirements is small compared to scenario B.

With scenario D we have shown that the overhead in reconfiguration costs can be dramatically reduced if the datapath is optimized for reconfiguration. Our results show that essentially there is a very small amount of logic that must be reconfigured in order to change the functionality of the datapath.

Our results suggest that there is a large trade-off between the implementation costs and the reconfiguration costs. The instance binding methods 1, 3, and 4 result in two extremes of the implementation costs/reconfiguration costs plane. Method 1 optimizes only for implementation costs and achieves some reuse of logic and interconnect as a by-product, while the methods 3 and 4 optimize only for the reconfiguration costs and still achieve acceptable results for the datapath implementation costs. We are confident that our method can lead to many more attractive results between both extremes by choosing appropriate weights in the cost function.

The current design points that can be achieved with the chosen weights are shown in Figure 5.17. The x-axis corresponds to the relative resource use of a solution compared to the result of scenario C using type and instance binding method 1. The y-axis corresponds to the relative module reconfiguration cost computed

for the same setting. Note that the diagram contains all generated solutions, even the ones that are non-optimal in either metric. The results look very similar for all benchmarks. Scenario C results in solutions with highest reconfiguration cost, but low resource use and static scenario B yields solutions with a largest datapaths and no reconfiguration cost. Scenario D yields many solutions in between. It can be observed that solutions with lower reconfiguration cost tend to use only few more resources.

It is interesting that the resource advantage of dynamic reconfiguration is rather limited when compared to our approach of merged datapaths, even in the best case. However we expect that the results would be different if benchmarks with more reconfigurable tasks would be used.

In the following we would like to discuss the qualitative behaviour of the different cost metrics for larger task sets. We assume that the tasks within a set are of similar size. In scenario A the resource use and memory requirements will grow linearly with the number of tasks and the datapath delay remains constant. Scenario B will show a more moderate increase in resource use and memory requirements, because more and more functionality may be shared between tasks at the cost of additional interconnect. A similar trend has been discussed in Section 3.5.3: the expected qualitative increase will be similar to Figure 3.13(a) and 3.13(b). Merging more tasks into one datapath also means that the datapath becomes more complex and hence, the datapath delay will increase.

In scenario C the resource use, the memory requirements, and the average reconfiguration time will remain almost constant even if the number of tasks increases. What really increases is the memory footprint of the reconfiguration data. Thus, if more reconfigurable tasks are used then the cap in the resource demands between scenario A and scenario C, D will grow.

The trend in the cost metric for scenario D will be somewere between scenario A and C, depending on the chosen cost function parameters during binding. It is either possible to obtain solutions with low resource use similar to scenario C, but more reconfiguration cost in terms of Slices and interconnect—or to obtain a solution with more static elements but larger datapaths. In general we expect a growth of reconfiguration cost in scenario D when the number of tasks is increased, because the binding algorithms need to compromise the resource and interconnect reuse between more and more tasks. The behaviour has already been observed in Figure 3.14(a) and 3.14(b). A sketch of the general trends is shown in Figure 5.18.

In Section 5.2.2 we considered the adaptation of a datapath with different reconfiguration mechanisms. Now we show the implementation and reconfiguration cost trade-off that results for our benchmark applications.

We consider scenario B as a solution that achieves the adaptation of the datapath by runtime control. Scenario B achieves a single cycle control of the datapath functionality, but sacrifices more resources to implement the dataflow control. Moreover

a fraction of resource instances is only used by some of the tasks.

Scenario C realizes only the minimal datapath control that is necessary to run a single task. Device reconfiguration is used to adapt the datapath to a new task. In scenario D it is assumed that as many resources and interconnect of a datapath should be reused in a new reconfigurable module. Our results indicate that this can result in a less resource efficient datapath implementation compared to scenario C.

In principle it would be possible to generate datapath that are a hybrid solution between scenario B and scenario D: each reconfigurable module may contains some unused resource instances in order to reduce the overall reconfiguration cost. This solution does not seem reasonable for the presented examples, because the gap between the scenario B and the scenario D is small. The reconfiguration-optimized solution for scenario D requires almost as many resources as the merged datapath in scenario B.[2]

Finally, we want to analyze the results obtained for scenario D more critical. From the presented results and the discussion of Figure 5.18 it seems that the opportunities to take advantage of scenario D can be limited, especially when compared to scenario B. One may object for scenario D that:

1. Dynamic reconfiguration does decrease resource usage compared to a static implementation (scenario B) only to a limited extend.
2. The datapath implementation with maximum reuse between reconfigurable modules may require as many resources as a static implementation.
3. When the number of tasks increases, the reuse is likely to decrease, thus reconfiguration cost will increase and approach that of full reconfiguration (scenario C).

In the following we will discuss the objections raised above.

Objection (1) is actually caused by the results presented for the merged datapaths (scenario B). Before our study there was no direct comparison available for these two implementation scenarios. When the comparison between a static and dynamically reconfigurable implementation was made, the basis has always been scenario A and scenario C. In this work we offer the merged datapath implementation as a design point which in fact narrows the gap between static and reconfigurable solutions. Still, scenario B has disadvantages compared to a dynamically reconfigurable solution: the datapath resource and memory requirements increase with the number of merged tasks and—due to increased complexity of the merged datapath–the maximum achieveable clock frequency is less than those for a smaller, dynamically reconfigurable implementation.

Regarding objection (2), the cost function used in scenario D results in the solution with the least possible reconfiguration cost. There exist many solutions in

[2]Note that scenario D is not optimized for resource use.

between, which may achieve a different trade-off between implementation and reconfiguration cost.

The efficiency of the resource reuse raises objection (3). As a solution, we propose that our methods should be used such that they best fit the application. Some general considerations have been discussed in Section 5.2. A real reconfigurable application provides several aspects that can be exploited. At first, reconfiguration is not required between all tasks. Thus, the binding does not need to produce a datapath that decreases the reconfiguration cost between any two tasks, but only between tasks that are reconfigured against each other. Second, not all tasks have the same potential for resource reuse. Third, the datapath of the tasks can differ significantly in size. Hence there are tasks that can be merged into the same datapath at a small increase in resource cost only and others that can not. The merging of some tasks can be performed until the limit of available device resources is exceeded. The technique reduces the number of necessary device reconfigurations and further allows to balance the size of the different reconfigurable modules, thus avoiding internal fragmentation.

In summary, our methods provide a solution for the implementation of reconfigurable systems that give the best possible balance between resource requirements and cost for dynamic reconfiguration.

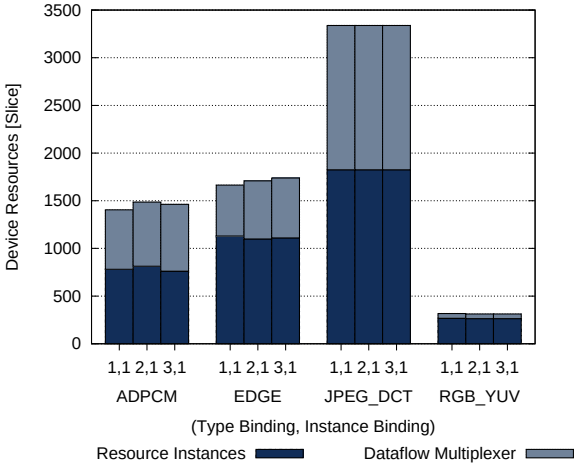
5.6 Summary

In this chapter we have described new methods for HLS that consider the reconfiguration cost during the implementation of tasks. We presented our tool flow, which is able to perform HLS of C-based functions to reconfigurable modules. The reconfigurable modules are based on an architecture template that provides several possibilities to integrate runtime device reconfiguration. We developed new concepts how runtime reconfiguration can be applied to these modules. We proposed the realization of multiple tasks in one reconfigurable module. Furthermore we proposed the concept of multi-level reconfiguration, which is based on the idea that as many parts as possible are reused in different reconfigurable modules. The concepts result in different implementation and reconfiguration cost trade-offs.

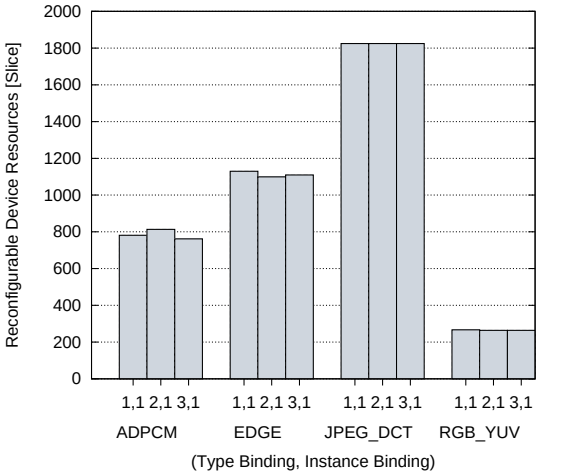
In order to facilitate the reuse between modules, we integrated our reconfiguration cost model and the virtual architecture model in a newly developed HLS tool. We proposed several methods on how to perform resource type binding and resource instance binding. The objective of these methods is the synthesis of datapaths that result in reconfigurable modules with minimal reconfiguration costs. Therefore we proposed a resource type binding that considers all tasks at the same time in order to select common resource types and common interconnect types for all tasks. These common types enable an efficient resource instance binding step. Here, we proposed the use of our virtual architecture model in order to perform a

simultaneous binding for all tasks. Our reconfiguration cost model allows an optimization of the resource instance binding in terms of reconfiguration cost. The result of our HLS tool are reconfigurable modules that exploit the reuse of datapath resources as much as possible and thus, can be reconfigured at low cost.

Finally we have performed several experiments with our HLS tool. We have shown that our new method provides a significant advantage over established approaches. Our results prove that our HLS tool can be used to implement more efficient datapaths: one method allows the realization of many tasks in one static module, without the need for dynamic reconfiguration. A second method realizes datapaths in reconfigurable modules that achieve very low reconfiguration costs.

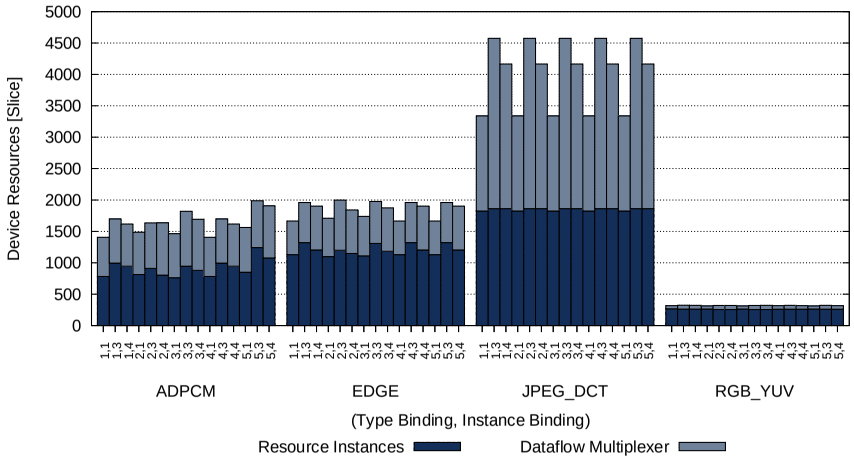


(a) Resource requirements for both resource instances and dataflow multiplexers.

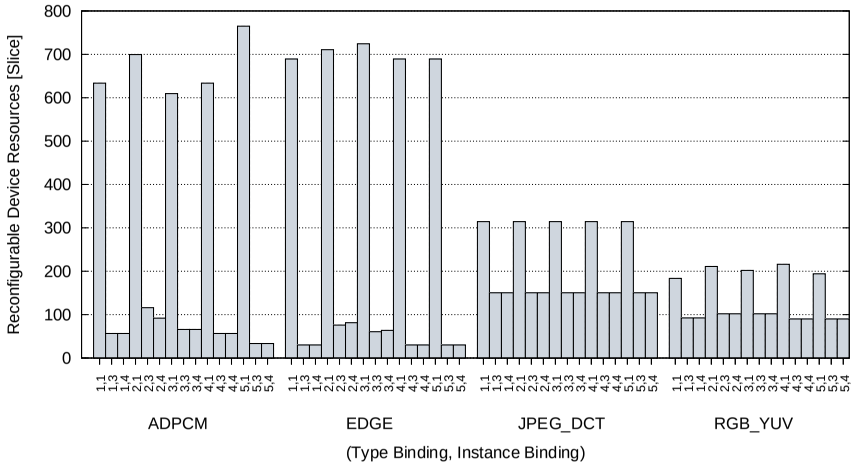


(b) Reconfiguration cost assuming full reconfiguration of resource instances.

Figure 5.11: Results of the datapath implementation using scenario C.

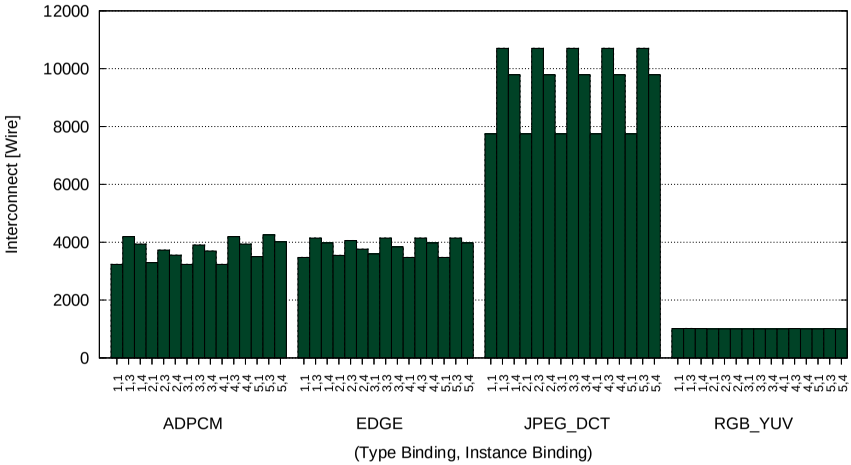


(a) Resource requirements for both resource instances and dataflow multiplexers.

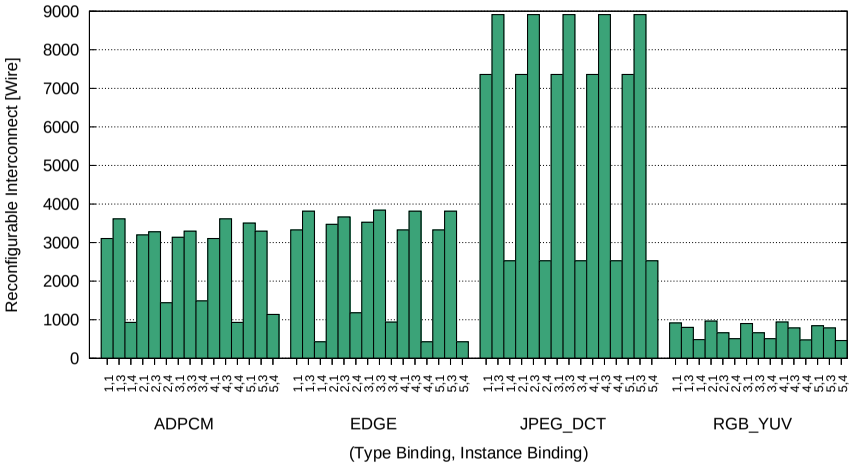


(b) Reconfiguration cost assuming partial reconfiguration of resource instances.

Figure 5.12: Results of the datapath implementation using scenario D.



(a) Interconnect wires.



(b) Reconfigurable interconnect wires.

Figure 5.13: Number of datapath interconnect and reconfigurable interconnect in scenario D.

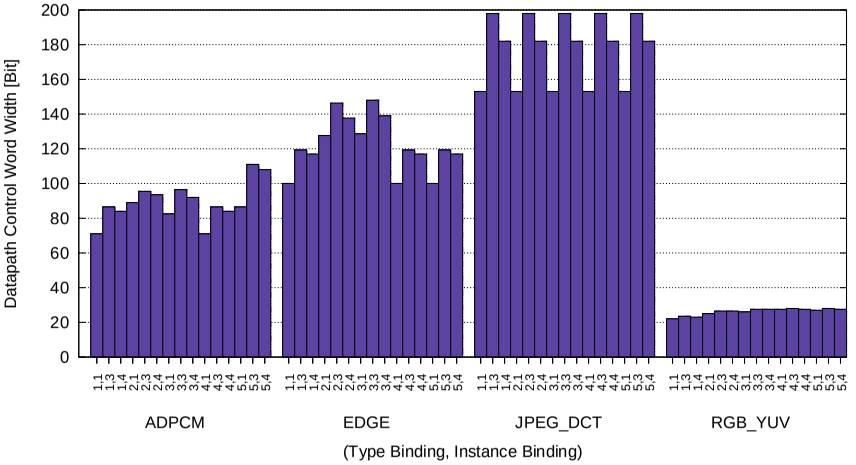


Figure 5.14: The average width of the datapath control word in scenario D.

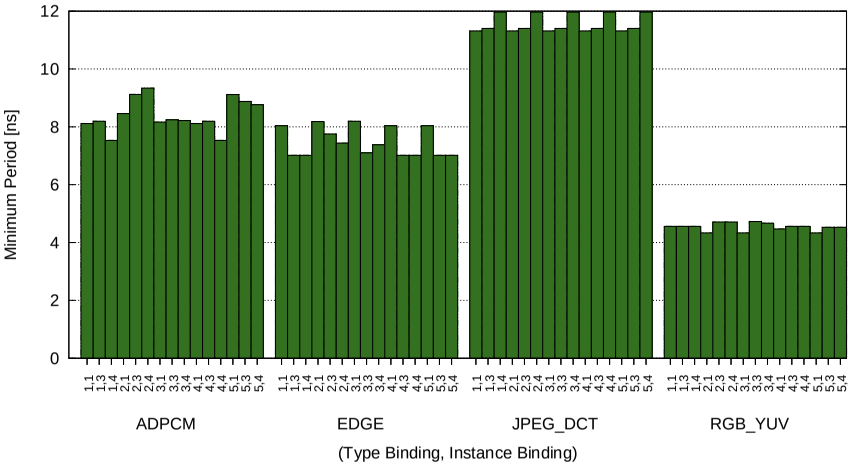
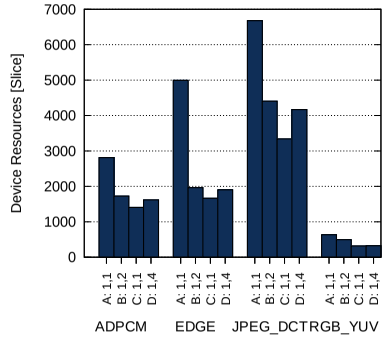
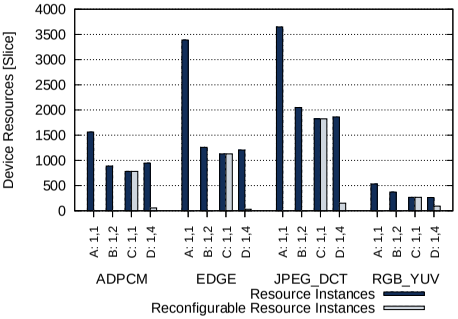


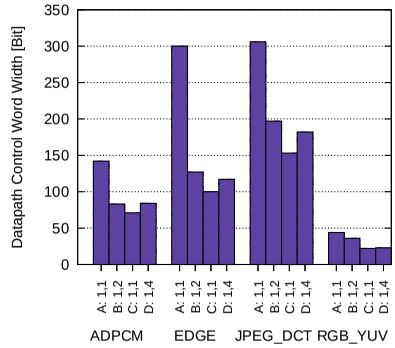
Figure 5.15: Minimum possible clock period of the datapath in scenario D. Lower values correspond to higher clock frequencies.



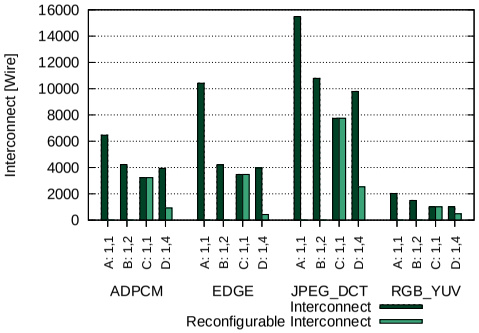
(a) Module instances and dataflow multiplexers



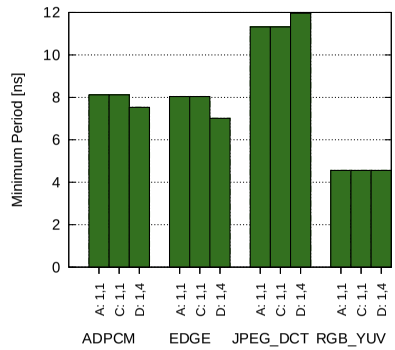
(b)



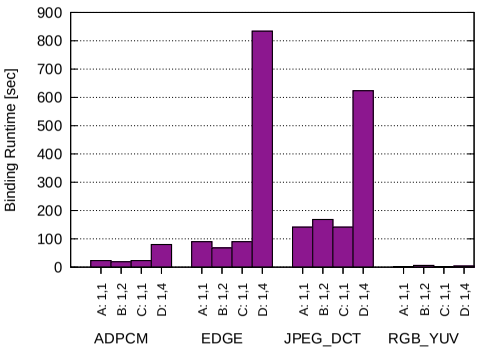
(c)



(d)



(e)



(f) Algorithm runtime for type binding and instance binding.

Figure 5.16: Comparison of results obtained with selected binding methods for scenarios A–D.

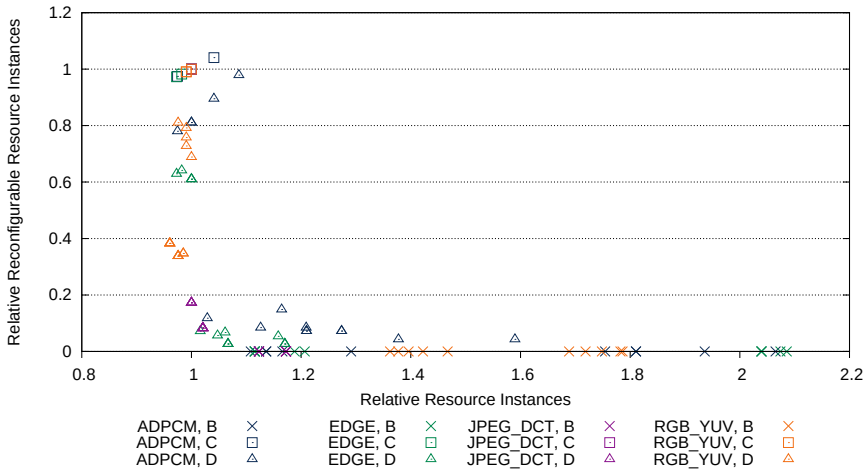


Figure 5.17: Implementation and reconfiguration cost trade-offs discovered by our instance and type binding methods. The data of each application was scaled by the solution computed for scenario C, type and instance binding method 1.

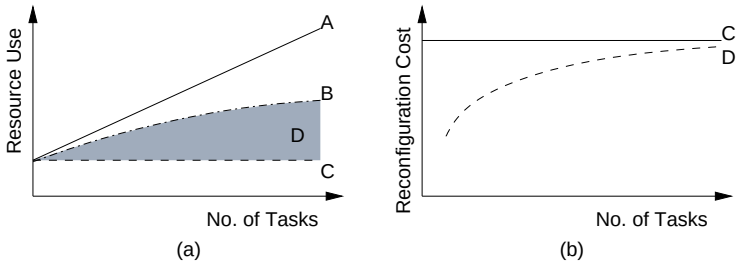


Figure 5.18: Expected utilization of device resources and reconfiguration cost for an increasing number of tasks for the scenarios A–D.

Chapter 6

Summary and Outlook

In this work we described several methods to reduce the reconfiguration overhead in FPGAs. The reconfiguration overhead has a severe impact on the efficiency of reconfigurable systems-on-a-chip. In fact, the gain obtained by the utilization of formerly temporary-unused FPGA resources is degraded by the required reconfiguration time and the memory footprint of configuration data. We developed several design and implementation methods that can be used to reduce reconfiguration time and configuration data by increasing the similarity between configurations.

In previous works, it is frequently assumed that reconfiguration overhead depends on the frequency of runtime reconfiguration and on the size of the reconfigurable area. This is a general assumption made in scheduling and placement algorithms. Other researchers observed that reconfiguration overhead can be reduced, when the similarities between tasks are considered. However, there has been no general approach to identify the similarity between configurations. Furthermore there has been no unified model that links the similarity between configurations with the reconfiguration overhead. In this work we have shown that both the similarity and the reconfiguration overhead are highly interrelated.

We established a reconfiguration cost model, which is based on the reconfiguration state graph. The reconfiguration state graph reflects the fact that reconfiguration is performed between predefined configurations. Our reconfiguration cost model takes into account the individual cost for the reconfiguration of individual reconfigurable elements. Thus the reconfiguration cost model assumes a fine grain reconfiguration of reconfigurable elements instead of the full reconfiguration of a reconfigurable module. We have shown that there is a distinct difference between the cost associated with reconfiguration time and configuration data. Our model can be directly applied to compute the reconfiguration cost for binary configuration data.

We have introduced a virtual architecture model in order to be able to assess the reconfiguration cost for structural representations, too. The aim of this model is to establish a relation between the elements of the structural representations. The re-

lation is then used to evaluate the reconfiguration cost at structural level. Therefore the reconfigurable elements of the structural representations are allocated to a elements of a virtual architecture. We introduced a new approach for the optimization of reconfiguration cost that can be applied to all several levels in the design flow using structural representations. The inherent relationship between the mapping to a virtual architecture and the reconfiguration cost has been used to find optimal allocations that minimize reconfiguration cost. Our methodology requires a new class of implementation constraints to be integrated in the implementation tools. Reconfigurable elements of different configurations that are allocated to the same resource in the virtual architecture must be assigned to the same resource in the FPGA, too.

We presented several general optimization methods that can be used to compute an allocation of reconfigurable elements such that the reconfiguration cost are minimal. In an initial experimental investigation we have shown that the reconfiguration time and the configuration data can be reduced considerably with an optimized allocation.

The methodology has been refined and adapted to the specific requirements for the mapping of synthesized netlists to device specific netlists. We were able to show that the estimated reconfiguration time for interconnect could be reduced to less than 28 % for synthesized netlists, when a fine grained reconfiguration is assumed. Our mapping tool could improve these results even further: for the device specific netlists, the reconfiguration costs are less than 17 % compared to a full reconfiguration of interconnect.

Another major contribution of this work is the development of a high-level synthesis tool for reconfigurable modules. We described how the reconfigurable modules can be realized such that they can be reconfigured at low cost. Therefore we proposed to integrate the functionality of several hardware tasks in one module and the reconfiguration of the modules at different levels. We have developed several methods for the optimized synthesis of the reconfigurable modules' datapaths. For this purpose we applied our virtual architecture model and the reconfiguration cost function to the high-level synthesis process. With a number of examples we have shown that our tool delivers cost efficient solutions that could not be obtained by previous methods. The integration of multiple hardware tasks results in compact, but flexible modules that are very resource efficient compared to previous solutions. No device reconfiguration is required between the integrated hardware tasks. Our tool can also synthesize datapaths that are optimized in terms of reconfiguration costs. We have shown that the resulting datapaths have a very similar structure. The reconfiguration time for resources is typically less than 10 % and for interconnect it is typically less than 26 % compared to the results obtained with existing methods. Moreover we have illustrated that there exists a trade-off between implementation cost and reconfiguration cost. This trade-off can be exploited with our tool in order to meet the requirements of the application.

In Figure 6.1 it is illustrated how our tools can be integrated into a complete RSOC design flow.

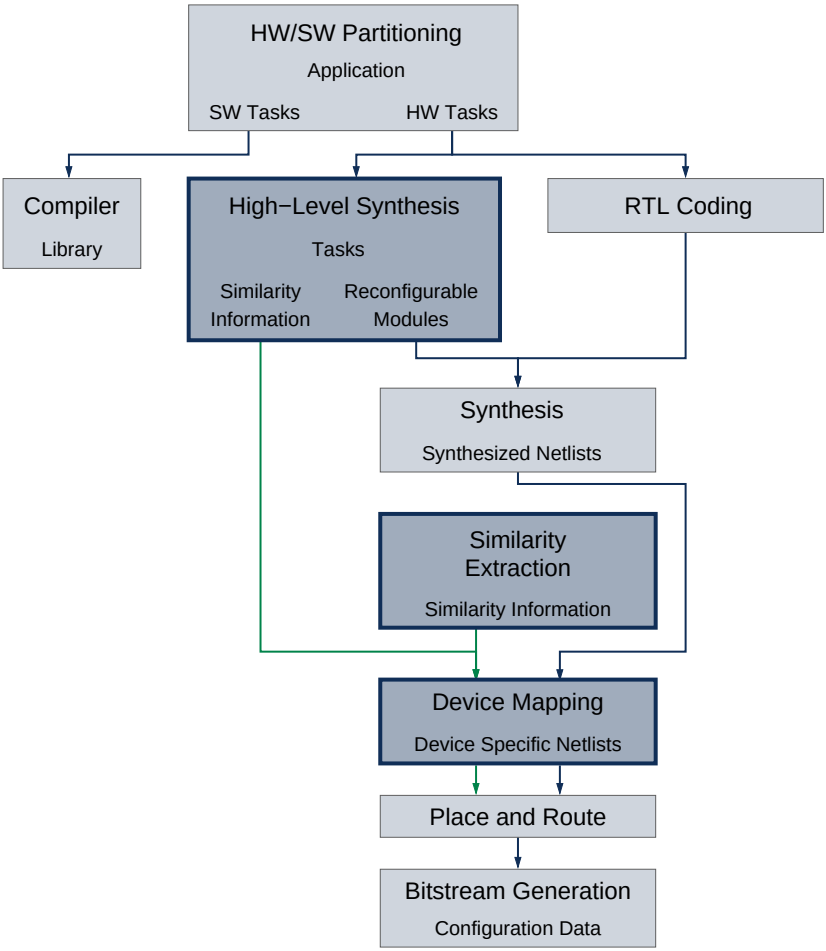


Figure 6.1: The proposed design flow. The highlighted design steps have been described in this work.

In hardware/software partitioning the application is divided into software and hardware tasks, which is done usually manually by a system design expert. The software tasks are implemented with a programming language and compiled into a software library.

The hardware task are implemented either manually in an RTL coding step or with an high-level synthesis tool from a behavioural description. Such a tool is presented in this work. At present, the designer chooses what tasks are integrated into what reconfigurable modules and between which reconfigurable modules dynamic reconfiguration is required. However, in future work this decision could be part of the optimization in the tool. Furthermore it might be possible to divide tasks automatically into different reconfigurable modules in order to adhere to tight resource constraints.

Our high-level synthesis tool provides detailed similarity information to the device mapping tool. For manually implemented RTL code, these information can be obtained with our tool for similarity extraction. Our device mapping tool takes advantage of the provided similarity information.

In the future, currently available place-and-route tools must be extended in order to respect the placement constraints generated by our device mapping tool. If this has been done, then the effect of our design flow on reconfiguration cost can be observed for binary configuration data.

With the completion of this work, we established a unique model for the optimization and evaluation of reconfiguration cost at all levels of the design flow. Although there remain some minor modifications to existing tools to be done, our tools provide the key functionality that is required for future developments.

Bibliography

[1] *Handel-C Language Reference Manual*, [Online], Agility Design Solutions Inc., 2008, www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf.

[2] A. Ahmadinia, C. Bobda, and J. Teich, "A dynamic scheduling and placement algorithm for reconfigurable hardware," in *Workshop on Organic and Pervasive Computing at International Conference Architecture of Computing Systems (ARCS 2004)*, ser. Lecture Notes in Computer Science, vol. Volume 298. Heidelberg, Germany: Springer, February 2004, pp. 443–465.

[3] "www.altera.com," [Online], Altera Inc., 2009, http://www.altera.com.

[4] J. Angermeier, M. Majer, J. Teich, L. Braun, T. Schwalb, P. Graf, M. Hübner, J. Becker, E. Lubbers, M. Platzner, C. Claus, W. Stechele, A. Herkersdorf, M. Rullmann, and R. Merker, "SPP1148 booth: Fine grain reconfigurable architectures," in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, Heidelberg, Germany, September 2008, p. 348.

[5] J. Angermeier and J. Teich, "Heuristics for scheduling reconfigurable devices with consideration of reconfiguration overheads," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, Miami, FL, April 2008, pp. 1–8.

[6] D. Aravind and A. Sudarsanam, "High level - application analysis techniques & architectures - To explore design possibilities for reduced reconfiguration area overheads in FPGAs executing compute intensive applications," in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, April 2005.

[7] L. Babel, "A fast algorithm for the maximum weight clique problem," *Computing*, vol. 52, no. 1, pp. 31–38, March 1994.

[8] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP - a self-reconfigurable data processing architecture," *The Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, September 2003.

[9] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, Jan–Mar 2000.

[10] C. Bobda and A. Ahmadinia, "Dynamic interconnection of reconfigurable modules on reconfigurable devices," *IEEE Journal on Design and Test of Computers,*, vol. 22, no. 5, pp. 443–451, Sept.–Oct. 2005.

[11] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, algorithms and applications*. Dordrecht, The Netherlands: Springer, 2007.

[12] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, J. Teich, S. P. Fekete, and J. van der Veen, "The Erlangen Slot Machine: A highly flexible FPGA-based reconfigurable platform," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005)*, Napa, CA, April 2005, pp. 319–320.

[13] M. Boden, T. Fiebig, M. Reiband, P. Reichel, and S. Rülke, "GePaRD - A high-level generation flow for partially reconfigurable designs," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '08)*, Montpellier, France, April 2008, pp. 298–303.

[14] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar, "Mapping a single assignment programming language to reconfigurable systems," *The Journal of Supercomputing*, vol. 21, no. 2, pp. 117–130, February 2002.

[15] G. Brebner, "A virtual hardware operating system for the Xilinx 6200," *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, (FPL 1996)*, vol. 1142, pp. 327–336, September 1996.

[16] G. Chen, M. Kandemir, and U. Sezer, "Configuration-sensitive process scheduling for FPGA-based computing platforms," in *Design, Automation and Test in Europe Conference and Exhibition (DATE 2004)*, Paris, France, 2004, pp. 486–493.

[17] C. Claus, F. Müller, and W. Stechele, "Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro devices," in *Workshop on Dynamically Reconfigurable Systems at the International Conference on Architecture of Computing Systems (ARCS 2006)*, ser. GI Lecture Notes in Informatics, March 2006, pp. 122–131.

[18] C. Claus, J. Zeppenfeld, F. Müller, and W. Stechele, "Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance

system," in *Design, Automation and Test in Europe Conference and Exhibition (DATE 2007)*, Nice, France, April 2007, pp. 1–6.

[19] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hübner, and J. Becker, "A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, Heidelberg, Germany, Sept. 2008, pp. 535–538.

[20] C. Claus, F. H. Müller, J. Zeppenfeld, and W. Stechele, "A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, March 2007, pp. 1–7.

[21] K. Compton and S. Hauck, "Configurable computing: A survey of systems and software," Department of Electrical and Computer Engineering, Northwestern University, USA; Department of Electrical Engineering, University of Washington, USA, Tech. Rep., 1999.

[22] "Cray XD1 supercomputer," http://www.cray.com, Cray Inc., 2004.

[23] A. Dandalis and V. Prasanna, "Configuration compression for FPGA-based embedded systems," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 13, no. 12, pp. 1394–1398, December 2005.

[24] A. DeHon, "Reconfigurable architectures for general-purpose computing," Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Tech. Rep. 1586, 1996.

[25] O. Diessel and H. ElGindy, "On scheduling dynamic FPGA reconfigurations," in *The 5th Australasian Conference on Parallel and Real-Time Systems (PART '98)*. Adelaide, Australia: Springer Verlag, September 1998, pp. 191–200.

[26] F. Dittmann and S. Frank, "Caching in real-time reconfiguration port scheduling," in *International Conference on Field Programmable Logic and Applications (FPL 2007)*, Amsterdam, The Netherlands, August 2007, pp. 740–744.

[27] ——, "Hard real-time reconfiguration port scheduling," in *Design, Automation and Test in Europe Conference and Exhibition (DATE 2007)*, Nice, France, April 2007, pp. 1–6.

[28] EIA/EDIF, "Edif version 2 0 0," ANSI/EIA Standard 548-1988, March 1988.

[29] G. Estrin, "Organization of computer systems - The fixed plus variable structure computer," in *Western Joint Computer Conference*, New York, NY, USA, 1960, pp. 33–40.

[30] C. Fraser and D. Hansen, *A Retargetable C Compiler: Design and Implementation*, C. Fraser and D. Hansen, Eds. Addison Wesley, 1995.

[31] C. W. Fraser, "A retargetable compiler for ANSI C," *SIGPLAN Notices*, vol. 26, no. 10, pp. 29–43, 1991.

[32] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005)*, Napa, CA, April 2005, pp. 149–158.

[33] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2000)*, Napa Valley, CA, April 2000, pp. 49–56.

[34] M. Gokhale and P. S. Graham, *Reconfigurable computing*, M. Gokhale, Ed. Springer, 2005.

[35] M. Gotz and F. Dittmann, "Reconfigurable microkernel-based RTOS: Mechanisms and methods for run-time reconfiguration," in *IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, San Luis Potosi, , Mexico, September 2006, pp. 1–8.

[36] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," in *Design, Automation and Test in Europe Conference and Exhibition (DATE 2001)*, Munich, Germany, 2001, pp. 642–649.

[37] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment within large apertures," in *8th Workshop on Design automation (DAC '71)*. New York, NY, USA: ACM, 1971, pp. 155–169.

[38] S. Hauck, Z. Li, and E. Schwabe, "Configuration compression for the Xilinx XC6200 FPGA," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 1998)*, Napa Valley, CA, April 1998, pp. 138–146.

[39] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," in *ACM/SIGDA sixth International Symposium on Field Programmable Gate Arrays (FPGA 1998)*, Monterey, CA, 1998, pp. 65–74.

[40] J. Heron, R. Woods, S. Sezer, and R. Turner, "Development of a run-time reconfiguration system with low reconfiguration overhead," *The Journal of VLSI Signal Processing*, vol. 28, no. 1–2, pp. 97–113, May 2001.

[41] H. Hinkelmann, P. Zipf, and M. Glesner, "A metric for the energy-efficiency of dynamically reconfigurable systems," in *Workshop on Dynamically Reconfigurable Systems at the 19th International Conference on Architecture of Computing Systems (ARCS '06)*, Frankfurt am Main, Germany, March 2006.

[42] Z. Huang and S. Malik, "Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks," in *Design, Automation and Test in Europe Conference and Exhibition (DATE 2001)*, Munich, Germany, March 2001, pp. 735–740.

[43] M. Hübner, C. Schuck, and J. Becker, "Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 2006.

[44] M. Hübner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic FPGA self-reconfiguration," in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, April 2004.

[45] D. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the I.R.E.*, September 1952, pp. 1098–1102.

[46] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "REPLICA: A bitstream manipulation filter for module relocation in partial reconfigurable systems," in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, April 2005.

[47] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. New York, NY, USA: Springer, 1972, pp. 85–103.

[48] I. Kennedy, "Exploiting redundancy to speedup reconfiguration of an FPGA," in *13th International Conference on Field-Programmable Logic and Applications (FPL 2003)*, ser. LNCS, vol. 2778, Lisbon, Portugal, September 2003, pp. 262–271.

[49] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, 4598, pp. 671–680, May 1983. [Online]. Available: http://citeseer.ist.psu.edu/kirkpatrick83optimization.html

[50] D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich, "A highly parameterizable parallel processor array architecture," in *IEEE International Conference on Field Programmable Technology (FPT 2006)*. Bangkok, Thailand: IEEE, December 2006, pp. 105–112.

[51] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "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, December 1997, pp. 330–335.

[52] Z. Li and S. Hauck, "Configuration compression for Virtex FPGAs," in *9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, Rohnert Park, CA, 2001, pp. 147–159.

[53] M. Majer, J. Angermeier, and J. Teich, "Main page - ESM wiki," [Online], 2009, http://www12.informatik.uni-erlangen.de/esmwiki.

[54] U. Malik and O. Diessel, "On the placement and granularity of FPGA configurations," in *IEEE International Conference on Field-Programmable Technology (FPT2004)*, Brisbane, Australia, December 2004, pp. 161–168.

[55] ——, "The entropy of FPGA reconfiguration," in *International Conference on Field Programmable Logic and Applications (FPL '06)*, Madrid, Spain, August 2006, pp. 1–6.

[56] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, and A. DeHon, "Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine," in *ACM/SIGDA tenth international symposium on Field-programmable gate arrays (FPGA '02)*. Monterey, CA: ACM, 2002, pp. 196–205.

[57] A. S. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in *ACM/SIGDA seventh International Symposium on Field Programmable Gate Arrays (FPGA '99)*. New York, NY, USA: ACM Press, 1999, pp. 37–46.

[58] "Arrix FPOA overview," [Online], MathStar Inc., 2008, http://mathstar.com/Documentation/Documentation0408/FPOA_Overview_REL_Final_v1.7.pdf.

[59] "Catapult C synthesis," [Online], Mentor Graphics, 2009, www.mentor.com/c-based_design.

[60] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[61] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 969–980, July 2005.

[62] "FSB development platform - overview," [Online], Nallatech Inc., 2008, http://www.nallatech.com.

[63] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," in *International Conference on Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003.

[64] J.-B. Note and E. Rannaud, "From the bitstream to the netlist," in *16th international ACM/SIGDA symposium on Field programmable gate arrays (FPGA '08)*. New York, NY, USA: ACM, 2008, pp. 264–264.

[65] T. Oppold, T. Schweizer, T. Kuhn, and W. Rosenstiel, "Cost functions for the design of dynamically reconfigurable processor architectures," in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, Kanazawa, Japan, 2004, pp. 443–450.

[66] *XPP-III Processor Overview*, [Online], PACT XPP Technologies Inc., 2006, http://www.pactxpp.com/main/download/XPP-III_overview_WP.pdf.

[67] J. H. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD 2004)*, San Jose, CA, November 2004, pp. 766–773.

[68] T. Pionteck, T. Staake, T. Stiefmeier, L. Kabulepa, and M. Glesner, "Design of a reconfigurable AES encryption/decryption engine for mobile terminals," in *International Symposium on Circuits and Systems (ISCAS '04)*, vol. 2, Vancouver, Canada, May 2004, pp. II–545–8.

[69] M. Platzner, N. Wehn, and J. Teich, Eds., *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*. Springer, 2009.

[70] K. P. Raghuraman, H. Wang, and S. Tragoudas, "A novel approach to minimizing reconfiguration cost for LUT-based FPGAs," in *18th International Conference on VLSI Design, 2005.*, Kolkata, India, January 2005, pp. 673–676.

[71] D. Rakhmatov and S. B. K. Vrudhula, "Minimizing routing configuration cost in dynamically reconfigurable FPGAs," in *15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, San Francisco, CA, April 2001, pp. 1481–1488.

[72] M. Reiband, "Optimierte netzlistengenerierung bei der high-level-synthese anhand der layoutvorgaben für dynamisch rekonfigurierbare FPGAs," Master's thesis, Technische Universität Dresden, July 2007.

[73] J. Resano, D. Mozos, D. Verkest, F. Catthoor, and S. Vernalde, "Specific scheduling support to minimize the reconfiguration overhead of dynamically

reconfigurable hardware," in *41st Design Automation Conference 2004 (DAC 2004)*, San Diego, CA, 2004, pp. 119–124.

[74] M. Rullmann and R. Merker, "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, March 2006, pp. 132–141.

[75] ——, "Maximum edge matching for reconfigurable computing," in *Reconfigurable Architectures Workshop at 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes, Greece, April 2006.

[76] ——, "A reconfiguration aware circuit mapper for FPGAs," in *Reconfigurable Architectures Workshop at the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), 14th*, 2007.

[77] ——, "A cost model for partial dynamic reconfiguration," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS)*, W. Najjar and H. Blume, Eds., Samos, Greece, July 2008, pp. 182–186.

[78] ——, "Synthesis of efficiently reconfigurable datapaths for reconfigurable computing," in *International Conference on Field-Programmable Technology 2008 (ICFPT '08)*, Taipeh, Taiwan, December 2008, pp. 277–280.

[79] ——, *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*. Springer, 2009, ch. Design Methods and Tools for Improved Partial Dynamic Reconfiguration, Marco Platzner and Norbert Wehn and Jürgen Teich eds.

[80] M. Rullmann, R. Merker, H. Hinkelmann, P. Zipf, and M. Glesner, "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)*, Prague, Czeck Republic, September 2009.

[81] M. Rullmann, S. Siegel, and R. Merker, "Optimization of reconfiguration overhead by algorithmic transformations and hardware matching," in *Reconfigurable Architectures Workshop at the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, April 2005, pp. 151–156.

[82] S. Sezer, J. Heron, R. Woods, R. Turner, and A. Marshall, "Fast partial reconfiguration for FCCMs," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 1998)*, Napa Valley, CA, April 1998, pp. 318–319.

[83] "SGI RASC RC100 blade," [Online], SGI, 2008, http://www.sgi.com/pdfs/3920.pdf.

[84] N. Shirazi, W. Luk, and P. Cheung, "Automating production of run-time reconfigurable designs," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 1998)*, Napa Valley, CA, April 1998, pp. 147–156.

[85] "www.siliconbluetech.com," [Online], SiliconBlue Technologies Corporation, 2009, http://www.siliconbluetech.com/.

[86] R. Stefan and S. Cotofana, "Bitstream compression techniques for Virtex 4 FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, Heidelberg, Germany, September 2008, pp. 323–328.

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

[88] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal of the ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[89] T. Stützle and H. H. Hoos, "MAX-MIN Ant system," *Future Generation Computer Systems*, vol. 16, no. 9, pp. 889–914, 2000.

[90] H. Tan and R. F. DeMara, "A physical resource management approach to minimizing FPGA partial reconfiguration overhead," in *IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, San Luis Potosi, Mexico, September 2006, pp. 1–5.

[91] J. Teich, *Digitale Hardware/Software-Systeme. Synthese und Optimierung*. Springer-Verlag, Berlin Heidelberg New York, 1997.

[92] J. Teich, S. P. Fekete, and J. Schepers, "Optimization of dynamic hardware reconfigurations," *The Journal of Supercomputing*, vol. 19, no. 1, pp. 57–75, May 2001.

[93] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi, and L. Jing, "High-level synthesis challenges and solutions for a dynamically reconfigurable processor," in *IEEE/ACM international conference on Computer-aided design (ICCAD '06)*. New York, NY, USA: ACM, 2006, pp. 702–708.

[94] M. Ullmann, M. Hübner, B. Grimm, and J. Becker, "An FPGA run-time system for dynamical on-demand reconfiguration," in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, April 2004.

[95] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*, D. Soudris, Ed. Springer, 2007.

[96] H. Walder and M. Platzner, "Online scheduling for block-partitioned reconfigurable devices," in *Design, Automation and Test in Europe Conference and Exhibition (DATE 2003)*, Munich, Germany, 2003, pp. 290–295.

[97] A. Weder, M. Rullmann, and R. Merker, "Ein Linux-basiertes, dynamisch rekonfigurierbares hardware-softwaresystem auf basis der Xilinx ML300 plattform," in *Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS 2005)*, Dresden, Germany, April 2005.

[98] J. Williams and N. Bergmann, "Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip," in *International Conference on Engineering of Reconfigurable Systems and Algorithms*. Las Vegas, Nevada: CSREA Press, June 2004, pp. 163–169.

[99] M. J. Wirthlin and B. L. Hutchings, "Improving functional density using run-time circuit reconfiguration," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 6, no. 2, pp. 247–256, 1998.

[100] M. Wirthlin and B. Hutchings, "A dynamic instruction set computer," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 1995)*, Napa Valley, CA, April 1995, pp. 99–107.

[101] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[102] *XC6200 Field Programmable Gate Arrays Product Description*, Xilinx Inc., San Jose, CA, 1997.

[103] *Xapp151: Virtex Series Configuration Architecture User Guide*, Xilinx Inc., 1999.

[104] *Xilinx Virtex 2.5 V Field Programmable Gate Arrays*, Xilinx Inc., San Jose, CA, 2001.

[105] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, Xilinx Inc., 2002.

[106] *Xapp290 – Two Flows for Partial Reconfiguration: Module Based or Difference Based*, Xilinx Inc., September 2004.

[107] *Virtex-II Platform FPGA User Guide*, Xilinx Inc., March 2005.

[108] *Early Access Partial Reconfiguration User Guide*, [Online], Xilinx Inc., 2006, http://www.xilinx.com/support/prealounge/protected.

[109] "JBits 3.0 SDK," [Online], Xilinx Inc., 2009, http://www.xilinx.com/labs/projects/jbits.

[110] "www.xilinx.com," [Online], Xilinx Inc., 2009, http://www.xilinx.com.

[111] "Xilinx university program Virtex-II Pro development system," [Online], Xilinx Inc., 2009, http://www.xilinx.com/products/devkits/XUPV2P.htm.

[112] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

Appendix A

Simulated Annealing

Simulated annealing (SA) is a well known heuristic to solve complex optimization problems. The implementation used in this work was inspired by Reiband [72]. The SA itself was originally described by Kirkpatrick [49]. The main control loop of our SA implementation is given in Algorithm 8. The aim of this algorithm is to find a solution s of the optimization problem such that the cost c of this solution is minimal.

The general concept of SA can be stated as follows: The solution s is initialized to an initial solution s_0, which is the starting point $s = s_0$ of the following iterations. At first, another solution s', which is close to the current solution s, is obtained by modifying s randomly. Then, the algorithm decides randomly whether to move to the new solution by setting $s := s'$ or not. The probability of moving to the new solution depends on the cost c' of the new solution s' and on the progress that the SA has already made. The globally best solution that occurs during the iterations is saved in s_b.

Any SA implementation has to determine several details that depend on the optimization problem:

1. How to create an initial solution s_0 ($s_0 = \mathit{getInitialSolution}()$)?
2. How to generate a new solution s' based on the current solution s ($s' = \mathit{PermuteSolution}(s)$)?
3. How to compute the cost function ($c = \mathit{getSolutionCost}(s)$)?
4. When to accept a new solution?
5. How to update the acceptance criteria?
6. When is the SA terminated?

The items 1–3 are problem dependent and are discussed in the appropriate sections in this work. However, the algorithm control is more general and is described in the following.

The acceptance criteria of the SA is controlled by a temperature t. The annealing is started with an initial temperature t_0. A robust approach to calculate t_0 is to run

a number t_{init} of initial iterations, in which every move to a new state is accepted. The average cost for all such generated solutions is used as t_0. During SA iterations, the temperature decreases according to a fixed schedule: every t_{const} iterations, the temperature t is decreased by a constant factor α, see line 42:

$$t := \alpha t \tag{A.1}$$

The decision, whether s' is accepted as a new solution s depends on the temperature t and the costs c, c' of the solutions s and s', respectively. At the beginning of the optimization, almost any new solution is accepted, even when the solution cost increase. When the algorithm proceeds and the temperature decreases, only new solutions with lower cost are likely to be accepted.

The algorithm uses a new random variable $v, 0 \leq v \leq 1$ in each iteration to determine whether a new solution is accepted or not. The acceptance probability for a solution is calculated in the function *getProbability*() that is defined as follows:

$$p = \begin{cases} 1 & \text{if } c' < c \\ \exp \frac{c-c'}{t} & \text{else} \end{cases}. \tag{A.2}$$

In any iteration, the new solution s' is accepted if $p \leq v$, cf. line 24.

Finally, the termination mechanism of the SA implementation is described. The termination is controlled by the progress of the optimization and the execution time of the algorithm. These mechanisms are used because the algorithm has no information whether a solution represents a global optimum or not.

The SA runs until no improved solution can be found during i_{fix} iterations, because then no significant improvements are expected in further iterations. The execution of the SA algorithm stops if the execution takes more than τ_{max} seconds, see line 52. The time limit can also be disabled. To be safe, the SA must run for at least i_{min} iterations, see line 49.

The following parameters that control SA behaviour have been proved to be useful during our experiments:

$$\begin{aligned} t_{\text{const}} &= 5TN \\ i_{\text{fix}} &= 2t_{\text{const}} \\ \alpha &= 0.95 \\ t_{\text{init}} &= 100 \\ i_{\text{min}} &= 1000. \end{aligned}$$

The parameter setting provides a good compromise between the algorithm runtime and the quality of the optimization result. The parameter t_{const} represents the number of permutable objects multiplied with a constant. For example, for the optimization of the node allocation, the number of permutable objects is given by the total

number of nodes in the input graphs. The objective of the parameter setting of t_{const} is to obtain 5 permutations of each object at the same temperature, on average.

The SA algorithm is given below:

Algorithm 8 Simulated Annealing Algorithm

```
1:  // initialize the initial (and best) solution
2:  s = s_b = getInitialSolution();
3:  c = c_b = getSolutionCost( s_b);
4:  // SA control variables
5:  k = 0; // count iterations
6:  k_t = 0; // count iterations with fixed temperature
7:  k_c = 0; // count iterations with fixed cost
8:  t_sum = 0;
9:  cont = true;
10: τ_start = getTime();
11: while cont do
12:    s' = PermuteSolution( s); // Pick some neighbour of solution s
13:    c' = getSolutionCost( s'); // Compute the solution cost
14:    if c' < c_b then
15:       s_b = s; // Yes, save new best solution
16:       c_b = c';
17:    end if
18:    if k < t_init then
19:       p = 1.0; // Always accept new state during initialization
20:       t_sum += c'; // Collect costs of random solutions
21:    else
22:       p = getProbability(c, c', t);
23:    end if
24:    if p ≥ getUniformRandom(0,1) then
25:       s = s'; // Yes, change state ( and save to old one)
26:       c = c';
27:       if c ≠ c' then
28:          k_c = 0; // Reset same cost counter
29:       end if
30:    else
31:       k_c++; // No, keep to old state
32:    end if
33:    k++;
34:    if k = t_init then
35:       // restart iteration with initial temperature
36:       t = t_sum/k;
```

```
37:       k_t = 0;
38:       // load best allocation found during temperature initialization
39:       s = s_b;
40:       c = c_b;
41:     end if
42:     // update temperature:
43:     k_t++;
44:     if k_t > t_const then
45:       k_t = 0;
46:       t = αt;
47:     end if
48:     // decide whether to stop or not
49:     if k > i_min and k_c > i_fix then
50:       cont = false;
51:     end if
52:     if τ_max ≥ 0 then
53:       τ = getTime();
54:       if τ − τ_start > τ_max then
55:         timelimit_exceed = true;
56:         cont = false;
57:       end if
58:     end if
59:   end while
```

About this Book

Partial dynamic reconfiguration of FPGAs has attracted high attention from both academia and industry in recent years. With this technique, the functionality of the programmable devices can be adapted at runtime to changing requirements. The approach allows designers to use FPGAs more efficiently: E. g. FPGA resources can be time-shared between different functions and the functions itself can be adapted to changing workloads at runtime. Thus partial dynamic reconfiguration enables a unique combination of software-like flexibility and hardware-like performance.

Still there exists no common understanding on how to assess the overhead introduced by partial dynamic reconfiguration. This dissertation presents a new cost model for both the runtime and the memory overhead that results from partial dynamic reconfiguration. It is shown how the model can be incorporated into all stages of the design optimization for reconfigurable hardware. In particular digital circuits can be mapped onto FPGAs such that only small fractions of the hardware must be reconfigured at runtime, which saves time, memory, and energy. The design optimization is most efficient if it is applied during high level synthesis. This book describes how the cost model has been integrated into a new high level synthesis tool. The tool allows the designer to trade-off FPGA resource use versus reconfiguration overhead. It is shown that partial reconfiguration causes only small overhead if the design is optimized with regard to reconfiguration cost. A wide range of experimental results is provided that demonstrates the benefits of the applied method.

The Author

Markus Rullmann commenced his university education in Electrical and Electronic Engineering at the Technische Universität Dresden in 1997. He received a DAAD scholarship to support his studies of mobile communications at the Newcastle University (UK) in 2000/01. Markus Rullmann graduated in 2003 at the department of Electrical and Computer Engineering back in Dresden. Since then he has been working as a research associate at the Technische Universität Dresden. The focus of his research has been FPGA based reconfigurable computing. His research was granted by the German Research Foundation (DFG) within the Priority Programme 1148 “Reconfigurable Computing Systems.” As a result of his work, he published several papers on reconfigurable computing at national and international conferences. Markus Rullmann received his PhD in 2010. Currently, he is associated with the FPGA Systems Group at Signalion GmbH.