Chapter 1
Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures

Johann Glaser, Clifford Wolf*

Abstract

Low power consumption or high execution speed is achieved by making an application specific design. However, today's systems also require flexibility in order to allow running similar or updated applications (e.g. due to changing standards). Finding a good trade-off between reconfigurability and performance is a challenge.

This work presents a design methodology to generate application-domain specific heterogeneous coarse-grain reconfigurable architectures. The specification of the reconfigurable architecture is given by a set of example applications which define the whole range of its required functionality. These applications are analyzed to extract common building blocks, which can be reused between them.

In the next step, the circuits of the application are merged to a single reconfigurable module. The major part of this work describes the according tool and its algorithm. Its main task is to optimize the interconnect by hierarchically grouping the functional units. Additional resources can be added to enable future applications. The tool generates the HDL source for a module with the instances of all blocks and the reconfigurable interconnect. The feasibility of the methodology is demonstrated by the design of reconfigurable architectures for digital filters as well as simple logic networks.

Key words: Programmable Logic Devices, Reconfigurable Architectures, Reconfigurable Logic, Design Automation, Integrated Circuit Interconnections

1.1 Introduction

In current system design a shift to employ reconfigurable logic tries to utilize their benefits for various applications. Typical wireless sensor network (WSN) nodes are

---

Institute for Computer Technology, Vienna University of Technology, Austria

* This work has been supported (in part) by the Austrian COMET K-project ECV under contract no. 815105.

supplied from batteries or utilize energy haversting. Therefore the main goal is to optimize a WSN node for ultra low power consumption. Unfortunately, the CPU as main controller consumes power even for very simple tasks. By adding a dedicated reconfigurable hardware module to offload the CPU for such simple tasks as sensor measurements or network MAC layer handling, a large reduction in the power consumption can be achieved [1]. These reconfigurable modules also enable the use of the SoC in multiple different environments, thus sharing the non-recurring engineering (NRE) costs.

Accelerators for computer vision systems should support various algorithms. Currently this is achieved by implementing all algorithms in parallel and switching between them. Since the algorithms also have common operations, a reconfigurable system can reduce the required hardware resources. In multi-standard and multi-function communication systems the same approach leads to a reduction of hardware resources [2].

Reconfigurable logic is classified by its granularity. The widely used FPGAs are fine-grained and pose a large overhead in terms of area and power. This is avoided by coarse-grained reconfigurable systems that achieve an ASIC-like performance at much lower power consumption and chip area [3, 4]. For the above mentioned applications, domain-specific reconfigurable circuits with heterogeneous, tailored blocks and a non-regular interconnection can provide further reduction in power and area [5].

In this work, a methodology for the design of heterogeneous coarse-grain reconfigurable circuits is presented. From a set of different actual applications, the set of required (possibly reconfigurable) hardware blocks and the interconnect between them is deduced. The grouping of the blocks is optimized to minimize the hardware resources of the interconnect.

This work is an extended version of [6]. First we review the design and usage of custom reconfigurable hardware. Then a detailed view on the design methodology is given. This is followed by a review and evaluation of interconnect topologies. The main part of this work is an optimization algorithm for the automatic synthesis of this interconnect. Then a short section introduces a feature-rich Verilog synthesis tool which is used for design entry of the presented methodology. This is followed by an evaluation of the algorithm results. The work ends with conclusions and future work.

1.2 Development of Reconfigurable Hardware

The generation of reconfigurable circuits is split in two phases. In the so called "*pre-silicon phase*" the reconfigurable hardware structures are designed for the application class. Secondly, in the "*post-silicon phase*" the reconfigurable silicon circuit is used to implement the actual application [7, 5].

In this work an approach is presented that provides the (semi-) automated generation of the pre-silicon circuit and can generate the configuration data for an actual application in the post-silicon phase.

1.2.1 Pre-Silicon Phase

In the pre-silicon phase, the reconfigurable circuit is designed. As first step, its specification is derived from the set of (usually similar) actual applications, which will be implemented in the reconfigurable logic. During this design space exploration, the "needs of [the] applications [...] drive the construction of the fabric" [3, p. 1]. This approach requires the a-priori knowledge of all future applications and it is generally not possible to implement a different application with the resulting fabric. To enable yet unknown applications, we propose oversizing, i.e., to include additional hardware and interconnect resources into the fabric.

The specification includes information on the employed blocks (also called functional units or cells) (e.g. adders, FSMs, ...), which can be reconfigurable themselves (e.g. an adder be reconfigured as a subtracter, reconfigurable FSM [8]). Additionally it includes the number of instances of each block as well as details on the connections among them.

1.2.2 Post-Silicon Phase

In the post-silicon phase after production the actual application has to be implemented by configuring the silicon structure designed in the pre-silicon phase. So, on the one hand, the post-silicon phase is limited by the results of the pre-silicon phase. On the other hand, the pre-silicon phase requires information on the actual implementations later used in the post-silicon phase to provide the required resources.

1.3 Design Methodology

The reconfigurable module as the result of the pre-silicon development phase will be integrated into the whole SoC. Therefore the resulting design data has to be compatible with an industry-standard ASIC design flow. This is best accomplished by delivering the reconfigurable module as a soft IP core. The required deliverables include structural and RTL (register-transfer level) hardware description (e.g., VHDL, Verilog) as well as guidelines and constraints for synthesis and place and route. Additionally, information and tools for the post-silicon phase have to be provided.

The design of such reconfigurable module soft IP cores should be assisted and automated by dedicated tools. This requires a systematic design approach which will be described in this section.

1.3.1 Specification

The start of the development requires a precise specification of the reconfigurable module. The application class of the module only coarsely defines the functionality. However, the interfaces of the reconfigurable modules to other modules of the SoC and outside the SoC can be derived. On the other hand, the functionality and inventory of the reconfigurable module itself can be specified in two different ways:

1. Define the functionality and the required flexibility in abstract terms.
2. Use a set of concrete applications, which define the whole range of the required functionality of the reconfigurable module.

This work only deals with the second form of specification. The design of a reconfigurable module is thus broken down to first developing a set of concrete applications. Then, from this set one reconfigurable circuit is derived which is able to implement each of these concrete applications.

The specification of each concrete, i.e., example application has to be easy to translate to a logic netlist to facilitate further processing by automated tools. It should employ an existing type of hardware description so that the designers do not have to learn a new one. Finally, the description should be supported by industry-standard verification tools to achieve a first-time-right SoC design. All these requirements are fulfilled by common hardware description languages like VHDL and Verilog.

1.3.2 Application Analysis

The first step of the development of a reconfigurable module is to develop and verify a set of example applications (see “App”s in Fig. 1.1). In the second step, these are processed to derive the inventory of the reconfigurable module. This consists of a pool of coarse-grain cells (e.g., FSMs, adders, ...) (which might be reconfigurable themselves) plus a reconfigurable interconnect for flexible connections among them [5, 6] (compare Fig. 1.3).

For the processing of the example applications, a special coarse-grain synthesis tool creates a netlist representation of each application. These netlists are analyzed to extract coarse-grain cells and to find commonalities between all example applications (see top “Synthesis” box in Fig. 1.1). Commonly used cells are candidates for reuse in the reconfigurable module. Analogously to the FSM plus datapath (FSM+D) concept, the control logic of each example application is mapped to an FSM using FSM extraction while all data processing is implemented using dedicated coarse-grain cells.

While FSMs are very generic building blocks, which easily can be implemented with a reconfigurable cell, e.g. as a TR-FSM [8], the development of a reconfigurable datapath is a more complex task. At the beginning of the development, a mostly manual approach is employed. The coarse-grain synthesis tool creates netlists with instances of simple coarse-grain cells (e.g., adders, shifters, ...). The designer has to investigate the schematic and manually identify related cells which together build a larger, more complex coarse-grain cell (e.g., calculating the absolute difference of two numbers, a

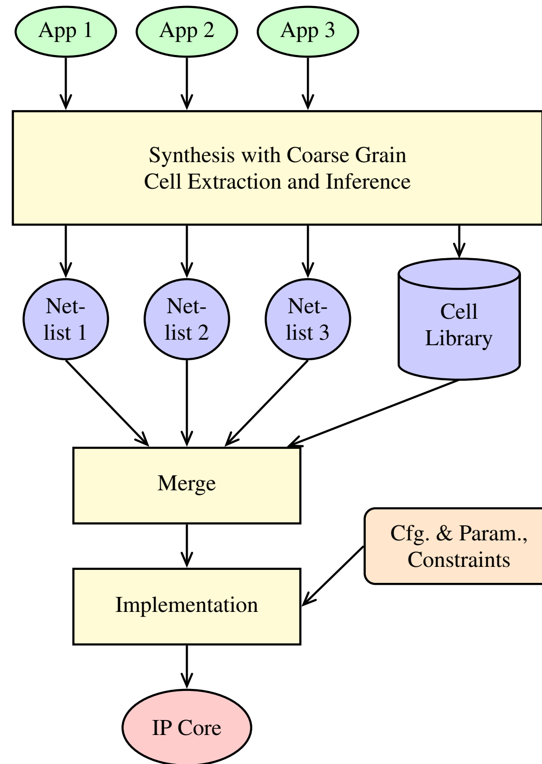

Fig. 1.1: Graphical representation of the design methodology (description is given in the text).

counter, ...). He can also generalize such groups to configurable multi-function cells (e.g. an adder and subtracter or a whole ALU). Further more, using frequent subgraph mining [2] allows to automatically identify sub-circuits which are common to multiple example applications.

For each such coarse-grain cell a small module is designed (again in a HDL). These together build the cell library (see Fig. 1.1). We call this semi-automatic procedure "coarse-grain cell extraction". It is the core point for optimization of the final reconfigurable module.

When the cell library for the reconfigurable module is finished, all example applications are analyzed again with "coarse-grain cell inference". This uses subgraph isomorphism to identify and replace all sub-circuits in each example application by coarse-grain cells from the cell library. For the further processing, each example application has to be described by a netlist of only FSMs and instances of cells from the cell library and connections among them (see "Netlist"s in Fig. 1.1).

1.3.3 Merge

After the application analysis, all separate example applications are merged to a single common reconfigurable module, which can implement each of the example applications (see Fig. 1.1). Therefore all used coarse-grain cells from the cell library including reconfigurable FSMs have to be instantiated in the appropriate number. To increase the flexibility of the final reconfigurable module, additional instances of critical cells can be added. Finally, a flexible and reconfigurable interconnect is created and optimized.

The result of this step is an RTL representation of the reconfigurable module plus meta-information for the post-silicon phase to specify how to setup the configuration data.

1.3.4 Implementation

To complete the reconfigurable module, storage and interfaces for configuration and parameterization are added (see Fig. 1.1). Together with guidelines and constraints for synthesis and place and route, the reconfigurable module is provided as soft IP core. It can be integrated into the whole SoC and processed with a standard ASIC design flow.

1.3.5 Verification

Verification is a major concern in ASIC development, therefore the described development methodology provides full coverage from the example applications to the finished IP core. Firstly, the functionality of each example application (see "App"s in Fig. 1.1) can be verified by simulation as well as formal verification due to the choice of common HDLs for specification.

Secondly, the individual netlists created by coarse-grain cell inference can be checked for equivalence to the original HDL. Since these netlists might already contain reconfigurable cells (e.g., FSMs), the configuration values have to be set to the proper values using according commands of the equivalence checking tools. The simulation of the netlist simply uses the original testbench but requires the application of configuration values before start.

Finally, it is also possible to verify whether the final reconfigurable module implements any given example application. Again, the according original testbench is employed and the required configuration data has to be applied before the start of the simulation. Additionally, equivalence checking can be employed to verify the logical equivalence of the (appropriately configured) reconfigurable module to each example application.

1.3.6 Post-Silicon Phase

In the previous sub-sections, it was assumed that the configuration data for each example application is generated together with the application analysis and merging steps. Therefore no dedicated post-silicon phase is required for these applications. However, for a new application a full post-silicon design phase is required. This is similar to designing FPGA applications, although the results of the pre-silicon phase limit the design space of these new applications.

In a first step, the new application is synthesized and coarse-grain cell inference is performed. Then, similar to the merging step above, the netlist is mapped to the inventory of the reconfigurable module and the signals are routed through the interconnect. Finally, the configuration data is generated to setup the cells and the interconnect. The result can be verified using simulation and equivalence checking.

1.3.7 Tools

The above described design methodology requires tools to assist the designer. For the synthesis of the example applications, a flexible and customizable coarse-grain synthesis tool is required. This will be described in short in Sec. 1.7. However, the major part of this paper is on the tool to merge the netlists of the example applications and generate and optimize the interconnect (Secs. 1.4–1.6).

1.4 Interconnect for Reconfigurable Modules

Most applications of coarse-grain reconfigurable logic are designed for computational tasks [4]. These use an array of homogeneous functional units connected with a highly regular interconnect (e.g. mesh structure), similar to FPGAs. In contrast, the presented approach assumes heterogeneous functional units (cell types), which also require a non-regular interconnect.

1.4.1 Common Topologies

Different interconnect topologies are evaluated in this section. The most powerful topology provides connections from every output to all inputs. The disadvantages are a large circuit overhead. On the other hand, a minimalistic interconnect with a small number of multiplexers to switch between alternative datapaths (compare [2]) does not allow to implement yet-unknown applications in the reconfigurable circuit.

Mesh structures are an alternative to the layered topology, but also assume homogeneous FUs that can be configured to perform each of its basic functionalities. The

interconnect itself requires a high number of switches which pose a high overhead in terms of silicon area and power.

In SoCs, a bus topology is used to connect the CPU with the memory and all peripherals. For reconfigurable logic circuits with all cells working in parallel, this leads to high traffic and thus congestions [4]. The utilization of every cell is reduced and the total processing time protracted, which is not acceptable in the domain of low-power circuits.

A tree based interconnect topology [9] allows to group the cells to provide short paths through lower levels of the tree for connections, which are used frequently by the different applications. On the other hand, connections to other nodes are still possible using higher hierarchical levels of the tree. This provides a large optimization potential to reduce circuit overhead but still results in a rich set of routing resources.

1.4.2 A Tree Topology

For the implementation of the reconfigurable modules the tree topology was chosen to connect the individual cells. First, a few terms have to be defined. The circuit is built out of multiple *cells*, which are instances of various *cell types* (previously called blocks, e.g. adder, FSM, look-up tables). Each has a number of input and output ports.

Analogous to the separation of the control logic and the data-path in the FSM+D concept, each port of the cell types implements a *connection type*, e.g. bit-wide, word-wide or other categories. The connection types are defined based on compatible signaling (e.g. identical bit width) as well as semantics (e.g. clock enable vs. other control signal).

All cells are connected using a reconfigurable *interconnect*. For every connection type a separate interconnect is implemented (see Fig. 1.2) which provides connections between all ports of its connection type.

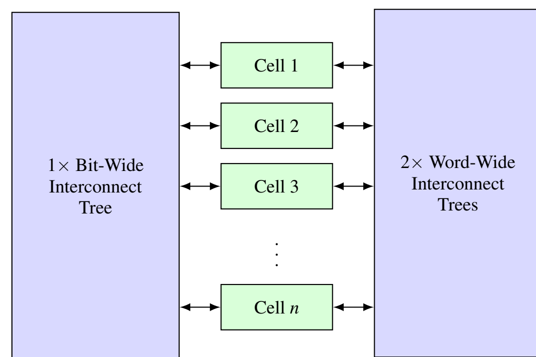

Fig. 1.2: Example interconnect with two different interconnect types (bit-wide and word-wide). The word-wide interconnect is implemented as two parallel trees.

In the post-silicon phase an actual application is implemented by connecting the cells as given by the netlist. This specifies *nodes* of certain cell types, which are *mapped* to the cells of the reconfigurable circuit. The ports of these nodes are connected with *nets* which are routed via the interconnect of the according connection type by setting the proper configuration.

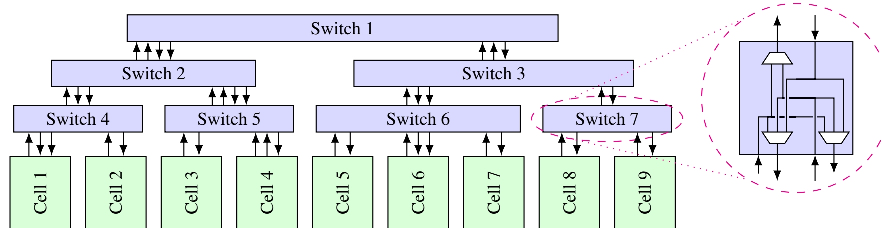

Fig. 1.3: Example interconnect with seven switches in three levels connecting nine cells of varying cell types

The interconnect is a tree (see Fig. 1.3) with the cells as leaf nodes and reconfigurable *switches* as inner nodes as well as the edges as connections (electrical nets).

The switches are unidirectional circuits that can be configured to connect any input port to any output port (see the detail in Fig. 1.3). The *degree* of a switch is the number of its *children*, (e.g. in Fig. 1.3, Switch 3 has a degree of two, Switch 6 has a degree of three). Each cell and each switch have a *parent* switch, except the top-most *root switch*. The *height* of the tree is the number of levels (e.g., Fig. 1.3 has a height of three).

The *routing length* of a net is the number of switches it passes from its source cell to its destination cell. The *total routing length* is the sum for all nets of a given netlist.

Each non-root switch in the tree has a number of connections to and from its parent switch. Only the number of these connections limits the capability of the interconnect to implement different netlists. Each switch can drive all outputs from any input, with one exception: A signal driven by one switch to another switch cannot be routed back to its originating switch.

To improve the connectivity, for each connection type multiple parallel trees with identical topology can be implemented (as also implemented by [10], compare Fig. 1.4 and the two word-wide interconnect trees on the right side in Fig. 1.2). Each cell is assigned to a (generally different) leaf node in each tree. Therefore each net can be routed in any tree. As each cell might be assigned to a different leaf node in each tree, the routing length of a net can be small in one tree but high in the other trees.

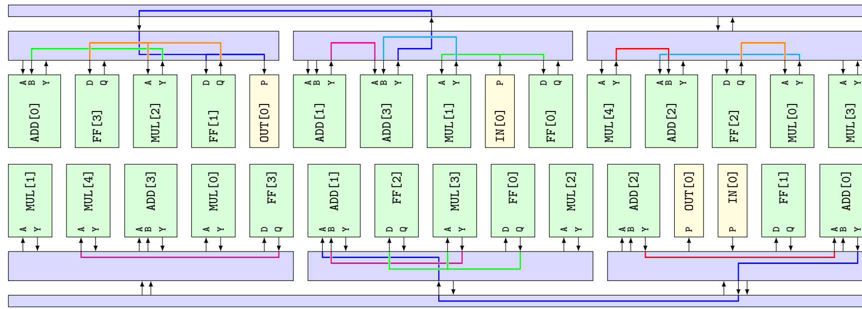

Fig. 1.4: Exemplary interconnect for the digital filters shown in Fig. 1.6 using two parallel interconnect trees (top and bottom). The routed signal paths show the post-silicon configuration for the `biquad-df1` filter. In the pre-silicon phase, the interconnect was optimized with the other three topologies. Note that the cells in the bottom half are the same as the cells in the top half, but in a different order, because they are mapped to different leaves in the second interconnect tree.

1.4.3 Analysis of the Tree Topology

In this section the tree topology as described in the previous section is evaluated. A set of six requirements is presented and the fitness of the tree topology to meet this requirements is analyzed.

1. Requirement: Allow random connections of the cells up to a certain degree.

   The set of netlists that can be implemented by a given interconnect tree is only limited by the number of connections between the switches and their parent switches and the tree layout (number of levels and degree of switches). An interconnect with only one big root switch is equivalent to a full-MUX interconnect that can implement any netlist. This might be useful for connection types with only a small number of input or output ports.

2. Requirement: Allow optimization of the interconnect for recurring pattern and similarities in the example netlists.

   The interconnect can be optimized towards the similarities in the example netlists by choosing cell to tree leaf mappings in a way that minimizes the interconnect utilization of the example netlists.

3. Requirement: Can be characterized using a relative simple and regular data structure. The existence of such a representation allows for easy manipulation and investigation of the interconnect topology.

   The whole interconnect can be described using only two simple data sets: Firstly the mapping of each cell to one leaf in each tree and secondly for each switch the number of connections to and from its parent switch. The first data set can be charaterized as a per-tree permutation and can be manipulated and optimized

easily by exchanging the assignments of two cells in one tree. The second data set is a list of integers where greater value implies more flexibility in the post-silicon phase, but also more chip resources.

4. Requirement: Prohibits over-optimization towards the example netlists that would prevent the interconnect to work with netlists that have similarities with, but are not identical to any, example netlist.

   As whole cells (instead of individual ports) are mapped to tree leaves, the optimization potential towards the individual datapaths is limited. There will always be nets that cannot be routed by only using the lowest layer of the interconnect. Thus, smart grouping of cells can be used to optimize, up to a certain degree, the interconnect to the requirements of the example netlists. On the other hand, the interconnect will not be limited to the example netlists.

5. Requirement: Allows for easy oversizing of the interconnect resources to broaden the spectrum of implementable netlists.

   Oversizing is done by increasing the number of connections between switches. Netlists that are similar but not part of the set of example netlists might have nets which result in a high routing length. With oversizing, extra routing resources help to improve these cases.

6. Requirement: Easy to implement with currently available logic synthesis tools.

   The interconnect topology provides only unidirectional links. This allows for an implementation using MUXes built from standard cells, as generated by ASIC synthesis tools. For the interconnect in most up-to-date FPGAs, unidirectional links are also reconsidered [11].
   An additional problem arises from potential combinational loops within the interconnect circuitry. This is eliminated by forbidding to route a signal back to its originating switch. On the other hand it is still possible to create loops through combinational cells connected to the interconnect. This issue must be taken care of by disabling timing arcs through these cells and applying maximum delay constraints [12].

In summary, the chosen tree topology seams to be well suited for heterogeneous coarse-grain reconfigurable architectures.

1.5 Interconnect Synthesis

A tool called InterSynth, which automatically generates the interconnect for the reconfigurable module, was implemented. It uses a set of example netlists (each representing an actual application, compare Sec. 1.2.1) with instances of cell types and connections among them. These are used to optimize the interconnect to provide cells and connectivity, suitable for implementing any of these netlists. The output is a synthesizable Verilog file that instantiates the cells and describes the reconfigurable interconnect.

```
S ← initial state

function KernLinOptimize(S, P, T)
    j ← 1
    S_0 ← S
    while P contains compatible pairs do
        S_j ← S_{j−1}
        (p_1, p_2) ← best candidate pair from P
        Swap T mapping of p_1 and p_2 in S_j
        Remove p_1 and p_2 from P
        j ← j + 1
    end while
    S ← best candidate from S_0 ... S_{j−1}
end function

if mode_align_netlists then
    repeat
        S_old ← S
        for all N = example netlist do
            P ← set of all nodes in N
            KernLinOptimize(S, P, node_to_cell)
        end for
    until S_old = S
end if

for i = 1 → max. interconnect levels do
    if mode_swap_cell_mappings then
        for all I = interconnects with min. i levels do
            P ← set of all leaves in I
            KernLinOptimize(S, P, cell_to_leaf)
        end for
    end if
    if mode_swap_node_mappings then
        for all N = example netlist do
            P ← set of all nodes in N
            KernLinOptimize(S, P, node_to_cell)
        end for
    end if
end for
```

Fig. 1.5: Intersynth Algroithm

In the pre-silicon phase, the algorithm first builds the interconnect topology with the given number of parallel trees, height of the trees and order of each level. The total number of leaves is given by the number of cells required by the example netlists. Then the cells are assigned to leaves in the interconnect trees (*cell-to-leaf-mapping*) and the required number of connections for each switch to and from its parent switch are determined so that the connections of all example netlists can be routed. In that course the algorithm also implements all example netlists. This means that for each netlist, each node is mapped to a cell (*node-to-cell-mapping*) and each net is routed via one of the interconnect trees.

1.5.1 Optimization Algorithm

During the interconnect optimization, the cell-to-leaf-mappings are permuted, so that a smaller number of connections to and from the parent switches (and therefore hardware resources) is required to still implement all example netlists. This is preformed using an iterative algorithm, a single iteration of which is shown in Fig. 1.5. It operates on the state *S*, which contains all node-to-cell-mappings for all netlists and all cell-to-leaf-mappings for all interconnect trees.

The optimization is based on the Kernighan-Lin algorithm [13], which is an heuristic procedure for solving partitioning problems by permuting the domain mappings of entities. In InterSynth it is used (in a slightly modified manner) to permute the node-to-cell- and cell-to-leaf-mappings in the state *S*. The function KERNLINOPTIMIZE in Fig. 1.5 implements the Kernighan-Lin algorithm.

For the first iteration of the algorithm a start state *S* with random mappings is used. For all further iterations the result of the previous iteration is used as a starting point. Experiments have shown that less than six iterations are usually enough for InterSynth to reach a stable state, whereas further iterations don't significantly improve the algorithms result.

The algorithm is controlled through the use of flags that enable or disable certain parts of the algorithm. Note that the KERNLINOPTIMIZE function is using different optimization goals in different parts of the algorithm. For example the term *best candidate pair* in KERNLINOPTIMIZE is using a different definition of *best* depending on the calling block. The flag `mode_align_netlists` enables a block that "aligns" the netlists so that similar subcircuits are mapped to the same set of cells. In this block the optimization goal for KERNLINOPTIMIZE is to minimize the number of unique pairs of connected cell ports over all netlists. The flag `mode_swap_cell_mappings` enables a block that permutes the cell-to-leaf-mappings for the individual interconnect trees and the flag `mode_swap_node_mappings` permutes the node-to-cell-mappings. In both blocks the optimization goal is to minimize the sum of the total routing lengths for all netlists in the top *i* levels of the interconnect trees. Therefore the first iteration of the *i*-loop only tries to reduce the utilization of the root switch and further iterations of the *i*-loop refine this first solution with respect to the other switching levels in a top-down manner.

For the *pre-silicon* procedure the algorithm is used with the flag `mode_align_netlists` enabled in the first iteration. Thus the actual algorithm is using aligned netlists as a starting point. The flag `mode_swap_cell_mappings` is set for all iterations and `mode_swap_node_mappings` is only set for the second half of iterations. Thus the algorithm first tries to find a good solution without modifying the aligned netlists and after that uses this solution as a starting point for an optimization run with all degrees of freedom. After this the number of required connections for each switch to and from its parent switch is calculated by using the maximum number of these connections used for each switch in the routing results generated by the algorithm. InterSynth also provides configuration options for oversizing.

In *post-silicon* runs the flag `mode_align_netlists` is never activated, as there is only one netlist in post-silicon runs. The flag `mode_swap_cell_mappings` is also never set during the post-silicon procedure, as the cell-to-leaf-mappings cannot be changed

once the chip has been manufactured. The flag `mode_swap_node_mappings` is set in all iterations of the post-silicon procedure. As information about the available routing resources is available during the post-silicon procedure this information is used by the post-silicon routing algorithm. Thus the post-silicon routing algorithm does not optimize for the shortest path but for least congestion.

1.5.2 Implementation Details

The actual implementation of InterSynth is using performance optimizations. For example, instead of copying S to $S_0, \ldots, S_{j-1}$, a journal of the swaps is maintained that can be rolled back to the best solution. When the number of utilized nodes of a certain type varies between the netlists, additional "dummy nodes" are added by InterSynth to level the number of used nodes across all netlists. This is necessary as InterSynth can only permute the existing cell-to-leaf-mappings. That means there must be mappings for all leafs in all trees in the initial state in order to make all possible mappings accessible to the optimization algorithm. The cell type descriptions used by InterSynth provide a flag to mark a cell input as possible feedback input. An input that does not have this flag set cannot be connected directly to an output from the same cell. For most cell types such connections would never be part of a valid netlist. The Verilog HDL code generated by InterSynth can be used as-is in the final ASIC design as InterSynth can be configured to not only include the cell instantiations and interconnect logic but also additional support code in the HDL output, such as connections of cell ports to ports of the generated module (for input and output purposes or distributing global signals such as clock and reset). It is also possible to embed configuration data for reconfigurable cells (ALUs, etc.) within the InterSynth config bitstream. Inputs and outputs of the whole reconfigurable modules are handled as special cell types and therefore are not explicitly drawn in Fig. 1.2 and 1.3. The automatically generated interconnect shown in Fig. 1.4 has only one input and one output labeled `IN[0]` and `OUT[0]`.

1.6 Evaluation of InterSynth

Two different application classes were used to evaluate InterSynth: digital filters (see Sec. 1.6.1) and logic functions (see Sec. 1.6.2). For both an identical interconnect configuration was used, which has two parallel trees of height three (although with different connection types). The switches in the bottom two layers have a degree of four and the top level (root) switch connects all switches of the second layer. In order to create more flexible interconnects, an oversizing rule for one additional connection to each switch to and from its parent switch was used.

1.6.1 Filter Networks

Two instances of the four different digital filter topologies as shown in Fig. 1.6 were concatenated in all 16 possible combinations to build netlists of higher-order filters. The cell types employed are (word-wide) adders, multipliers and flip-flops. The Verilog code for this test cases can be found in Lst. 1.1.

Test 1) From the pool of 16 netlists a random sample of n was selected and used for the pre-silicon phase to optimize the interconnect. Then the post-silicon phase was attempted with each of the 16 netlists. This test was performed 1000 times each for $n \in \{1, \ldots, 6\}$. The percentage of failed post-silicon runs per post-silicon netlist and number of pre-silicon netlists is shown in the center part of Tab. 1.1. It shows that increasing the number of example netlists n in the pre-silicon phase results in less failed attempts in the post-silicon phase. The average resource usage of the generated interconnect is expressed with two figures: the number of bits of the configuration data and the number of 2-to-1 MUXes (MUX2) required to build the interconnect. Both numbers are normalized to the total number of cell ports. The bottom part of Tab. 1.1 gives their mean for $n \in \{1, \ldots, 6\}$. For the case of $n = 1$ pre-silicon netlist, the average number of MUX2 and configuration bits is shown in the right part of the table for every pre-silicon netlist.

The test also shows that post-silicon implementation of the topology `fir4-df2.fir4-df2` fails in a significant fraction of the generated interconnects, especially for $n \leq 2$. This can be explained by the differences in the `fir4-df2` topology compared to the other three topologies in Fig. 1.6: All multipliers in `fir4-df2` are driven directly from the input (which therefore has a fanout of five) and all delay outputs are connected to adder inputs while in the other topologies delay outputs are connected to delay or multiplier inputs. It is worth mentioning that `fir4-df2.fir4-df2` does not require more routing resources than the other topologies (see right part of Tab. 1.1). It only requires a different interconnect because it is composed of different patterns. Thus an interconnect that can implement `fir4-df2.fir4-df2` as well as the other 15 topologies needs more resources than one that can only implement the 15 others.

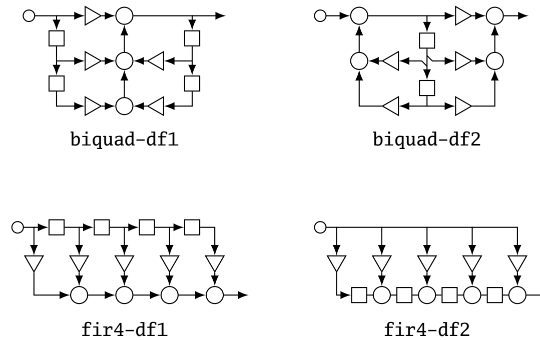

Fig. 1.6: Filter topologies used as test netlists. Circles represent adders, squares represent delays and triangles represent configurable constant factor multipliers.

Topology	Number of Pre-Silicon Netlists 1	2	3	4	5	6	Single mux2	bits
biquad-df1.biquad-df1	2.4 %	0.1 %	0.0 %	0.0 %	0.0 %	0.0 %	5.74	4.07
biquad-df1.biquad-df2	1.8 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	5.80	4.08
biquad-df1.fir4-df1	2.9 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	5.75	4.07
biquad-df1.fir4-df2	2.5 %	0.1 %	0.0 %	0.0 %	0.0 %	0.0 %	5.85	4.11
biquad-df2.biquad-df1	2.2 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	5.81	4.11
biquad-df2.biquad-df2	0.8 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	5.73	4.06
biquad-df2.fir4-df1	3.5 %	0.1 %	0.0 %	0.0 %	0.0 %	0.0 %	5.77	4.08
biquad-df2.fir4-df2	3.0 %	0.1 %	0.0 %	0.0 %	0.0 %	0.0 %	5.80	4.11
fir4-df1.biquad-df1	13.4 %	0.5 %	0.0 %	0.0 %	0.0 %	0.0 %	5.88	4.13
fir4-df1.biquad-df2	3.4 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	5.77	4.06
fir4-df1.fir4-df1	14.7 %	0.4 %	0.0 %	0.0 %	0.0 %	0.0 %	5.86	4.11
fir4-df1.fir4-df2	4.1 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	5.85	4.12
fir4-df2.biquad-df1	6.0 %	0.0 %	0.0 %	0.1 %	0.0 %	0.0 %	5.86	4.13
fir4-df2.biquad-df2	1.7 %	0.1 %	0.0 %	0.0 %	0.0 %	0.0 %	5.77	4.08
fir4-df2.fir4-df1	4.4 %	0.2 %	0.0 %	0.1 %	0.0 %	0.0 %	5.83	4.10
fir4-df2.fir4-df2	34.5 %	6.2 %	1.6 %	0.5 %	0.2 %	0.3 %	5.86	4.13
avg. mux2 / port	5.82	7.15	8.24	9.40	10.31	11.04		
avg. bits / port	4.10	4.61	5.01	5.43	5.76	6.01		

Table 1.1: Filter Network Post-Silicon Errors and Resource Usage vs. Number of Pre-Silicon Netlists

# of Trees	MUX2 / port — Degree of Interconnect Trees 2	3	4	5	6	Post-Silicon Errors / 1k — Degree of Interconnect Trees 2	3	4	5	6
1	10.74	8.39	8.42	7.21	8.34	116	82	81	43	32
2	8.51	7.86	8.27	8.31	9.14	46	34	5	6	2
3	8.48	8.57	9.75	10.39	11.80	0	0	0	0	0
4	9.57	10.21	12.23	13.22	15.22	0	0	0	0	0

Table 1.2: Number of Trees and Degree of Switches vs. Interconnect Resource Usage and Post-Silicon Errors

Test 2) The resource usage of the pre-silicon results where compared to the resource usage of an interconnect with a random, i.e, unoptimized cell-to-leaf-mappings (`mode_swap_cell_mappings` disabled in all iterations of the algorithm). The difference in the resources needed for these two cases is an indicator of the optimization potential utilized by InterSynth to optimize the interconnect for the application domain described by the example netlists. When $n = 4$ pre-silicon netlists are used and *no* additional routing resources are added, an average number of 3.0 (stddev 1.1) word-wide MUX2 per cell port are required to implement the filter example. When the InterSynth cell to leaf mapping is replaced with a random mapping and InterSynth is only used for the node-to-cell-mappings, this number increases to 7.2 (stddev 0.6). This shows that InterSynth can drastically optimize interconnects for scenarios like this one with a relatively large number of cell types compared to the number of cells.

Test 3) The number of parallel interconnect trees was varied from one to four and the degree of the switches was varied from two (binary tree) to six. For each of these interconnect configurations, two random pre-silicon netlists where used for optimization. The average resource utilization (number of MUX2) for each configuration is given in the left part of Tab. 1.2.

Each optimized interconnect was used for 1000 post-silicon netlists. The number of errors (i.e. the post-silicon netlist could not be routed within the interconnect) is shown in the right part of Tab. 1.2. For a single interconnect tree, even with a high degree of switches, a large number of post-silicon errors are present. Two parallel trees and a degree of four and above result in an acceptable number of post-silicon errors. Therefore two parallel trees with switches of degree four are a trade-off with resource utilization. More parallel trees result in a large increase of resource utilization and might also result in a wiring congestion on chip in larger scenarios.

1.6.2 Logic Networks

Random logic functions with six inputs and one output were generated and ABC [14] was used to convert these logic functions to netlists of inverters, two-input AND gates and two-input XOR gates. Of course such a problem would be better solved by rather using lookup tables than configurable interconnects and basic logic gates, but this is a simple method for generating a virtually unlimited pool of "similar" large netlists. For this test InterSynth was configured with oversizing rules to add 10 % plus 5 cells of each kind to compensate for the variation in the cell usage in the generated netlists.

Test 1) For the pre-silicon phase, four random example netlists were used to optimize the interconnect. The results from this pre-silicon phase were then tested using 1000 other random netlists (limited by the number of available cells) for the post-silicon phase. This was performed 50 times. The post-silicon run failed in only 0.05 % of these 50000 tests.

Test 2) An average number of 16.8 (stddev 0.8) MUX2 per cell port are required to implement this testcase (with four pre-silicon netlists) regardless of the question whether the cell-to-leaf-mapping was optimized or not (i.e, `mode_swap_cell_mappings` was enable or disabled). This shows that while it is possible to use InterSynth for large homogeneous networks like this test case, it doesn't have an advantage over distributing the cells regularly.

1.7 Yosys

In order to provide a convinient way for design entry, a feature-rich HDL synthesis tool with the name Yosys[2] was implemented. Yosys is a generic Verilog synthesis tool[3] that can be used in a wide variety of application domains [15].

[2] A left-recursive acronym for "Yosys Open Synthesis Suite".

[3] VHDL support is in development as of this writing.

The Verilog code in Lst. 1.1 was used to create the InterSynth netlists for the filters used in the evaluation presented in the last section. Additional input files for Yosys include a small synthesis script and an additional Verilog file that describe how Yosys should map the RTL constructs to the coarse grain cell library.

Besides simple HDL synthesis Yosys can be used for a wide range of advanced analyzes and circuit transformations. It can extract FSMs and perform various operations on extracted FSMs, such as recoding and moving additional function from logic networks into the FSM. In coarse-grain environments this can be used to move control logic into a generic FSM cell, e.g. TR-FSM [8].

Yosys also supports technology mapping by finding subcircuit isomorphism, allowing coarse-grain cells to implement richer logic function than the RTL cells used by Yosys internally. Yosys also has limited support for frequent subcircuit mining, easing the identification of possible coarse-grain cell types during the pre-silicon design phase.

1.8 Conclusion

A design methodology for application-domain specific heterogeneous coarse-grain reconfigurable logic architectures is presented. One or multiple such resulting reconfigurable modules are integrated into an SoC to off-load its CPU. This results in a large reduction of power consumption. Contrary to FPGAs, a coarse-grain and heterogeneous architecture is used, which allows further reduction in power and area.

In the pre-silicon phase, the application class for the reconfigurable module is defined and specified by several example applications. These are synthesized and analyzed to extract common logic structures as coarse-grain cells (including reconfigurable FSMs) and to build a cell library. The example application circuits are transformed to only instantiate such coarse-grain cells.

The major part of this work presents an algorithm to merge these example application netlists to a single reconfigurable module. It optimizes a tree structured interconnect and the selection of coarse-grain cells which are able to implement all example applications. Spending additional hardware resources even allows to implement yet-unknown applications with the resulting silicon.

The evaluation of the algorithm was performed using digital filter topologies. With only two example netlists and slight oversizing in the pre-silicon phase, nearly all other example netlists could be realized in the post-silicon phase. Additionally, a large optimization potential to keep the hardware resources limited was demonstrated.

We propose improvements to InterSynth in the following areas: The routing algorithm for the pre- and post-silicon phases can be improved, for example to support routing of a single net in multiple trees.

The over-all optimization procedure can also be improved: As depicted in the right part of Tab. 1.1, the hardware resources (MUX2) of the interconnect increase when more pre-silicon netlists are used, even when all example netlists can be routed. A consolidation step after the pre-silicon procedure would help reduce the hardware resources in this cases.

Listing 1.1: Verilog code for generating the filter netlists.

```
module filter(input clk, input [31:0] in, output reg [31:0] out);
    parameter type = 0;
    parameter k1 = 1, k2 = 2, k3 = 3, k4 = 4, k5 = 5;
    reg [31:0] next_tmp [3:0], tmp [3:0];
    integer i;
    always @*
        case (type)
            0: begin // biquad-df1
                next_tmp[0] <= in;  next_tmp[1] <= tmp[0];
                next_tmp[2] <= out; next_tmp[3] <= tmp[2];
                out <= k1*in + k2*tmp[0] + k3*tmp[1] +
                               k4*tmp[2] + k5*tmp[3];
            end
            1: begin // biquad-df2
                next_tmp[0] <= in + k1*tmp[0] + k2*tmp[1];
                next_tmp[1] <= tmp[0];
                out <= k3*in + k4*tmp[0] + k5*tmp[1];
            end
            2: begin // fir4-df1
                for (i = 0; i < 4; i = i+1)
                    next_tmp[i] <= i > 0 ? tmp[i-1] : in;
                out <= k1*in + k2*tmp[0] + k3*tmp[1] +
                               k4*tmp[2] + k5*tmp[3];
            end
            3: begin // fir4-df2
                next_tmp[0] <= in*k1;
                next_tmp[1] <= in*k2 + tmp[0];
                next_tmp[2] <= in*k3 + tmp[1];
                next_tmp[3] <= in*k4 + tmp[2];
                out         <= in*k5 + tmp[3];
            end
        endcase
    always @(posedge clk)
        for (i = 0; i < 4; i = i+1)
            tmp[i] <= next_tmp[i];
endmodule

module filter2(input clk, input [31:0] in, output reg [31:0] out);
    parameter type = 0;
    wire [31:0] tmp;
    filter #( .type(type % 4) ) F1 (.clk(clk), .in(in),  .out(tmp) );
    filter #( .type(type / 4) ) F2 (.clk(clk), .in(tmp), .out(out) );
endmodule

module top;
    genvar i;
    generate for (i = 0; i < 16; i = i+1) begin:list
        filter2 #( .type(i) ) F ();
    end endgenerate
endmodule
```

InterSynth is a generic tool for creating interconnects using the procedure described in this work. It is implemented in C++ and released as an Open Source project at `http://www.clifford.at/intersynth/`. The scripts used to run the experiments in Sec. 1.6 are included.

Yosys is a generic versatile tool for digital circuit synthesis. Besides its other uses, it can be used as Verilog-frontend for InterSynth as well as for circuit analysis in the pre-silicon and design phase. Is is also released as an Open Source project at `http://github.com/cliffordwolf/yosys`.

References

1. J. Glaser, J. Haase, M. Damm, and C. Grimm, "Investigating Power-Reduction for a Reconfigurable Sensor Interface," in *Proceedings of Austrochip 2009*, Graz, Austria, 7. Oct. 2009.
2. J. Ou, F. Muhammad, J. Haase, and C. Grimm, "A Technique for the Identification of Reconfigurable Resources of Flexible Communication Systems," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2011, pp. 256–263.
3. G. Mehta, J. Stander, J. Lucas, R. R. Hoare, B. Hunsaker, and A. K. Jones, "A Low-Energy Reconfigurable Fabric for the SuperCISC Architecture," *Journal of Low Power Electronics*, vol. 2, no. 2, pp. 148–164, Aug. 2006.
4. Z. ul Abdin and B. Svensson, "Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing," *Microprocessors and Microsystems*, vol. 33, no. 3, pp. 161–178, 2009.
5. J. Glaser, K. Gravogl, J. Haase, and C. Grimm, "A Reconfigurable Architecture for Ultra-Low Power Wireless Sensors," *The Mediterranean Journal of Electronics and Communications (MEDJEC)*, vol. 7, no. 3, pp. 255–266, 2011.
6. C. Wolf, J. Glaser, F. Schupfer, J. Haase, and C. Grimm, "Example-Driven Interconnect Synthesis for Heterogeneous Coarse-Grain Reconfigurable Logic," in *Forum on Specification and Design Languages (FDL)*, 18.–20. Sept. 2012, pp. 194–201.
7. J. Glaser, J. Haase, and C. Grimm, "Designing a Reconfigurable Architecture for Ultra-Low Power Wireless Sensors," in *Proceedings of the Seventh IEEE, IET International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, Z. Ghassemlooy and W. P. Ng, Eds., Northumbria University, Newcastle upon Tyne, United Kingdom, 21.-23. July 2010, pp. 343–347.
8. J. Glaser, M. Damm, J. Haase, and C. Grimm, "TR-FSM: Transition-based Reconfigurable Finite State Machine," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 3, pp. 23:1–23:14, Aug. 2011.
9. Z. Marrakchi, H. Mrabet, U. Farooq, and H. Mehrez, "FPGA interconnect topologies exploration," *International Journal of Reconfigurable Computing*, vol. 2009, pp. 1–13, 2009.
10. R. Ferreira, J. Vendramini, L. Mucida, M. Pereira, and L. Carro, "An FPGA-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Oct. 2011, pp. 195–204.
11. G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in FPGA interconnect," pp. 41–48, IEEE.
12. H. Bhatnagar, *Advanced ASIC chip synthesis using Synopsys Design Compiler, Physical Compiler, and PrimeTime*, Kluwer Academic Publishers, Boston, 2002.
13. B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 1, 1970.
14. Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification," HG 120213 `http://www.eecs.berkeley.edu/~alanmi/abc/`.

15. C. Wolf and J. Glaser, “Yosys – A Free Verilog Synthesis Suite,” in *submitted to: Proceedings of the 21st Austrian Workshop on Microelectronics Austrochip*, 10. Oct. 2013.