HLS-ENABLED
DYNAMIC STREAM PROCESSING
CONTENTS, TOOLS AND IMPLEMENTATION

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2021

Student id: 9886891

School of Computer Science
Contents

Abstract 15

Declaration 17

Copyright 18

Acknowledgements 19

1 Introduction 20

1.1 FPGA technology ................................................. 21
  1.1.1 Basics of an FPGA ........................................ 22
  1.1.2 RTL programming ....................................... 24
  1.1.3 Stream processing ..................................... 25
  1.1.4 FPGAs vs other Heterogeneous solutions .............. 26
    1.1.4.1 CPUs and GPUs vs FPGAs ......................... 27
    1.1.4.2 ASICs vs FPGAs ................................ 28
    1.1.4.3 Summary ........................................ 28
  1.1.5 Limiting factors ....................................... 29
  1.1.6 High-Level Synthesis tools ........................... 30
  1.1.7 Partial reconfiguration ............................... 31

1.2 Motivation ................................................. 35
  1.2.1 Research hypothesis ................................. 36

1.3 Publications ............................................... 37
  1.3.1 Primary author publications ......................... 37
  1.3.2 Co-author publications ............................. 38
  1.3.3 Contributions ....................................... 39

1.4 Thesis contents ........................................... 40
2 Related work
   2.1 HLS tools ........................................ 41
      2.1.1 Academic tools .............................. 42
      2.1.2 Commercial tools ............................ 42
      2.1.3 Summary ...................................... 45
   2.2 Partial reconfiguration tools ..................... 46
      2.2.1 Commercial toolflows ....................... 46
      2.2.2 Academic tools .............................. 47
      2.2.3 Summary ...................................... 51
   2.3 HLS and partial reconfiguration projects ........... 52
   2.4 Partial reconfiguration on Maxeler systems ......... 54

3 From static to dynamic HLS specifications ................. 57
   3.1 Maxeler dataflow platforms ....................... 58
   3.2 Language extension .............................. 60
      3.2.1 Existing limitations ......................... 62
      3.2.2 Extension in practice ....................... 63
   3.3 Dynamic Maxeler model .......................... 67
      3.3.1 MaxJ preprocessor ........................... 68
      3.3.2 Front-end description ....................... 69
   3.4 Design choices .................................. 70
   3.5 Summary ........................................ 71

4 RTL-level Specifications ................................ 72
   4.1 Internal Maxeler architecture ..................... 72
   4.2 Static system .................................... 74
   4.3 Kernels and Internal Communication ................ 76
   4.4 From MaxJ towards implementation ................. 81

5 Low-level aspects ....................................... 84
   5.1 Static system .................................... 84
      5.1.1 Reconfigurable region ....................... 85
      5.1.2 Static implementation ....................... 87
   5.2 Partial module implementation ................... 90
   5.3 Bitstream generation ............................. 93
      5.3.1 Bitstream extraction at netlist level ....... 93
5.3.2 Bit-level bitstream extraction ....................................... 94
5.4 Back-end overview ......................................................... 96
5.5 Customized Tools .......................................................... 98
  5.5.1 GoAhead ............................................................... 100
  5.5.2 Bounding Box Generator .............................................. 101
    5.5.2.1 Implementation .................................................. 102
    5.5.2.2 Mitigation Strategies ............................................. 106
  5.5.3 BitMan ................................................................. 108
6 Results ........................................................................ 109
  6.1 Experimental setup ....................................................... 109
  6.2 Image processing application ............................................ 110
    6.2.1 Resource Utilization .................................................. 111
    6.2.2 Compilation and Configuration Time ................................ 113
  6.3 Additional flexibility ..................................................... 115
  6.4 Scalable Filtering Modules for Database Acceleration ................. 116
    6.4.1 Filtering module implementation ................................... 117
    6.4.2 Implementation Results .............................................. 120
7 Conclusions .................................................................. 123
  7.1 Discussion and Contributions ............................................ 124
  7.2 Future work ............................................................... 126

Word Count: 28738
List of Tables

2.1 Most widely used commercial and academic HLS tools. . . . . . . . . 43
2.2 Summary of the existing tools specializing on partial reconfiguration.
Some of the contents are taken by the work in [1]. . . . . . . . . . . . 50

6.1 Resource utilization of generated PRGroups and unique kernels. The
added FIFOs introduced in Section 3 are included in those number. . . 113
6.2 Time needed for reconfigurable module generation, the static part of
our reconfigurable module and the full static design. Compilation time
does not include the RTL generation time as it generates the same
VHDL code. The FS design includes only the Maxeler compilation
time, as it is full static. The total modification and implementation
time for a dynamic project is the maximum compilation time (i.e. 74
minutes). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
6.3 Utilization and throughput of the evaluated module configurations. Note
resource columns in brackets represent only logic columns. . . . . . . 121
List of Figures

1.1 Big data market and growth from 2011 until 2026, from reports by Statista in [2]. ................................. 21
1.2 Comparison between energy efficiency, processing capabilities data growth and storage requirements in data centers. Figure depicts that the demand for processing is increasing slower than the data traffic and storage capacity. Figure originates from [3] and [4]. .................................. 22
1.3 Design of a LUT, following the truth table depicted on the left side of the figure. The green boxes correspond to the SRAM cells, where the truth table values are saved. The example is showing the output result of Y if the select bits are 0101. .................................................. 23
1.4 An FPGA architecture containing BRAMs, DSPs, LUTs, Switch matrices (SM) and IO blocks. .......................... 24
1.5 Simple pipeline example depicting 3 functions (multiplication, shift and addition) and their intermediate pipeline registers. The pipeline has an input and output streaming interface. .............................. 26
1.6 Baidu’s search engine accelerated with a Kintex-7 FPGA vs Intel Xeon CPU vs NVIDIA K10 GPU. The figure was found on [5]. .......................... 27
1.7 Diagram depicting a simple partially reconfigurable example. On the left, the figure contains an FPGA device containing three mutually exclusive applications. The right-top subfigure illustrates the same design using partial reconfiguration. The accelerators in the right subfigure are more aggressively parallelized due to the resources available for not implementing all of the processes in the same design. The colors of the accelerators are the same for the processes on the subfigures. The right-bottom subfigure is very similar to the right subfigure. In this example, the designer used a smaller device to implement the application again using partial reconfiguration. .............................. 32
1.8 Time-multiplexing between different applications. On the left figure, the applications are dynamic and the designer can change between them. The output of the device can be used as an input of the second application. A time diagram depicting the changes of accelerators in the dynamic system is depicted in the top of the figure. In the timing diagram there is latency between each load, which corresponds to the time that is required to load a partial bitfile to the FPGA.

3.1 Maxeler programming model from a user point of view. The Maxeler design flow is depicted in more details in Figure 3.3.

3.2 Instantiation of kernels in manager files. A manager can contain multiple kernels, while the user defines the exact connections among them. In the figure, we instantiated a simple pipeline in the top manager. Additionally multiple streams can be used from and to the CPU or DDR memory. In the bottom example, multiple streams are used for the inputs and the outputs that can be set by a designer and use DDR memory, CPU, or even both.

3.3 High level view of Maxeler static MaxJ to Maxfile flow from the design phase, programming in MaxJ to compilation and implementation. A designer programs the kernels and manager in the MaxJ dialect and after running simulation tests successfully, the Maxeler tool generates the VHDL code through the MaxCompiler. The generated RTL code is implemented using the ISE toolchain until the final generation of the bitfile and the corresponding MaxFile.
3.4 Block diagram describing a PRManager. The PRManager is split into two parts: the static instances and the dynamic instances. A PRManager can contain an arbitrary number of connections and an arbitrary number of static kernels, with a Reconfigurable Region module and an ICAP module. The static instances of the PRManager can also contain full static connections like static kernel 1, where its input and output does not relate with the dynamic system. Finally, a single stream needs to be instantiated to the ICAP module for reconfiguration. In addition, the PRManager can contain multiple PRGroups, as the dynamic instances. The only requirements that the PRGroups share the same reconfigurable region and provide the same physical interface. The interface needs to match all PRGroups and the Reconfigurable region instantiation.

3.5 Example block diagrams of PRGroups. All depicted PRGroups have the same interface 3 inputs and 2 outputs. A PRGroup can form a single pipeline like PRGroup_0 or a more complicated group like PRGroup_1. A kernel that is instantiated in a PRGroup can be also instantiated in another group as shown in PRGroup_0 and PRGroup_1 for kernel 2. Finally, PRGroup_N is a single kernel in this group and it is used to depict that in order to define a single kernelled partial module, it needs to be defined as a group.

3.6 Front-end implementation in a flow diagram. The input is the initial project with a prefix name of PROJECT_NAME and the preprocessor will generate the corresponding projects (project directory outputs are depicted as 3d boxes). After the project generation, the MaxCompiler will process the code and generate an RTL project version of each project generated by the MaxJ preprocessor. Last, the preprocessor will generate the final RTL top level entities of each existing group (left side of the Figure). The outputs of the front-end processing are several top level VHDL files for each group and several RTL projects generated by the MaxCompiler.

4.1 High level view of Maxeler generated system, before modifications. The example contains the Maxeler surrounding system and a MaxJ kernel.
4.2 High-level view of the static part of a dynamic design. The reconfigurable region acts as a dummy kernel to define the connections to the dynamic system and it will be replaced by a project-specific reconfigurable region with location constraints and in-region connections. The custom HDL entity contains the ICAP and its control.

4.3 Initial module chaining in Maxeler using a chaining component. The chaining component is part of each connection in the Maxeler generated RTL description. As it is a fundamental component in the generated RTL code by Maxeler, its interface is used to set a template for our RTL-level connections.

4.4 Modified kernels after the chaining module split. Kernels contain an integrated FIFO that existed in the initial chaining module while its I/O control is placed in the static system. The yellow border defines a PRGroup or a unique kernel.

4.5 PR code generator flow diagram. The tool starts by extracting the generated code from Maxeler and creates a project directory for each one of the existing Maxeler projects (including dummies). The tool automatically detects the kernels in each groups and performs the necessary changes before grouping them back into the final RTL version of the group. In the static system, the tool adds an I/O control on each input and the corresponding reconfigurable region. Each unique kernel of the groups is stored in a directory (bottom right corner) to be used as external partial modules, that act as building blocks for future use.

4.6 High level view of Maxeler static MaxJ to MaxFile flow, similarly depicted in Figure 3.3. This figure is used to depict an overview of the static implementation which will later be compared with an implementation supporting partial reconfiguration.
4.7 MaxJ to MaxFile in the here proposed dynamic approach. The figure depicts a more complex implementation that the existing static Maxeler implementation depicted in Figure 4.6, as we have more entities to process in the dynamic implementation. The two figures have the same initial implementation. Both approaches start from MaxJ and include programming the manager and the kernel files, before being processed by the front-end tool. The output of each additional process is marked by a dashed line box. The outputs of the front-end process are being processed by the MaxCompiler until the RTL generation. The generated RTL will be modified by the mid-level processing tool to create the implementation-ready entities to be sent to the back-end process. Finally, the output of the back-end process is a library of partial modules and a static bitstream, which, as in the static approach, will complete the final Maxfile of the project.

5.1 Fully placed and routed reconfigurable region, as shown by the FPGA editor tool. The region interface constitutes of 512-bits for the input and 512-bits for the output, plus 16-bits of input and 16-bits of output for implementing handshaking signals for the communication of the region and the rest of the system. The signal path is regularly routed across the span of the reconfigurable region. Additionally, pipeline stages are instantiated in the region to improve timing and constrain the routing within the region (red boxed components in figure). Those pipeline stages are double, which means that the pipelining occurs both on the front and backwards path.

5.2 Low-level implementation of the static system. The toolflow starts from the static RTL part that was generated by the mid-level processing. GoAhead generates a VHDL version of the reconfigurable region and its corresponding area constraints. Additionally, routing blockers are used to constrain routing in a predefined fashion to match the routing of the generated accelerators. The Xilinx vendor tools are used to map and place and route the final design and ultimately generate the configuration bitfile.
5.3 Empty reconfigurable region implementation (red boxed part). The interface contains 512-bits of input and 512-bits of output, plus some extra handshaking signals. The picture depicts the top-right part of the chip. We can also observe the predefined routing which is regularly structured within the reconfigurable region. The rest of the system that surrounds it is freely implemented by Xilinx vendor tools.

5.4 Low-level implementation of MaxJ generated accelerators. The toolflow starts by using the modified RTL accelerators that the mid-level produces. The Bounding Box Generator (see Section 5.5) is used to bound the modules into rectangular boxes and to calculate possible placement positions for those boxes. This occurs by the given resources for each accelerator or PRGroup. GoAhead is used to generate the constraints for a predefined bounding box. Alongside the placement constraints, GoAhead generates the routing and clock blockers of the accelerators. After placing and routing the design, its netlist can be saved in a library containing all the netlists of the accelerators of a project.

5.5 Fully placed and routed PRGroup depicting the connection macros and the interface connections. The connection macros substitute surrounding static system based on the routing within the reconfigurable region. The zoomed figure shows that the routing is constrained to strictly stay inside the module bounding box (in this case the top border). However, within the modules, there is no constraint for the routing itself. Furthermore, the interface wires on the left and the right side of the module physically placed design are implemented in such fashion that allows direct streaming to other modules or back to the static surrounding system.

5.6 Extracting partial bitstreams using the netlist level approach. Here, the main tool is GoAHead that stitches netlists that were extracted, as described in Section 5.2. GoHAhead can place those netlists in the reconfigurable region. The tool works in the background in a "copy and paste" fashion in order to apply the changes to the static netlist. Having the new netlist with the placed module, we can extract a partial bitstream by using the Xilinx differential bitstream generation.
5.7 Stitching of reconfigurable modules in a reconfigurable region in a pipelined fashion. The kernels placed in the reconfigurable region are marked with red boxes. Those kernels have been copied in a GoAhead netlist format to the previously routed static system. The kernels correspond to independent building blocks to be combined for solving more complex problems. 

5.8 An approach for generating partial bitfiles from full module bitstreams. The tool extracts the module specific information given by its module bounding box from the full bitstream of the module and overwrites the placement location in the full bitstream. The tool that handles the above process is called BitMan and it is presented in Section 5.5.

5.9 Two programming phases of a reconfigurable region. Initially the empty reconfigurable region (see Figure 5.3) is programmed with 2 kernels, before resetting the region back to its empty state with a reverting bitstream and loading another individual module. The reverting occurs between the transition from one configuration to the other (not obvious in the figures), with the resetting bitstream.

5.10 An alternative approach for generating a system that uses partial reconfiguration. Splitting and implementing multiple smaller designs, results in faster bitfile generation (for static system and partial modules).

5.11 Toolflow diagram depicting the steps that are performed after we extract the RTL code from the Maxeler Compiler to enable moving from a static implementation flow to a dynamic implementation flow. The left half of the figure illustrates the steps occurring while implementing the static part, while the right half focuses on the partial module generation. External tools are marked in blue, while Xilinx tools are marked with grey.

5.12 GoAhead GUI displaying device primitives and an example of a resource footprint. The resource footprint is a string of symbols, depending on the resource that it occupies. The resources are depicted with a yellow box on the top half of the figure, while an example resource footprint is depicted in the bottom, containing a slice-L, a slice-M, slice-L, a slice-M, slice-L and a BRAM.
5.13 Example of a reconfigurable region, spanning 3 clock regions, and some module requirements. The region is modeled with the shown alphabet, and a module is presented as a set of primitive requirements.

5.14 Algorithm applied to each module specification. Transition 1 detects the number of regions. Note that the algorithm supports different number of clock region height, however in our case, the module height is defined by the reconfigurable region (height of 2 clock regions). Transition 2 happens at every step until all resource requirements for the module are met. Transition 3 occurs once the bounding boxes starting at the current start position were found. Transition 4 allows for mitigation strategies to be employed if routing fails. This is the major contribution of this PhD thesis in [6], alongside a case study for the tool.

5.15 Bounding boxes for the defined module in the reconfigurable region, starting only at the first resource slice. Same with the example in Figure 5.13, the tool has detected 3 possible placement positions. Those positions for the module can be applied by the used as design alternatives, should they are needed.

5.16 Mitigation Strategy 2: Leaving the top and bottom of a partial module unmapped and just use the switch matrices. The right subfigure shows that in the top row of the module, only the switch matrices are used but not the primitives.

5.17 Mitigation Strategy 3: Unused primitives and used switch matrices frame around the module, to relax routing if necessary. The primitives on the sides of those kernels are completely unused and some routing is occupied to relax the internal wiring.

6.1 Fully implemented PRGroups, as shown in Xilinx’s FPGA editor tool. The illustrated PRGroups are (l.t.r.) Mean with an RGB filter, Brightness filter with a Skin detection filter, and a Min and Max filter.

6.2 Fully implemented kernels, as shown in Xilinx’s FPGA editor tool. The illustrated kernels are (l.t.r.) Brightness correction, RGB to GreyScale, Skin Color Detection, Gaussian Blur, Sobel ED filter, and Mean Filter.

6.3 Architecture of the provided module for N 32-bit data elements and K DNF clauses from [7].
6.4 Fully implemented filter modules targeting a Xilinx Virtex-6 device with capacity of 8 a, 16 b, or 32 c DNF clauses (see Table 6.3).
Abstract

HLS-ENABLED
DYNAMIC STREAM PROCESSING

CONTENTs, TOOLS AND IMPLEMENTATION
Charalampos Kritikakis
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2021

High Level Synthesis (HLS) tools enable application domain experts to implement applications and algorithms on FPGAs. The majority of present FPGA applications follows a stream processing model which is almost entirely implemented statically. Thus, most of the HLS FPGA designs do not exploit the benefits enabled by partial reconfiguration.

In this thesis, we propose a generic approach for implementing and using partial reconfiguration through an HLS design flow for Maxeler platforms, directly from the Maxeler user experience through a language extension. Our flow extracts HLS generated HDL code from the Maxeler compilation process in order to implement a static FPGA infrastructure as well as run-time reconfigurable stream processing modules. As a distinct feature, our infrastructure can accommodate multiple partial modules in a pipeline daisy-chained manner, which aligns directly to Maxeler’s dataflow programming paradigm. In addition, design choices are enabled through the proposed flow, through software and hardware implementations. Through this approach, application domain experts can design and integrate a dynamic system without focusing on the low-level details required by partial reconfiguration, while allowing flexibility by arbitrarily changing of mutually exclusive functions. All the above are done directly from the HLS aspect, with reduced implementation time of a minimum of 25% and with up to 10ms configuration overhead. The benefits of the proposed flow are demonstrated by two case studies. In the first, a dynamically reconfigurable video processing pipeline,
which delivers 6.4GB/s throughput, will be presented. This case study showcases the benefits of a dynamic implementation flow over a fully static flow. In addition, a case study for filtering database module acceleration is presented in a Maxeler platform.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses
Acknowledgements

First of all, I would like to thank my family for supporting me through my educational journey. They are people that believed in me from the first moment and making them proud through such accomplishments is the best way to repay their trust.

My childhood friends were also there for me to admire my accomplishments through the years and keeping me fully occupied during vacations. However, during this PhD I met some fantastic people that were there to inspire me and ease the transition to a whole new life and place. Could not have asked for more!

I would also like to thank the University of Manchester, that allowed me to be a part of its research team and contribute to the historically enormous contribution to science. It has been an honor to complete my studies in a world-class university. Special thanks to the APT group as well and all the people constituting it, staff PhD students and professors. I can not be more grateful to current and former PhD students that guided and advised me through the course of this PhD.

Last but not least, I can not thank enough my supervisor team and especially my supervisor Dirk Koch. In a time interval of 3 years, I have gained both technical and professional knowledge that originate mostly from experience. In addition, I can not help but admire Dirk’s effort and passion of such a well-known researcher in the field of hardware architecture. I can not skip to thank Dirk for his patience and his eagerness in supporting me. Dirk’s group is filled with brilliant people that are available to help and discuss any issues that we may have in our PhD. I will always be a part of this group and I am thankful for meeting and working with all those people.
Chapter 1

Introduction

Over the last years, there is a steep rise in the use of social networking platforms and applications, and social networks, thus the big data domain constantly grows [8]. This massive amount of data play a crucial role in the machine learning domain [9] and requires substantial processing for the data analysis. All the above have driven an increasing demand in the need for both storage and processing systems to cope with large volume of data. The steep increase of big data and its market forecast is depicted in Figure 1.1.

Moreover, due to the fact that data volume and velocity are grown with an exponential trend, the processing capabilities of super computers and data centers should increase remarkably to catch up with the current, as well as the expected, processing requirements. Thus, research is ongoing about the possible solutions, in order to significantly increase performance of computing systems. The outcome of this research is mostly bound to many-cored computing systems or heterogeneous computing.

Figure 1.2 depicts the rate of increase of both processing power and data growth (data traffic and storage capacity). Over the last decades, the community was relying on Central Processing Units (CPUs) for performance increase, following the von Neumann paradigm. The above was enabled by the radical increase in transistors in processors, as predicted by the Moore’s law [10] and by achieving higher CPU clock frequency over the years (the clock frequency rise is less significant in the last decade). In addition, multiple CPUs were used in parallel in data centers to enhance the existing performance. However, due to the growing concerns over energy efficiency of the CPUs, there are various ongoing research efforts focusing on heterogeneous computing to increase performance in applications.
Figure 1.1: Big data market and growth from 2011 until 2026, from reports by Statista in [2].

1.1 FPGA technology

Heterogeneous systems are platforms that are constituted by different types of processing hardware. Such systems are constituted by CPUs and additional heterogeneous devices, such as Graphic Processor Units (GPUs), Application-specific integrated circuits (ASICs) and Field Programmable Gate Arrays (FPGAs). Heterogeneous computing systems are considered to be a power efficient solution for performance increase of computing systems in the near future. Each one of the above devices contain unique characteristics and benefits, which will be discussed in Section 1.1.4.

This work focuses on FPGAs and how those devices can be used for acceleration and it contributes to the field of reconfigurable computing. Reconfigurable computing covers methods and techniques to implement computing systems with FPGAs. This covers architecture models, applications and algorithms, as well as programming and implementation tools. This thesis will contribute in the programmability of those devices, by allowing a more flexible programming experience to not only FPGA programmers, but also application domain experts.
CHAPTER 1. INTRODUCTION

1.1.1 Basics of an FPGA

FPGA devices were first developed by Altera back in 1983. Those devices are programmed through Boolean logic that is implemented by look-up tables (LUTs). LUTs implement boolean functions as a truth table and are mapped into memories, where the address input are the function inputs of a truth-table and the memory entries are the truth-table values. The memory itself is implemented by multiplexers that are mapped, based on k select inputs, where k is the number of input of a LUT. The logic inputs of the multiplexer are mapped into a memory (usually SRAM), where the Boolean logic of the LUT is stored, in the form of a truth-table. The SRAM in a device can store up to $2^k$ values. The LUT output can be connected to a flip-flop to store the state of a synchronous circuit. A LUT can implement every logic function that needs k bits. Xilinx uses 2 types of LUTs called SLICE(L), which is used for logic, and SLICE M, which can be also used for memory. Multiple LUTs, D-type flip-flops and full adders constitute a Configurable Logic Block (CLB for Xilinx) or Adaptive Logic module (ALM for Intel), which is considered as a fundamental building block on the reconfigurable devices. An example of a LUT is depicted in Figure 1.3.
Figure 1.3: Design of a LUT, following the truth table depicted on the left side of the figure. The green boxes correspond to the SRAM cells, where the truth table values are saved. The example is showing the output result of \( Y \) if the select bits are 0101.

In addition to LUTs, FPGAs contain Digital Signal Processing units (DSPs). DSPs are usually used to compute multiplications, divisions and floating point arithmetic operations. If implemented with LUTs, those operations will take a large amount of resources and latency and it is preferable for those operations to be calculated using DSPs. Lastly, on FPGAs there are several Block Random Access Memory (BRAM) cells that their size can vary in the range of 6 to 36 Kb of memory. Through the vendor tools, the user can customize the width and the depth of the BRAMs, using vendor specific configurations. In order to create larger memory components, multiple BRAMs can be connected to generate arbitrarily sized components. Different devices have different amount of BRAMs and it is advised to be used to implement large memory components (e.g. FIFOs, LIFOs or even line buffers in video processing application), compared to an implementation using LUTs. LUTs are preferred to implement relatively smaller memory components as distributed memories.

The aforementioned contents (i.e. LUTs, DSPs and BRAMs) constitute the design primitives of an FPGA. Apart from those primitives, FPGAs contain reconfigurable routing resources in the form of switch matrices. A switch matrix is a programmable interconnect that contains several connections to neighbor primitives and it is able to connect different inputs and outputs of CLBs and other primitives to create more complex logical implementations. Switch matrices are only used for connections between primitives. The FPGA fabric provides wires of different length, in order to implement local connections inside user defined components of the architecture and global connections between those components. Furthermore, FPGAs provide dedicated clock
routing resources to implement clock trees with low clock skew. The last resources that exist on an FPGA are the I/O cells that exist on the banks surrounding the FPGA chip. Those cells are used to map the inputs and outputs of the FPGA to the outside world or the existing peripherals implemented on-board. A simple example of an FPGA is illustrated in Figure 1.4.

Figure 1.4: An FPGA architecture containing BRAMs, DSPs, LUTs, Switch matrices (SM) and IO blocks.

1.1.2 RTL programming

Until recently, FPGA designers were programming FPGAs through Register Transition Level (RTL) languages, such as VHDL, Verilog, or System Verilog. The code in those languages is compiled through the FPGA vendor’s implementation tools to generate the final design, which is mapped in the FPGA resources. Designers can also define constrains as their desired options for the implementation process, e.g. location constrains and I/O constrains. Constrains can also be defined for the existing clocks to
1.1. FPGA TECHNOLOGY

define the routing and the span. Through the compilation and implementation process, there are several stages that are described briefly below:

- **Synthesis**: The process of translating RTL code to logic. This process extracts logic design from an RTL behavior and outputs the implementation into elementary Boolean logic functions and primitives (e.g. memories).

- **Map process**: The process of mapping/fitting the logic generated by the synthesis tool to the available resources of the target device.

- **Place and Route**: The process of placing the resources and connecting them in the target device. The routing also includes the clocks of the design. After this process, a timing analysis takes place to verify that the design meets the timing requirements and to avoid setup and hold timing errors. In addition, power analysis can be performed as well, if power needs to be considered for the specific application. Place and Route is typically the most time consuming process of a physical implementation.

- **Programming file generation**: The process of generating the final programmable file of the implemented design. This file contains all the information of every LUT, BRAM, DSP, and switch matrix for the generated design.

The programming file is used to program an FPGA device with an implemented application. An FPGA can be programmed with different programming files to implement completely different applications. Hence, FPGAs present significant benefits compared to CPUs and other heterogeneous solutions, which will be described in more detail in Section 1.1.4.

### 1.1.3 Stream processing

During the programming of an application in RTL languages, the programming model needs to focus both on the application running on the FPGA, as well as the data movement towards the application and the device itself. Usually in FPGAs, designers tend to use processing pipelines of multiple smaller functions to enhance on-chip performance. An example is shown in Figure 1.5 where 3 functions are computed for the input stream and results in an output stream. This practice does not require global control, which is usually slow in complex systems. The input of those pipelines are multiple input streams to keep the processing pipeline full and multiple output streams...
that return the result of the process. This programming model is called stream processing and it is the most preferred model for FPGAs applications [11] [12].

A stream contains an arbitrary flow of data in a specific data type, which eases the implementation of functions that exist on an FPGA design, as it allows designers to define application specific types in a bit format. Stream processing on FPGAs also allows for parallelization, because multiple streams of data can be used to send data to multiple implemented processing pipelines. Thus, the designer can implement a different pipeline for each set of input and output streams. This depends on the resources that a processing pipeline occupies in the device and the available streams on the implementation.

### 1.1.4 FPGAs vs other Heterogeneous solutions

FPGAs are used in a wide range of applications for solving compute intense problems and the total market size of those devices is expected to reach 10 billion by 2020 [13]. The use of the FPGA technology is stimulated by the fact, that FPGAs offer significant speedup and energy efficiency [14]. Additionally, when taking advantage of fine and coarse-grained parallelism, designers can customize hardware for algorithms [15] and their data movement [16] [17]. Moreover, power consumption is a growing concern regarding the power budgets and it is a major factor when it comes to large computing systems e.g. data centers. For such cases, FPGAs have demonstrated significant power consumption benefits. As an example, FPGAs were used as accelerators in the Baidu Search Engine [18] and its throughput was increased by 2x for FPGA vs GPU, while it increased performance by more than 10x over a 32-cored CPU system. The benefits in
1.1. FPGA TECHNOLOGY

Figure 1.6: Baidu’s search engine accelerated with a Kintex-7 FPGA vs Intel Xeon CPU vs NVIDIA K10 GPU. The figure was found on [5].

power and performance are depicted in Figure 1.6. Moreover, these improvements become ever more significant, if we think about the throughput–to–energy efficiency ratio, which exceeds 20x. More specifically, the power consumption of FPGA was more than 4x lower than the CPU and more than 10x lower than the GPU. Such achievements resulted in industry adopting FPGA acceleration widely and, as a consequence, all major cloud service providers have corresponding offerings (e.g. Amazon F1 instances [19], Alibaba cloud services [20] and Microsoft Azure [21]). In the next subsections, we are going to present the major advantages and disadvantages of other heterogeneous solutions and CPUs compared to FPGAs.

1.1.4.1 CPUs and GPUs vs FPGAs

FPGAs have several advantages over CPUs and GPUs. Implementations on FPGAs have shown an order of magnitude lower latencies compared GPUs [22] and CPUs [20] [18]. Moreover, FPGAs are almost completely programmable, in terms of logic and internal connections, compared to GPU/CPU architecture and data movement cannot be configured for different applications. As mentioned before, FPGAs present higher processing power to energy efficiency ratio compared to GPUs [23].

On the other hand, GPUs are more easily programmable that FPGAs. Hardware Design Languages (HDLs), such as VHDL and Verilog, are considered as more complex languages compared to high level languages, such as C, C++, etc. In addition to that, FPGAs are commonly more expensive (10K+ USD) compared to the high-end GPUs that cost around 7-8K USD.
In addition, FPGAs are fully programmable (i.e. internal functionality, interface, data movement) compared to the instruction-based architecture of a CPU, which is predefined as CPUs are instruction-based general purpose ASIC (not configurable after production). CPUs use different instructions to execute different functions. The programmability of an FPGA allows for on chip parallelization that can significantly boost performance, compared to a CPU. Similarly to GPUs, CPUs have significantly higher power consumption than FPGAs.

1.1.4.2 ASICs vs FPGAs

ASICs are considered a strong competitor in computing systems in several applications. Its major advantage compared to FPGAs is their performance because ASICs have full control over the chip (they do not pay a latency penalty for switch matrices and configuration memory), as it is produced entirely from scratch for a specific application, and the achievable clock frequency is generally higher. On top of that, ASICs can be more power efficient compared to an FPGA.

However, FPGAs are reconfigurable circuits and that means that they are programmable. Thus, by using an FPGA someone can implement any computable application by taking advantage of its reconfigurability. Moreover, FPGAs have been proven useful for applications that need to be updated frequently (e.g. radar signal processing [24] or networking [25]), as ASICs would require re-designing of the chip from the scratch to implement an update. Additionally, ASIC’s time-to-market and price is in most cases higher than an FPGA. This becomes clearer if we consider debugging and validation of the implemented function, which on an FPGA can occur even physically, using vendor specific post programming logic analyzer tools e.g. Chipscope. However, after the production of an ASIC, fixing design errors is almost impossible.

1.1.4.3 Summary

A summary of the aforementioned benefits is listed below:

- FPGAs are adaptable, as they can implement and compute any proven computable application. Note that, different algorithmic solutions for a problem may work better in other heterogeneous devices. For example there are multiple solutions for sorting multiple inputs, however different sort algorithms perform
better on CPUs, GPUs and FPGAs. However, FPGAs allow in particular to allocate resources (e.g., on-chip memory) as needed and the data movement can be orchestrated with the data processing.

- FPGAs can have orders of magnitude performance and high throughput to power ratio benefits, when the resources are used for fine and coarse grained parallelization (also shown in Figure 1.6).

- An FPGA application can have reasonably fast time-to-market, as the only necessity is the reconfigurable system and the necessary application.

1.1.5 Limiting factors

From the comparison amongst existing computing systems, the limiting factors of FPGAs can be summarized to programmability and accessibility to software programmers, implementation time and device cost. In addition to this, due to the high complexity of the existing applications, designers usually implement mutually exclusive applications on the same implementation on the FPGA chip. Thus, it would be beneficial if mutually exclusive applications could be implemented differently or as different sub-applications. The above becomes more important, if we take in consideration the price of those chips and the performance gains, should parts of the device be able to be reconfigured partially, such that only the currently needed functions are configured to the FPGA.

Thus, this PhD topic focuses on providing solutions for device programmability, implementation time and accessibility to software programmers. This will be done by introducing not currently introduced programming techniques to application experts, with respect to the FPGA technology. In order to solve the fact that FPGAs are hard to be programmed through HDLs, High-Level Synthesis (HLS) tools are introduced. These tools are used to program FPGAs, using high level languages, such as C, C++, Java instead of any other Hardware Description Language (HDL), like VHDL or Verilog. Thus, HLS tools make FPGAs more user-friendly to application-domain experts. HLS tools can even increase the productivity of FPGA experts as well, as it is much easier to use a back-end compiler that generates the surrounding system of the application while the programmer focuses only on the application itself. A more detailed analysis of those tools is given in Section 1.1.6.

Moreover, the problem of implementing mutually exclusive functions in the same
CHAPTER 1. INTRODUCTION

application can be solved by partial reconfiguration. Partial reconfiguration is the process of changing a portion of a reconfigurable design after this design is implemented. This allows for implementing a static design and loading the mutually exclusive applications arbitrarily on the FPGA, as needed at run time. This can significantly boost parallelization, as more resources of a function are dedicated to the function itself and not shared amongst different functions. Partial reconfiguration can benefit implementation time as well, as it is generally easier to implement multiple smaller sub-architectures, rather than one complex implementation that combines everything. An analysis of partial reconfiguration is given in Section 1.1.7.

1.1.6 High-Level Synthesis tools

HLS tools allow FPGA users to behaviorally design the implemented application through a high-level language (e.g. C, C++, Java, MATLAB, etc). The high-level description is then translated to RTL languages to follow the FPGA vendor’s toolflow. The translation usually occurs via an HLS compiler. The compiler generates the described applications, as well as an interconnect protocol to connect the application to the host. Each one of the HLS tools and compilers follow a tool specific interconnect protocol to generate the connections between distinct parts of the implementation.

HLS tools have shown tremendous improvements after the first implementation back in 2002 by Synopsys in [26]. The current quality of the generated RTL code has been improved dramatically and it is very close to handcrafted RTL code, or better in some cases [27]. In addition, latest HLS compilers support lexical processing, algorithm optimization control analysis, resource allocation, scheduling, and input and output processing.

An overview of the most widely used HLS tools and compilers is listed in Table 2.1 and a discussion of those tools is given in Section 2.1. The primary benefits of HLS tools are listed below:

• Designing at a high level of abstraction as a) the designer focuses mostly on the core functionality and not the implementation details, b) the I/O infrastructure is generated much easier than in RTL and c) the code is generalized and it can be used with different FPGAs (no FPGA specific primitives).

• Verifying the design at a high level of abstraction, which is in general easier to be performed in high-level descriptions.
• Benefiting from high-level specification language and their existing software development tools (e.g. Eclipse, Visual Studio, etc.) and inherit advanced language features, such as polymorphism and classes.

1.1.7 Partial reconfiguration

The idea behind partial reconfiguration is that multiple accelerators share a portion of the device at run-time. The shared physical location in the FPGA is called Reconfigurable region. The reconfigurable region is implemented in the static part of a partially reconfigurable design. A design can have multiple reconfigurable regions in the static part that allow for hosting different accelerator modules. Those modules that can be loaded in the reconfigurable region are called reconfigurable modules. A design can have multiple reconfigurable modules and, in most vendor PR flows, each reconfigurable module belongs to a specific region. To define a reconfigurable module, the user needs to define its connections to the static system. These connections are called static connections or static interface. The aggregate of the design containing the reconfigurable region and the modules are called dynamic reconfigurable system.

An example of a reconfigurable system is given in Figure 1.7. On the left, a static reconfigurable system is implemented, having 3 mutually exclusive accelerators (Process A to Process C). This design has a shared input towards the implemented processes and the output is managed through a process signal that controls the output multiplexer (MUX). The output forwards the results of only one of the processes at a time. On the other hand, on the right-top subfigure, the designer can decide to load different applications on the dynamic system to compute different results. As we can see, each one of the applications is significantly larger (the size of a process corresponds to the occupied resources), because in a dynamic design, a designer has a larger amount of free resources that can be used to exploit a larger level of parallelization on the application. Another case, that shows the benefits of partial reconfiguration is the right-bottom subfigure. A designer can select a smaller device to implement the dynamic application. Such an approach can be taken into consideration in the case that the designer aims for low power consumption. Smaller devices in general consume less power compared to larger devices and thus, it may be considered in some cases to prioritize power over performance. In another case, a designer may not have a larger device available that contains more resources and, thus, solve the resources issue using a physically smaller device.
CHAPTER 1. INTRODUCTION

Figure 1.7: Diagram depicting a simple partially reconfigurable example. On the left, the figure contains an FPGA device containing three mutually exclusive applications. The right-top subfigure illustrates the same design using partial reconfiguration. The accelerators in the right subfigure are more aggressively parallelized due to the resources available for not implementing all of the processes in the same design. The colors of the accelerators are the same for the processes on the subfigures. The right-bottom subfigure is very similar to the right subfigure. In this example, the designer used a smaller device to implement the application again using partial reconfiguration.

The latter case unveils another benefit of partial reconfiguration and that is time-multiplexing of accelerators. Time-multiplexing with partial reconfiguration can play a significant role in the design of a complex application that can not fit on a single chip. Such an application is depicted in Figure 1.8. The difference between Figure 1.8 to Figure 1.7 is the reusability of the result between the process A and B.

A designer can create a dynamic system and change between different accelerators, as shown in the figure. For that, the output is connected with the input of the reconfigurable region to output the final result. In addition, the designer should orchestrate the accelerator swap at run-time, as depicted in the timing diagram in Figure 1.8. This diagram also contains some configuration latency to illustrate the context switching overhead for changing an accelerator. This is the most significant drawback of partial
reconfiguration, but it can be hidden by the performance increase that is available by having more resources per application.

The latency exists due to the configuration of a dynamic accelerator. A dynamic accelerator is being configured through a configuration port, for example Xilinx devices offer the Internal Configuration Access Port (ICAP) or Processor Configuration Access Port (PCAP). The configuration port is an instantiated primitive on the FPGA that allows partial reconfiguration. A user that implements a dynamic system must include an input towards such a port. After the physical implementation phase, a programming file (e.g. bitfile for Xilinx) is generated. In a dynamic system, the dynamic accelerators are resulting into different partial bitfiles. A partial bitfile contains all the information of the FPGA resources and the routing of a dynamic accelerator placed in a specific location on the chip. Thus, the data size of a bitfile is smaller than a full bitfile. In case of an accelerator switch, the partial bitfile has to be streamed to the configuration port. During this phase, at least the reconfigurable region and its I/O controllers will remain idle. In some systems, a designer may decide to freeze (stopping non-configuration clock) the entire device, except the ICAP input during the reconfiguration phase. This
method is called passive, while freezing only the region and its I/O is called active partial reconfiguration.

As an alternative solution provided by partial reconfiguration in Figure 1.8, someone could propose extending to 2 devices or use a single device and switch between 2 full configuration bitstreams. The first alternative requires a significantly higher power, financial budget and, in several applications (e.g. space exploration), higher size budget. Depending on the application those budgets can be proven significant. In addition, for this example an interconnect between the devices is required. The maximum speed FPGA bandwidth can reach up to tens of Gbps, when taking advantage of the full I/O bandwidth. Thus, moving data through such a connection in an implemented algorithm can easily act as the bottleneck of the application, especially if the functional split (i.e. split of the full application between devices) requires high bandwidth.

The solution using partial reconfiguration, in this example, can be as fast as the speed of the ICAP input, that can reach up to 2 GB/s [28] [29] to transfer bitfiles, where their size reaches hundreds of MB. Thus, reconfiguration can occur in the realm of ones of milliseconds, based on the implementation speed (in MHz) and the size of the partial bitfile (assuming an inversion phase to empty the region as well). Although, partial reconfiguration has a latency overhead (in ms), the overall implementation will be faster, given a latency trade-off compared to a constant off-chip speed interconnect through the whole processing time. Some cases that the interconnect solution would be faster are either if the process executes relatively fast, or if the accelerator switch occurs very often (both cases depend on the application). Thus, the implementation depicted in Figure 1.8 could prove to be more efficient in most of the cases compared to the implementation with two FPGAs.

The second alternative would require to reprogram the device after a process finishes. The reconfiguration of the whole device will be much slower than reconfiguring just one part, but it will be much easier to implement two separate applications than use partial reconfiguration and adapt a system to its principles. However, the latency between the changes is much greater than just loading a full configuration file. This is because the results of the first application should be saved externally (e.g. DDR memory) before the configuration occurs. The data of complex applications may be tens or hundreds of GB. The connection of the FPGA to a computer usually is a PCIe connection. The maximum speed of PCIe 4.0 is 8 GB/s (32GB/s using all lines), which would require seconds to write the data to the main memory and re-write it back on the FPGA.
memory. This can prove to be the bottleneck for most applications and especially for applications that require minimizing the existing latency [30] [31] [32].

A dynamic approach on FPGAs can not be used in every case. Partial reconfiguration provides solutions mostly to large designs that can not fit a device, but that can be split into smaller accelerators to make an implementation possible. In addition, mutually exclusive applications can benefit significantly from PR, as they can be implemented separately as partial modules. Apart from space benefits, PR can offer energy efficiency from moving to a smaller device or significant performance benefits from maximizing the parallelization of accelerators by dedicating to it all the resources of a region.

1.2 Motivation

Implementing a dynamic design is considered as a low-level problem for FPGA experts. It requires strict floorplanning of an FPGA device for both implementing a reconfigurable region and offer as many resources as possible to the accelerators. The accelerators should be implemented based on the size and location of the reconfigurable region, as well as the run-time system that needs to be timed correctly to program the device with the respective accelerator. Moreover, in current dynamic systems, an accelerator is practically hardwired to a single reconfigurable region. This means that, in order to perform any change, either on an accelerator or on the static system, all the accelerators and the static system need to be compiled from the scratch. Given the time consuming implementations on FPGAs, this is not ideal if most of the parts remain the same or the change does not occur on the connecting interface between the static system and the accelerators.

For HLS tools, there is currently very basic support on some of the HLS tools for application domain experts to apply PR techniques on their design. Application domain experts are using FPGAs to write increasingly complex applications via available platforms, such as Amazon F1 [19], and tools, such as Maxeler [33] provided by vendors. Those platforms and tools do not provide sufficient support for their users to apply PR techniques.
1.2.1 Research hypothesis

This research project investigates the following research question: “How can a dynamic hardware approach be modeled automatically, implemented and executed by a runtime system on an HLS approach”?

This research includes the introduction of partial reconfiguration in a system and a tool to allow its users to be able to describe a dynamic application. Thus, this work extends the functionality of a state-of-the-art tool (i.e. Maxeler) in a way that an application domain expert can use the same high-level language to describe the functionality and the dynamic behavior of an application. It is important to include an additional subset of the existing language of the tool, but also keep the existing functionality intact. In addition, the extended functionality should be supported by the tool and compiled into the corresponding functionality.

Therefore, this work includes the automatic generation of RTL code that follows a PR approach. Such an approach should include a generalized interface protocol that it is the same for both the static system and the accelerators. Additionally, the data movement should be taken into consideration on the RTL side towards the active dynamic accelerator. An important detail, when transiting from a static system to a dynamic, is the generation of a configuration port instance (i.e. the ICAP). The final version of the static system should include an input that drives the configuration port, while disabling the reconfigurable region and its I/O. The here presented approach includes constraining and automatic placement of components in reconfigurable regions. This is not included in any other PR flow and it is of great importance as the introduced flow is going to be used by non-FPGA experts. This placement methodology also takes in consideration multi-module placement in the same reconfigurable region to utilize its occupied resources wherever possible.

Lastly, the functionality should include a back-end toolflow to implement the dynamic design. This toolflow needs to include all the stages of the physical implementation, as described in Section 1.1.1. The modules and the reconfigurable region of the static part should hide the implementation details that have to be followed for partial reconfiguration. For example, for using the resources of the modules, we should generate constraints to place them accordingly. In addition, all the subparts of the dynamic application should be generated independently, in order to generate autonomous parts such that each change on a single subpart of a dynamic design functionality (i.e. a dynamic accelerator or static system) will only require to recompile the subpart itself. To achieve all the above, we need to adapt existing or create new tools that automate
the placement and create the necessary files to constrain the implementation and create independent designs.

1.3 Publications

The paragraph below present the papers produced during this PhD. The author of this thesis produced two papers as the main author (Section 1.3.1) and two papers as a co-author (Section 1.3.2).

1.3.1 Primary author publications

End-to-end Dynamic Stream Processing on Maxeler HLS Platforms
Charalampos Kritikakis, Dirk Koch
Application-specific Systems, Architectures and Processors (ASAP) 2019
Description
In this paper, a generic approach for implementing and using partial reconfiguration through an HLS design flow for Maxeler platforms is proposed. The developed flow extracted HLS generated HDL code from the Maxeler compilation process in order to implement a static FPGA infrastructure as well as run-time reconfigurable stream processing modules.

As a distinct feature, our infrastructure can accommodate multiple partial modules in a pipeline daisy-chained manner, which aligns directly to Maxeler’s dataflow programming paradigm. This allows the decomposition of complicated problems into basic building blocks that can be easily stitched together. The basic building blocks are entirely developed using Maxeler’s MaxJ language. The benefits of the proposed flow were demonstrated by a case study of a dynamically reconfigurable video processing pipeline delivering 6.4GB/s throughput.

Enabling Dynamic System Integration on Maxeler HLS Platforms
Charalampos Kritikakis, Dirk Koch
Description
This is an invited journal of the paper submitted in ASAP 2019. This work includes an
extended version of the Maxeler flow to introduce dynamic FPGA designs. On top of the paper presented in ASAP 2019, this work includes a Maxeler language extension to allow designers to describe a dynamic system based on its contents, which are the static part (reconfigurable region and static accelerators and ICAP instance) and dynamic part (dynamic accelerators). In addition, several adaptations were included to the previously presented implementation sections of the extended paper, to create a full flow.

1.3.2 Co-author publications

Scalable Filtering Modules for Database Acceleration on FPGAs
Kristiyan Manev, Anuj Vaishnav, Charalampos Kritikakis, Dirk Koch
The University of Manchester, UK
International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART) 2019

Abstract
Database sizes are growing faster than the processing power in the post-Moore era due to the advent of big data applications, which make hardware acceleration mandatory. However, currently, database acceleration using FPGAs has mainly been static and with limited accelerator functionality, reducing the potential performance gains from customization on FPGAs. In this paper, we propose a dynamic stream processing architecture for SQL query execution on FPGAs. It achieves this by building pipelines based on scalable database accelerator primitives and partial reconfiguration. Further, we introduce novel optimization techniques to design a scalable filtering module for database restriction and Boolean evaluation. It features multiple PEs that operate in parallel and implements DNF solver to implement Boolean expression evaluation. Our evaluation shows that not only the system can support the acceleration of filtering in all TPC-H queries but provide up to 17.7GB/s throughput and scales linearly with datapath size.

Contribution
The main contribution related to this thesis includes the automatic physical implementation of static infrastructure and the database accelerator modules on a Maxeler Max3 system.

HLS Enabled Partially Reconfigurable Module Implementation
Nicolae Bogdan Grigore, Charalampos Kritikakis, Dirk Koch
Abstract
Making full use of the capabilities of the FPGA as an accelerator is difficult for non hardware experts, especially if partial reconfiguration is to be employed. One of the issues that arises is to physically implement modules into bounding boxes of minimum size for improving fragmentation cost and reconfiguration time. In this paper, we present a method which automates the modules designing step, fulfilling module resource requirements and architectural FPGA constraints. We present a case study that shows how our automatic module implementation flow can be used to generate run-time reconfigurable bitstreams that are suited for stitching together processing pipelines directly from a Maxeler MaxJ HLS specification. This takes into consideration design alternatives, fragmentation, and routing failure mitigation strategies.

Contribution
The contribution related to this thesis is to indicate the requirements based on different families (in this project, the Virtex 6), the mitigation strategies to resolve possible implementation failures on tightly constrained implementations, and provide a proof of concept via the Maxeler generated modules. In Chapter 5, it is explained how this tool benefits the here proposed toolflow.

1.3. Contributions
Our approach to enable dynamic partial reconfiguration directly through the Maxeler platforms will benefit both application domain experts and FPGA experts, as an out-of-the-box solution for the existing tools. The benefits of our approach are listed below:

- A Maxeler language extension and its corresponding automatic front-end preprocessing to enable dynamic Maxeler implementations to be described by the designer directly from the Maxeler Java.

- An automatic mid-end code generator to modify the outcome of the Maxeler Compiler and extract PR-ready RTL code for automatically implementing dynamic accelerators and the static system.

- A tool that allows for defining the bounding boxes for dynamic accelerators automatically and that searches for possible placement positions for those partial modules.
• A back-end toolflow for run-time reconfigurable systems that enables application domain experts to implement FPGA partial configuration bitstreams from HLS code that is written in Maxeler’s Java dialect.

• Enabling design choices through software, as well as portability, module replication and module stitching on Maxeler systems.

• An implementation and evaluation of a dynamic image processing system running on a Max3 platform and an implementation of scalable filtering modules for database acceleration on the Maxeler platforms.

1.4 Thesis contents

The remainder of this thesis is organized as follow:

• Chapter 2 provides a description of the related work in the field of partial reconfiguration, HLS tools and projects combining both partial reconfiguration and HLS tools.

• Chapter 3 describes the front-end adjustments to the Maxeler tool to introduce partial reconfiguration from a high-level perspective. This includes a language extension and a parser.

• Chapter 4 analyzes the mid-end processing that occurs to introduce the necessary changes to the output RTL code from the HLS tool.

• Chapter 5 provides an analysis of the low-level aspect of the generated architecture and our automated approach to hide the low-level details from the designer.

• Chapter 6 presents two test cases for the here developed partial reconfigurable HLS toolflow and describes the benefits of this approach.

• Chapter 7 summarizes the thesis contributions and discuss about the here presented toolflow.
Chapter 2

Related work

This PhD thesis focuses on the introduction of partial reconfiguration on HLS tools and more specifically, the Maxeler toolflow. This work contributes on the design time and adaptability of the reconfigurable fabric, while enabling modular accelerators that are independent from the static system. In order to proceed to the implementation phases, we analyzed the existing topics of research that focus on improving the devices, in terms of the tools and the programmability. Those areas of research, amongst other specifications, address problems by providing new HLS frameworks and partial reconfiguration techniques and tools. There are cases that various projects or vendor tools attempt to provide a more complex solution for introducing partial reconfiguration on HLS platforms and HLS High Performance Computing (HPC) systems.

This section surveys the existing HLS tools in Section 2.1, both academic and commercial. Then, the related topics on partial reconfiguration tools is presented in Section 2.2. The projects combining those specifications are presented in Section 2.3 and, finally, projects that are enabling partial reconfiguration on a Maxeler platform are analyzed in Section 2.4.

2.1 HLS tools

High-level synthesis tools are an important part of this PhD thesis, as we research ways to allow programmers to use partial reconfiguration through those tools. Thus, this section focuses on existing academic (Section 2.1.1) and commercial (Section 2.1.2) compilers. Lastly, this section presents the main reasons for our decision to use Maxeler as the HLS tool of this work. A brief comparison table is given in 2.1.
2.1.1 Academic tools

DWARV [34] is an academic HLS compiler based on the CoSy commercial compiler infrastructure [35], while it uses the Molen [36] polymorphism paradigm. Due to the development of the compiler through CoSy, DWARV inherits the tool’s extensibility of the compiler for further optimizations. The generation of VHDL from C occurs through a Control Data Flow Graph (CDFG). The second version of the DWARV compiler has been presented in [34] and it has shown up to 4.41x speed-up compared to the LegUp compiler [37].

BAMBU [38] is a modular framework for high-level synthesis, from Politecnico di Milano in 2012. This compiler covers most of the C core functionality, such as function calls, pointer arithmetic, dynamic memory addresses and a wide-range of floating point arithmetic (using the library for FloPoCo [39]). In addition, BAMBU can generate multiple Pareto-optimal implementations to trade-off latency and resource requirements. The compiler supports different devices and implements a novel memory architecture to cope with the complex constructors existing in C. This compiler was compared with the LegUp compiler in [38] and presented on average 23.5% faster accelerators. In addition, the accelerators produced by BAMBU required less resources, due to the aggressive algorithms used by the compiler.

The KiWi [40] is an HLS compiler that converts C# to Verilog for FPGA coprocessors. This compiler is inspired from parallel constructs, e.g. events, monitors and threads, which exist in C# and which are closer to hardware concepts. The KiWi compiler work also addresses problems, such as multi-FPGA designs, specific organization of the shared memory and assembly of the debug infrastructure.

HerculeS [41] [42] HLS was designed to remove the human effort by implementation tools that provide significant design assist to software-oriented developers. The benefits of the approach in [42] are addressing 1) the internal functionality of the hardware compiler, 2) the manipulation of Static Single Assignment (SSA) code, 3) the automatic IP integration and 4) backend C interface code generation. The work also presents significant computation time reduction as well as area and speed benefits.

2.1.2 Commercial tools

LegUp [37] [52] was an originally open-source HLS tool developed at the University of Toronto in 2011 and it was commercialized in 2017 through LegUpComputing. The tool covers tasks, such as allocation, scheduling and binding. LegUp is written in
2.1. HLS TOOLS

Table 2.1: Most widely used commercial and academic HLS tools.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Owner</th>
<th>Input</th>
<th>Output</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW ARV [45]</td>
<td>TU. Delft</td>
<td>C</td>
<td>VHDL</td>
<td>2012</td>
</tr>
<tr>
<td>Intel HLS [46]</td>
<td>Intel</td>
<td>C, C++</td>
<td>Verilog</td>
<td>2017</td>
</tr>
<tr>
<td>LegUp HLS [48]</td>
<td>LegUp Computing</td>
<td>C, C++</td>
<td>Verilog</td>
<td>2011</td>
</tr>
<tr>
<td>MaxCompiler [33]</td>
<td>Maxeler</td>
<td>Maxeler Java</td>
<td>VHDL</td>
<td>2010</td>
</tr>
<tr>
<td>Synphony C [49]</td>
<td>Synopsys</td>
<td>C, C++</td>
<td>VHDL, Verilog, SystemC</td>
<td>2010</td>
</tr>
<tr>
<td>KiWi C [50]</td>
<td>University of Cambridge</td>
<td>C#</td>
<td>Verilog</td>
<td>2008</td>
</tr>
<tr>
<td>HerculeS [41]</td>
<td>Ajax Compilers</td>
<td>C, NAC</td>
<td>VHDL</td>
<td>2012</td>
</tr>
<tr>
<td>BAMBU [51]</td>
<td>Polit. di Milano</td>
<td>C</td>
<td>VHDL, Verilog</td>
<td>2012</td>
</tr>
</tbody>
</table>

modular C++ to allow experimentation with newly developed HLS algorithms. The tool operates in two ways, 1) it can synthesize C code to hardware descriptions, or 2) it synthesizes C programs to hybrid systems that contain a MIPS softcore processor and the accelerators compiled from the initial C code. The programming model of LegUp also supports Pthread and OpenMP in order to parallelize multiple functions by using a pragma. In addition, the compiler also supports automatic bitwidth minimization, HLS debug, multi-cycling and loop-pipelining. LegUp targets specifically multiple Intel (former Altera) FPGA families, but was also used with Xilinx FPGAs.

The Bluespec compiler [53] [54] is a tool that uses Bluespec System Verilog (BSV) as its design language. BSV is considered as a high-level HDL language based on Verilog. BSV includes behavioral modeling of complex systems by using formal specification concepts and abstraction mechanisms to describe and organize the implementation of complex systems. The tool combines a specification in HDL with a verification language. This language is different from the other HLS languages, as it is tailored to FPGA experts and includes low-level details in its functionality. However, it has been introduced to software-developers as well but it can be used only through the Bluespec Compiler (BSC) tool [55].

Catapult-C [44] is an HLS tool that initially was designed for describing ASICs by
Calypto Design Systems and later on acquired by Mentor in 2004. Currently, the tool covers ASIC, FPGA and even SoC designs. Due to this, the tool offers flexibility to its users for device specific optimization design choices. Example of choices include settings for clock frequency, mapping and place and route parameters and streaming interfaces and RAM settings. In addition to this, the tool provides the users with multiple and flexible data types, including arbitrary length integers and floating-point arithmetic.

Synphony C [49] is a high-level synthesis tool which was originally designed for hardware DSP programming. Originally, it was developed by PICO and it was acquired by Synopsys in 2010. The tool supports memory and streaming interfaces and contains internal performance optimization methods, such as loop-unrolling and loop-pipelining. As for most Synopsys tools, Synphony C can interact with other tools in the Synopsys product family, such as Synplify, an FPGA synthesis tool and the Design Compiler for ASIC synthesis.

Intel (which acquired Altera Inc. in 2015), introduced Altera SDK [56], which includes a compiler OpenCL (i.e., C language variant) code to FPGA designs. Initially, OpenCL code was widely used for heavy parallelization on GPUs. Intel synthesizes OpenCL to a deeply pipelined FPGA implementation that connects to x86 based host processors. In 2017, Intel HLS [46] became the HLS tool used for all Intel FPGA devices, after Altera became a subsidiary of Intel in 2015. The HLS tool offers fast and easy design verification through a C++ model, as well as algorithmic development of a design and automatic integration. The tool provides an implementation report that includes suggestions for behavioral information, including loop information, pipeline status, initiation interval, component visualization and information about local memory systems. After the implementation step, a time and area analysis is provided to the user, as well as a throughput analysis for users to optimize their design.

Vivado HLS [47] is the HLS tool used for the devices of the FPGA vendor Xilinx, which is one of the major FPGA vendors, alongside Intel. Vivado HLS is a tool that originates from AutoPilot HLS [57], which became the core of the Xilinx HLS tool. The tool is aimed at synthesizing C, C++ or System C functions into IP Cores into different HDLs (VHDL, Verilog). Xilinx provides its users with built-in libraries for several application domains. Additionally, Xilinx provides a port of most of the Open Source Computer Vision (OpenCV) framework to aid designers. Furthermore, the tool can make use of the extensive library of Xilinx IP cores, which are available through the Vivado synthesis tool (e.g. to support floating-point arithmetic). As in
other HLS tools, Vivado HLS provides the user with several options to optimize the
design in terms of resource usage and performance, such as performance metrics, array
and function optimization and loop optimization. Xilinx also introduced SDAccel [58]
as its OpenCL approach. This tool allows software developers with basic experience
about reconfigurable systems, to develop applications on OpenCL, C and C++ by de-
scribing the desirable hardware architecture. Additionally, SDAccel provides the full
driven stack to integrate FPGA accelerators in x86 servers through a PCIe connection.

As mentioned in Section 1.1.6, the Maxeler system and compiler is used during
the course of this PhD. Maxeler is the only fully dataflow-based system that provides
solutions to multiple problems on FPGAs. Being a dataflow platform is of benefit to
partial reconfiguration, as the resulting RTL code natively complies with PR princi-
pies. To give some examples of these principles, the users define the interface of the
building components of the architecture and also define the architecture of those com-
ponents. This makes it possible to define the interface and the partial modules that will
constitute the dynamic architecture. However, this is debatable in a control flow sys-
tems. Maxeler exploits parallelism in a spatial domain, where multiple streams of the
dataflow are processed on the implemented functionality to compute a specific task. In
addition, Maxeler has an extensive library of dataflow accelerators from its commu-
nity. Thus, a greater community would benefit from an existing solution, which covers
partial reconfiguration through an HLS approach.

2.1.3 Summary

Furthermore, Maxeler is the only Java based HLS tool, which makes it even more
accessible to software developers. Most of the existing HLS tools are programmed
through C, which nowadays is considered a high-level language closer to hardware de-
signers than software developers. The here presented academic and commercial HLS
tools have in common that they do not support partial reconfiguration well. The HLS
tools of the major FPGA vendors provide means to change individual accelerators on
an FPGA, but still require substantial manual work to orchestrate the configuration
and the computation. Furthermore, the major FPGA vendors only support exchanging
large monolithic accelerator modules well. This PhD thesis will address both the HLS
description of a run-time reconfigurable system and the implementation of very flex-
ible reconfigurable systems in a way that it is usable by application domain experts.
Currently, there are two projects focusing on partial reconfiguration on the Maxeler
platform, as it will be shown in Section 2.4, while the functionality of the tool is going to be presented in Section 3.1.

2.2 Partial reconfiguration tools

Another topic that this PhD integrates, besides HLS tools, is partial reconfiguration. From 1995, after the first application using partial reconfiguration, there has been enormous improvement in the approaches taken for designing dynamic implementations. This does not only include commercial toolflows, but also academic tools that have introduced unique approaches to resolve specific issues of the vendor tools, such as relocatability of accelerators and re-implementation of the whole design, given any change on the dynamic accelerator or the static part of the system. This section covers the commercial toolflows in Section 2.2.1 and the academic tools in Section 2.2.2.

2.2.1 Commercial toolflows

The development of partially reconfigurable architectures on FPGA was introduced in 1995, when Xilinx was the first to patent an FPGA that could store multiple configurations at the same time [59]. A famous processing FPGA architecture to support partial reconfiguration was the XC6200 series, however the concept of partial reconfiguration became more popular with the availability of Virtex-II on the market [60]. Partial reconfiguration is supported and used on all of the major FPGA vendors (i.e. Xilinx [61] and Intel[62]).

Xilinx initially introduced PlanAhead [63] as its tool for partial reconfiguration. This tool offers an RTL based solution to the users to describe the desired distinct parts of a partially reconfigurable design, i.e. the dynamic accelerators and the static part (including the reconfigurable regions that will host reconfigurable accelerators). However, from Virtex-7 devices onwards, Xilinx introduced the Vivado toolflow. This toolflow includes PlanAhead in its toolflow, and can be used by a user either in either HLS or RTL code. The same occurs on SDAccel (through an OpenCL programming flow). The approaches of those tools, however, do not take full advantage of partial reconfiguration [64], as well as not covering automation of low-level details (e.g. constraints, clock routing), when introducing a dynamic toolflow on an HLS tool. This includes location constraints and blocking mechanisms generated for the routing. During the implementation using the Xilinx toolflow, the dynamic accelerators and the
static system design have an implementation dependency. This feature, although it provides more slack on the application when physically placed on the region, has the drawback that it requires re-implementation in the case of any change in either the static part or the dynamic accelerators. In addition to this, relocation and reusability of implemented accelerators across different systems is not possible for accelerators generated through this flow.

Altera announced the support for partial reconfiguration in Altera Quartus-II [65]. Altera also included partial reconfiguration on the newest version of Quartus Prime. Currently, Altera has been acquired by Intel. Intel introduced module variations of the generated accelerators that are named personas, while the programming versions of the device are called configuration revisions. The tool initially creates a base revision of the implementation and then, it creates multiple revisions from the existing dynamic accelerators. The placement phase partitions and locks resources for the static system and the accelerators and this step is followed by the full implementation of those sub-parts. Quartus creates an initialization revision, which is the only revision that a device can be initialized with, and then a user can load multiple other revisions, that work as partial configuration files. Currently, Intel also provides a PR simulator in their toolflow that allows users to simulate the behavior of their dynamic design. In addition, as in SDAccel, Intel allows partial reconfiguration through its FPGA SDK tool for OpenCL specific acceleration [66]. FPGA SDK provides an abstract integration of the existing accelerators, however the granularity of the implemented system is restricted to a monolithic node (e.g. individual accelerator modules can not be chained together).

The PR flow from Altera/Intel has some additional restrictions. For instance, a reconfigurable region can not be extended to the full length of the device and certain built-in memory elements (LUT-RAM) can not contain initialization values, when implemented inside a reconfigurable region. Another restriction is that during the configuration of the device, a logic "1" should be written in all memory locations of the device, otherwise a configuration error can occur. Finally, as in the Xilinx PR toolflow, static routing can be a part of the reconfigurable regions and that forces the tool for recompilation of the whole design, for every change in either the dynamic accelerators or the static system.

2.2.2 Academic tools

OpenPR [67] is an open-source PR tool that uses bitstream manipulation and a database of logic and wiring. Bitstream manipulation is enabled by an open-source tool for
FPGA development which is called Torc [68]. The user of this tool has to provide an XML file that describes specific parts of the reconfigurable design (which includes the dynamic accelerators and the static part). Then, the PlanAhead tool is used to floorplan the design, by generating constraints for the subparts of the system. After the generation of the netlists (mapped or placed and routed descriptions of a design) for each subpart, the tool follows a different path for implementing the partial bitfiles and to generate the static system. For the clock routing, OpenPR extracts the clock tree information (i.e. clock span across the design) from the static design and inserts the clock routing unchanged into the reconfigurable modules. In addition, the RTL implementation constraints generated are blocking entirely the region from static connections to be routed in those predefined locations. This allows the users of OpenPR to create independent designs and support for changes in either part (i.e. static part or accelerator) may not necessarily need re-implementation of the whole design. In addition to this, OpenPR in [67] was compared with the Xilinx PR toolflow using a Xilinx Virtex-V device, where it demonstrated a significant speed-up in the compilation time.

The OSSS+ R framework [69] allows the algorithmic description of applications in C and C++, while taking advantage of object-oriented programming for partial reconfiguration. As a distinct feature, OSSS+ foresees the concept of polymorphism to be used with partial reconfiguration. This allows it, for example, to use different reconfigurable accelerator modules depending on the datatype to be processed by the accelerator. Through this framework, a designer can define dynamic accelerators based on the accelerator usage of the whole application. The tool uses Fossy synthesis [70] which is generating the corresponding VHDL code. The generated RTL code is implemented using the vendor tools. However, a user of this tool needs to create wrappers for each module, as well as manually floorplan the reconfigurable regions.

PaRAT [71] targets to bridge the gap between partial reconfiguration and HLS tools. This is performed by extracting the architecture and the control information directly from the Vivado HLS code. Then, the tool creates a model of the dynamic system, using a partial reconfiguration modeling language (PRML) [72]. A PRML model is a directed acyclic graph, where PRML nodes and edges are the graph’s nodes and edges, respectively. The nodes represent accelerator modules and the edges denote dependencies and flow data. An execution path is a directed cycle between any pair of memory nodes. The information extracted can then be used by the Vivado toolflow to implement a dynamic system. However, PaRAT uses the Xilinx tools and thus the generated dynamic system’s accelerators are not independent.
CoPR toolflow [73] focuses on raising the abstraction level for designing dynamically reconfigurable systems. To achieve this abstraction level, CoPR automates usually manual operations required for partial reconfiguration and hides the low-level specifications from the designer. This is done through an XML description of the configurations and by introducing software code responsible for changing configurations at runtime. The tool requires a system configuration, as well as its corresponding library of modules. The tool synthesizes those modules and generates the necessary reconfigurable regions and constraints. CoPR then uses the Xilinx tools to generate the design. This toolflow does not support the latest Vivado tool suite.

R. Oomen presented in [74] a tool that automates the generation of partially reconfigurable designs for the Xilinx Vivado toolchain. The tool utilizes the already existing a bus macros to create the communications between the static system and the reconfigurable modules. For this, it incorporates the connections made of a two-input AND gate for each interface signal wire, which can be used to decouple the module from the static system. Then, the most complex partial module is implemented in all the reconfigurable regions, using multiple interfaces for each reconfigurable regions. This means that the implementation foresees the module using the most resources and the most interface wires. One of those interfaces is chosen as the main interface to be replicated amongst all the regions, using placement and routing constraints. This work, however, does not address the isolation between the static system and the reconfigurable regions.

Impress [75] was introduced in 2018 to cover the gap between tools that did not adapt, after the transition of Xilinx from ISE to Vivado toolchain. This tool is based on the TCL language and it specifically targets and uses the Vivado toolflow to generate the IPs that constitute a dynamic design. The partial bitfiles are relocatable amongst the existing reconfigurable regions. This is achieved by isolating the reconfigurable region and the module implementations and define a footprint for the implemented modules. Footprint is the number of consecutive resource columns that a module occupies (see Section 5.5.)

The project in [76] introduces, amongst other contributions, a TCL-based library that allows partial reconfiguration using the Vivado toolflow. EFCAD in [77] focuses on introducing a toolflow that supports Verilog to bitfile on-chip compilation. The on-chip compilation is performed by the ARM processors on an UltraScale+ device, while partial reconfiguration is also enabled for the compilation, demonstrating some early results.
Table 2.2: Summary of the existing tools specializing on partial reconfiguration. Some of the contents are taken by the work in [1].

<table>
<thead>
<tr>
<th>Tool</th>
<th>High-level</th>
<th>Partitioning</th>
<th>Floorplanning</th>
<th>Physical implementation</th>
<th>Relocation</th>
<th>Run-time mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx PlanAhead</td>
<td>NA</td>
<td>NA</td>
<td>Partially</td>
<td>Supported</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Xilinx Vivado</td>
<td>NA</td>
<td>NA</td>
<td>Partially</td>
<td>Supported</td>
<td>Partially</td>
<td>NA</td>
</tr>
<tr>
<td>Xilinx SDAccel</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
<td>Supported</td>
<td>NA</td>
<td>Supported</td>
</tr>
<tr>
<td>Intel Quartus</td>
<td>NA</td>
<td>NA</td>
<td>Partially</td>
<td>Supported</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Intel OpenCL</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
<td>Supported</td>
<td>NA</td>
<td>Supported</td>
</tr>
<tr>
<td>OpenPR</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>Partially</td>
<td>Supported</td>
<td>NA</td>
</tr>
<tr>
<td>OSSS+R</td>
<td>Yes</td>
<td>Supported</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>Partially</td>
</tr>
<tr>
<td>PaRAT</td>
<td>Yes</td>
<td>Partially</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>COPR</td>
<td>Partially</td>
<td>Supported</td>
<td>Partially</td>
<td>NA</td>
<td>NA</td>
<td>Partially</td>
</tr>
<tr>
<td>Impress</td>
<td>Partially</td>
<td>Partially</td>
<td>NA</td>
<td>NA</td>
<td>Supported</td>
<td>NA</td>
</tr>
<tr>
<td>GoAhead</td>
<td>NA</td>
<td>Partially</td>
<td>Supported</td>
<td>Partially</td>
<td>Supported</td>
<td>NA</td>
</tr>
</tbody>
</table>

GoAhead [78] was also introduced as a dedicated tool for partial reconfiguration for Xilinx FPGA families. GoAhead handles low level details for pre and post implementation design and can be used for generating placement and routing constraints. Its commands include functions for setting constraints, defining regions for partial reconfiguration, defining locations of partial modules or even modifying, extracting and merging design, in the form of netlists. In addition to this, a series of commands can extract routing and placement information of a design.

In addition to the tools introduced so far, there exist other PR tools and models that have either not being fully completed but they presented some different approaches on the PR topic. In [79], the Caronte methodology is used to define and allocate resources and tasks to specific reconfigurable regions. The GePaRD flow was introduced in [80] to enhance the existing Xilinx PR flow by using a high-level description as an input for generating a simulation and a physically-aware architecture. The works in [81] and [82] purpose a system containing a data plane, that implements the data processing, and the control plane, that manages the functionality directly from software. The control plane after implementation reconfigures the data plane controlled by software written by a designer. Furthermore, the work in [83] presented ASAP, a graphical user
interface tool that offers a design methodology for dynamic applications, which includes application analysis, partitioning, mapping and scheduling. The authors of [83] showed some initial results about multiple generated accelerators, however there was no other work extending or providing more results about this tool. In [84], the authors purpose an approach that implements the accelerator with its necessary communication interfaces. The interfaces were predefined at design-time for the reconfigurable system and the accelerators to allow those regions to host any of the reconfigurable modules. Finally, in [85], it has been proposed to use Unified Modeling Language (UML) for high-level description of a dynamic system.

2.2.3 Summary

This section presented the existing work in academic and industry toolflows. In order to extend the functionality of an HLS tool to support partial reconfiguration, we need to use external tools during the physical implementation phase. As referred in Section 2.2.2, there exist academic tools that offer significant implementation benefits as compared to the current commercial toolflows (i.e. relocatability of partially reconfigurable modules and generalization of the dynamic accelerators and the static part). This work will present a toolflow starting directly from an HLS model, handling and hiding all hdl modifications and floorplanning required in a dynamic project. The flow will also allow flexibility on the accelerators side for relocation and replication after the project is complete, without the requirement of re-implementing the design. Also, low-level mitigation strategies are included to solve possible physical implementation failures of components.

As our vendor for this PhD thesis, we use a Xilinx device, as most of the academic tools are focusing on the Xilinx toolflow. Hence, it was decided to use three academic tools in this project, alongside the Xilinx tools. Those tools are GoAhead [78], Bounding Box Generator [6] and BitMan [86]. GoAhead is the core tool of our toolflow, that is used for the generation of PR specific constraints and physical implementation of reconfigurable modules. The main reasons we decided to use this tool amongst the existing is that it:

- Provides a built-in GUI interface that allows for FPGA experts and software programmers to have a visual representation of the chip and the resources used by the static system or a reconfigurable module.
Can be automated though its GoAhead specific scripting files (goa scripts) to generate constraints and modules.

Able to constrain the clock and all other routing in a module or a static system and set an interface for the implementation.

Can be controlled through external parameters to generate multiple accelerator designs and reconfigurable regions automatically.

Furthermore, Bounding Box Generator was introduced during this PhD. This tool allows constraining accelerators into strict bounding boxes, while also detects possible placement positions for the generated accelerators at design time. In this project, Bounding Box Generator is used to extend the capabilities of GoAhead by generating automatically constraints for any given accelerator in a reconfigurable region. Lastly, BitMan was used to enable both design and run-time relocation of modules and module replication, as needed. More details about the functionality of the tools and how they are used in the design approach of this thesis are provided in Section 5.5.

2.3 HLS and partial reconfiguration projects

The previous section introduced existing tools that allow for implementing partially reconfigurable systems. This section focuses on project HLS frameworks or extensions that are designed specifically to support partial reconfiguration.

In [87], the framework PCIeHLS was presented to provide the necessary infrastructure for partial reconfiguration using OpenCL. The infrastructure consists of PCIe, DDR memory, an ICAP for partial reconfiguration and clock managing blocks. This framework allows time-multiplexed use of the reconfigurable regions, as well as region combination, to provide more parallelization of the executed application and module relocation. Furthermore, the system supports module relocation to simplify module management. Experiments showed a physical implementation and reduced implementation time.

The work in [88] introduced the concept of resource elastic visualization on FPGAs. This work allows resizing of applications in multiple regions to enhance performance using partial reconfiguration. The resizing occurs, on a possible request by another application and resource are re-allocated to serve all the requested applications simultaneously. The accelerators are programmed through OpenCL and the
time-multiplexed applications are called tasks. Each task gets allocated a time slot and resources, based on the current utilization. The results in this work demonstrated 25-36% performance increase, better utilization of the reconfigurable region and lower wait time for the accelerators.

The EXTRA platform [89] allows programmers to use C or C++ languages to implement reconfigurable systems a hardware-software co-design framework, with parallel memory access and transparent virtualization of the application. The virtualization is achieved via an internal tool, called RACOS. RACOS [90] is an OS-like implementation that allows scheduling of reconfigurable hardware and accelerators transparently from the user. In addition, the CAOS [91] framework was used for HPC systems to provide the interfaces and architectural templates for the reconfigurable systems. Additionally, PolyMem [92] was developed as a high-bandwidth memory implementation by loading critical data for the application in the distributed memory blocks. Lastly, the DAER [93] framework was used for efficient reconfigurable application mapping on the FPGA fabric. The platform was evaluated using an image application. In its second version, EXTRA [94] allows for creating fundamental building blocks for dynamically reconfigurable HPC system implementations, while supporting the newest devices and additional tools. The second version did not introduce any additional implementation or result on the EXTRA framework.

A Run-Time Reconfigurable Manager (RTSM) tool was introduced in FASTER [95]. This project covers low-level tasks by managing placement, scheduling and partial reconfiguration. The tool also focuses on low-level tasks, such as fragmentation configuration caching, prefetching and configuration bitstream compression and decompression, while managing the implementations’s power footprint. This work was experimented using a Maxeler Max3 platform (not through using the Maxeler software) and resulted in resource utilization benefits with some performance benefits.

The RTSM tool in the FASTER project was extended by [96] and [97]. Those projects enable designers to describe, implement and verify dynamically reconfigurable applications on FPGAs. It supports coarse-grained parallelism, by enabling the swapping of reconfigurable modules which is controlled at design time, and also fine-grain reconfiguration, that allows for changing small parts of the FPGA at run-time. A project is initially described in XML and contains a description of the desired partially reconfigurable architecture. Then, the XML description is analyzed and implemented through a toolchain of the FPGA vendor, which provides different implementation options for the existing static and dynamic accelerators. This tool-chain
includes micro-reconfiguration, which was firstly implemented within the FASTER project. The previously presented RTSM [95] was used in this toolflow, as well as verification methods for streaming designs, run-time reconfigurable designs and HW-SW co-design systems. This work also presented three different case studies, built from the FASTER framework. This work uses vendor tools for implementing partially reconfigurable modules and thus, module replication and relocation is not supported.

The work in [98] presents another RTSM implementation for scheduling dynamic application on dynamically reconfigurable systems, which is integrated in the FASTER project. This work offers portability of the RTSM for scheduling SW and HW implementation, support for loops, forks, joins and branches in complex graphs and support for multiple types of scheduling. The project was evaluated via three case studies to evaluate its scheduling performance.

Finally, the ECOSCALE project [99] presents an OpenCL based run-time system for exascale systems. This project extends Xilinx OpenCL framework by introducing a scalable approach that extends into multiple compute nodes, i.e. workers. The workers are connected as a tree-implementation. The ECOSCALE project uses a Partitioned Global Address Space (PGAS) memory model which allows an accelerator module to access any memory across the entire machine.

2.4 Partial reconfiguration on Maxeler systems

As mentioned previously in this Chapter, Maxeler has a wide range of applications [100]. Some of those applications can benefit by the introduction of partial reconfiguration on the Maxeler tools, in order to allow application domain experts to implement such applications. For this, there are a few projects that focused on enabling such features on those platforms, by introducing the benefits from applications, introducing reconfigurable interconnects or extending the existing Maxeler flow itself.

The first work applying partial reconfiguration on a Maxeler platform, was the MSc project of J.J. Jensen [101]. In that work, the author presents a library of database accelerators that can be stitched together at run-time. The goal of this project was to show that some queries can be accelerated in hardware by stitching together basic building blocks at run-time. The building blocks would be chosen depending on the query that is currently executed. The accelerators used in that project were generated through the CustomHDL interface from Maxeler, while the surrounding static system was generated by the MaxCompiler.
The case study, presented in [101], contained accelerators to compute four operations. The operators compute an AND, OR, ">" and "=" on string values, and reported a throughput of 6.25 GB/s (due to the Maxeler clock frequency at 100 MHz). All the generated accelerators could be run at 300 MHz. The project was not fully implemented on an FPGA, however the author proves that accelerating queries in FPGAs dynamically can offer significant speedup and that putting together smaller building blocks can create a datapath with significant results.

In [102], the authors present a different approach for reconfigurability in the Maxeler platforms. In this framework, a user can program multiple kernels in Maxeler Java, which is compiled to a programmable file for the FPGA. In such an application, there must be multiple accelerators to be connected by a novel Reconfigurable Interconnection Element. This interconnect is connected with all the existing accelerators in order to be programmed to create different pipelines of already implemented kernels during run-time. This work is based on the Xilinx partial reconfiguration flow that does not allow direct communication between reconfigurable modules. With the help of the interconnect element, programmable connections can be set between a chain of modules. However, such an approach require the implementation of even the idle accelerators (not partially used but not reconfigured), which makes the approach wasteful with respect to the amount of resources.

R. Cattaneo [103] also implemented a system for dynamic reconfiguration on Maxeler. This work considers a video streaming application like the one considered in its case study. The application includes four filters, which are Noise cancellation, Greyscale, Edge detection, and Threshold filter. The achievable throughput is 64.8 MB/s on a single stream and [103] also by using multiple regions and multiple stream operators with 4 input streams and 8 regions, where the aggregated throughput was 176.4 MB/s. As in [103], a PR toolflow was presented for the Maxeler platforms. However, the compilation and implementation process was not introduced. This is essential, as partial reconfiguration remains still a low-level description challenge even for FPGA experts. Thus, a brief description on the approach taken for implementing the static system and the accelerators is considered necessary. However, for this PhD thesis, we analyzed and we will present the necessities of such an approach that allows the implementation of a run-time reconfigurable system, incorporating all low-level aspects, in order to propose a way that those aspects can be hid and used by application domain experts.

However, the most important difference to this PhD thesis is that the authors in
[103] have not considered placing multiple modules in a single region. The work in [103] suggests chaining modules at compile time, but that would be prohibitively slow for run-time adaptive systems such as database acceleration where the exact chain of the modules is only known at run-time. In addition, [103] does not consider automatic placement for accelerators, which is essential when introducing a low-level implementation for non-FPGA experts. Lastly, the authors mention that they are using the Xilinx PR flow for their implementation. Thus, each kernel is therefore hardwired in a region and cannot be replicated or relocated. This makes sharing for more throughput much more difficult and every modification would require full system re-compilation and implementation.
Chapter 3

From static to dynamic HLS specifications

Application-domain experts are required to provide an HLS description, in order to use an HLS toolflow for implementing a dynamically reconfigurable system. To introduce this description in the Maxeler programming model, this project needs to include a language extension. The first step in making an HLS language extension, which supports partial reconfiguration is to set a well-defined HLS approach, while taking in consideration the already existing HLS language. For example, an application expert that has experience with an HLS framework and a high-level language should feel confident using the extension which supports partial reconfiguration, without the need of readapting. The main goal of this PhD project is to use a language extension to hide the complexity which arises from partial reconfiguration. Such complex low-level functions are clock and interface routing for and within the region, location constraints or detecting relocation positions that an application expert would avoid interacting with when using a dynamic FPGA heterogeneous system. This language extension has to cover a number of design choices, as well as to keep the initial programming model intact.

This section focuses on describing the initial programming method used in Maxeler platforms and analyzes the philosophy behind the here proposed language extensions. By introducing an extended version of the language, a pre-parser is necessary to process the extensions for designing a partially reconfigurable system. Additionally, we discuss the design choices enabled by the middle and back-end sides of this toolflow for dynamically reconfigurable system generation, as being analyzed in more detail in Chapter 4 and Chapter 5.
3.1 Maxeler dataflow platforms

Maxeler Workstations are heterogeneous computing platforms that are using both a CPU and FPGAs to perform computations. Maxeler’s FPGA devices are named Dataflow Engines (DFEs) in the Maxeler programming model. Those devices are programmed by a Java dialect, called MaxJ. The software can connect from and to the implementation running on the FPGA device through PCIe or DDR memory channels, or even both, and the connections are handled by the MaxCompiler itself through I/O specific drivers.

![Maxeler Programming Model](image)

Figure 3.1: Maxeler programming model from a user point of view. The Maxeler design flow is depicted in more details in Figure 3.3.

In the Maxeler programming model [104], which is shown in Figure 3.1, a designer has to focus on three basic coding parts. First, the CPU interface code which is written in C code and manages the data I/O of the system. Second, the main kernels/accelerators, which contain the actual acceleration functionality to be implemented on the FPGA and third the manager, that focuses on the connection between memory and/or PCIe with the reconfigurable system, as well as the internal connections between the instantiated kernels. The manager and the kernels are written in MaxJ. Figure 3.2 depicts two managers, containing three and four kernels as block diagrams. In the Maxeler programing model, the user can define an arbitrary number of kernels that can be connected internally and externally amongst them. In the first example, the user defines a simple pipeline of kernels to perform a computation, while in the second
example the user has multiple inputs and outputs, from and to the software, and the kernels have different interfaces among them.

The manager orchestrates the connections between the software and the hardware and within the hardware application itself for each Maxeler project. Each Maxeler project contains exactly one active manager file which contains the full implementation of the desired application. The generated files before and after the RTL generated version of the manager file are handled by the MaxCompiler. The manager file also defines and generates the software interface commands. Each manager can have multiple software interface commands, which are enabled for more complicated problems, such as enabling specific parts of an implementation, and enabling or disabling the inputs and outputs of a system. Moreover, a user can define these interfaces in the manager for more control over the whole application. The available Maxeler software interfaces to the DFE are called Basic static, Advanced static and Advanced dynamic. Basic Static allows a single function call to run the DFE using simple actions defined for the particular Maxeler programing file (Maxfile). Advanced Static allows control of loading the DFEs, setting multiple complex actions, and optimization of CPU and DFE collaboration. Advanced Dynamic allows for full scope of dataflow optimizations, fine-grained control of allocation and de-allocation of all dataflow resources. The here presented extension should allow for each interface type to function transparently.

After debugging and simulating the MaxJ code, Maxeler compiles the kernels and the manager of the system and the corresponding VHDL code is being generated. After the VHDL code generation, the tool follows the FPGA vendor toolchain for synthesis, mapping, routing and finally generating the bitfile. In this thesis, the ISE tools suite from the FPGA vendor Xilinx is being used. At this point, the Maxeler flow performs an additional step, which is to create a monolithic binary file that contains the full static configuration of the system. The file is called Maxfile by Maxeler and it is the final product of the Maxeler compilation toolflow. The Maxfile contains the software interface commands that were instantiated in the manager. Those commands handle the connections between the CPU and the FPGA. In order to run the system, Maxeler executes the functions provided in the generated Maxfile via the C interface code. Through this interface code, a user can define the input of each stream and manage the corresponding output of the FPGAs. Figure 3.3 shows an overview of the whole design and the implementation flow.

Maxeler offers a Custom HDL interface in order to allow the integration of optimized RTL code to be used within Maxeler’s framework. The only difference to the
CHAPTER 3. FROM STATIC TO DYNAMIC HLS SPECIFICATIONS

Figure 3.2: Instantiation of kernels in manager files. A manager can contain multiple kernels, while the user defines the exact connections among them. In the figure, we instantiated a simple pipeline in the top manager. Additionally, multiple streams can be used from and to the CPU or DDR memory. In the bottom example, multiple streams are used for the inputs and the outputs that can be set by a designer and use DDR memory, CPU, or even both.

The original HLS enabled flow is that a designer provides links to directories with VHDL modules or netlists, while the kernel file acts as a wrapper for the connections to the Manager. The final result is still a Maxfile, although the MaxCompiler treats the HDL hierarchy as a top module and only focuses on the dataflow before and after the external HDL code.

3.2 Language extension

The main difference of a static and a dynamically operated FPGA system is that a static system contains one or more static configurations of the device, while the dynamic counterpart contains a static system with multiple components that can be selectively loaded in the device to perform different computations (including combinations of those components for more permutations). In a similar, but static approach of multiple full configurations, the number of configurations should match all foreseen permutations at design time (i.e. possibly tens of configurations). The dynamic system
3.2. LANGUAGE EXTENSION

is constituted by a static part and its dynamic accelerators. Having all accelerators in a list of loadable components, there can be numerous combinations for multiple accelerators. However, neither traditional HLS description languages nor Maxeler’s MaxJ have built-in constructs that allow expressing the reconfigurability of a dynamic system.

A static implementation on an FPGA needs to be resetted and reloaded in order to switch between functions. However, should we consider mutually exclusive applications implemented in an FPGA, the design should be able to load different accelerators as they are needed, without the need of resetting the device. In such cases, partial reconfiguration would be a viable solution compared to a static design with multiple bitstreams or a design containing all mutually exclusive functions. However, this switching should occur relatively seldom, in order to minimize the configuration overhead.

In order to offer the benefits of partial reconfiguration, we need to support building and operating run-time reconfigurable applications, Hence, the system needs:

- A methodology to specify partially reconfigurable modules that use a predefined
interface to communicate between the static part and predefined connections be-
tween different modules for internal module communication. These physical
hardware interfaces have to match communication primitives used in software
(Section 3.2.2).

- High-level language constructs to orchestrate the configuration at run-time (Sec-
tion 3.3).

- Allowing an arbitrary number of modules and connections to be instantiated as
reconfigurable modules and providing design freedom (multiple connections and
static accelerators) on the static part (Section 3.4).

- Drivers in the run-time system that carry out the actual configuration of the
FPGA with kernels as well as managing input and output data (Section 3.2.2).

- A surrounding system on the FPGA that allows placing of different modules at
run-time and providing interfaces that incorporate partially reconfigurable accel-
erators/kernels (Section 3.3.2).

This section explains the adaptions made on the Maxeler specifications, before
MaxCompiler compiles the MaxJ code to RTL. It also analyzes the current limitations
and how they have been mitigated, in order to ease the design of a dynamic process
from a user’s point of view.

### 3.2.1 Existing limitations

The first goal while designing a language extension is to allow all previously existing
functions to be available in the extended version of the tool without any changes. With
this, we allow using all types of existing connections available from MaxJ from the
software to the FPGAs and vice versa. Furthermore, an extension should allow simu-
lation and the tool’s debug/error messages have to function with the extended dynamic
applications, as used by Maxeler. With this, we maintain the high-level user experience
as know from Maxeler systems.

For implementing a module library of reconfigurable accelerator modules, the Max-
eler compiler cannot extract RTL generated code of kernels that is not going to be used
in an application (i.e. code that is not instantiated in the manager file). In other words,
there is no mechanism available to the user to force the tool to generate the dynamic
components (i.e. dynamic kernels) unless we instantiate them in a Maxeler project.
However, instantiating them is not an option as the tool will generate additional connections for those kernels that can not be removed by an after-processing mechanism.

Another issue is that the MaxCompiler strictly compiles only one manager file (see Section 3.1). This makes sense for a fully static FPGA systems. However, for run-time reconfigurable systems that introduce not only kernels as partial modules, but also multi-kernel blocks, a user will need a new HLS description mechanism. An example of a multi-kernel block is depicted in Figure 3.5, however a more analytical description of those blocks is going to be given in the next section. Therefore, we need to introduce multiple manager-like functions that allow the instantiation of an arbitrary number of kernels and arbitrary connections amongst those kernels. Thus, there is a necessity for supporting arbitrary connections within each set of kernels (e.g. multi-kernel group that acts as a partial module entity).

Lastly, we have to provide a way to allow the creation and the definition of dynamic connections in the Maxeler manager. This is not trivial, if we consider different run interfaces that are available in the C interface code. In the original Maxeler approach, a user could set different interfaces to be orchestrated from software, as mentioned in the previous section. In a Maxeler design, a user can operate multiple streams. However, some streams and some parts of the design may be independent from the rest of the system, while not being mutually exclusive. An example of this is shown in Figure 3.4, in which the implementation can have an ICAP module and static kernel 1 that are independent from the reconfigurable pipeline. A user can choose to load data only to the ICAP module, without triggering the other three streams or just perform a computation on the static kernel 2. Note that this problem is different than partial reconfiguration, as the three depicted streams in the figure are not mutually exclusive to each other. Due to the dynamics available through run-time reconfiguration, a user needs to be able to declare static and dynamic connections in the description of the dynamic system. To achieve this kind of description, additional functions need to be included wherever it is necessary in MaxJ.

3.2.2 Extension in practice

To mitigate the aforementioned limitations it was decided to introduce the concept of PRGroups. A PRGroup is a manager-like Java function that can contain from one to an arbitrary number of kernels. In addition, a PRGroup is described as part of a Java function, that can contain the same capabilities as a manager constructor in the original Maxeler model. In Maxeler, a constructor is used to instantiate kernels and connections
CHAPTER 3. FROM STATIC TO DYNAMIC HLS SPECIFICATIONS

Figure 3.4: Block diagram describing a PRManager. The PRManager is split into two parts: the static instances and the dynamic instances. A PRManager can contain an arbitrary number of connections and an arbitrary number of static kernels, with a Reconfigurable Region module and an ICAP module. The static instances of the PRManager can also contain full static connections like static kernel 1, where its input and output does not relate with the dynamic system. Finally, a single stream needs to be instantiated to the ICAP module for reconfiguration. In addition, the PRManager can contain multiple PRGroups, as the dynamic instances. The only requirements that the PRGroups share the same reconfigurable region and provide the same physical interface. The interface needs to match all PRGroups and the Reconfigurable region instantiation.

for a manager. Thus, a PRGroup will follow a similar approach to directly comply with the Maxeler programming model. A PRGroup can have arbitrary internal connections between the instantiated kernels. Those internal connections will be set by a Maxeler user. The PRGroups allow a user to define building blocks containing any number of kernels and also switching of accelerator modules/ kernels in a dynamic application.

PRGroups are instantiated in the PRManager, which can contain an arbitrary number of PRGroups as well as connections to and from the static part of the design. Alongside the PRGroups, the PRManager contains the static kernels that constitute the static part of the dynamic implementation. A PRManager example case is depicted in Figure 3.4.

The static instances and the PRGroups constitute a dynamic design. The PRManager contains a description of the dynamic design, as shown in Figure 3.4. The static system is composed of the peripheral modules (e.g., for DDR memory or PCIe), static
Figure 3.5: Example block diagrams of PRGroups. All depicted PRGroups have the same interface 3 inputs and 2 outputs. A PRGroup can form a single pipeline like PRGroup_0 or a more complicated group like PRGroup_1. A kernel that is instantiated in a PRGroup can be also instantiated in another group as shown in PRGroup_0 and PRGroup_1 for kernel 2. Finally, PRGroup_N is a single kernel in this group and it is used to depict that in order to define a single kernelled partial module, it needs to be defined as a group.
define the interfaces of the static system. This includes all the inputs and outputs from
and to PCIe and DDR memory. In Figure 3.4, the output streams of the reconfigurable
region need to be instantiated both by setDynamicStream and setStream.

The PRManager must contain at least 2 static instances: the reconfigurable region
and the ICAP core. The reconfigurable region is given through our Maxeler extension
library and it is implemented as an empty module with a connection from the input
to the output. The reason for this is that we need a way to force the tool to generate
the necessary physical connections for the region. Furthermore, the ICAP core is nec-
essary to load the partial module configuration bitstream into the FPGA. In the here
presented system, there may exist accelerator kernels that will not be swapped at run-
time, as illustrated in Figure 3.4. Those accelerators are called static kernels and they
will be part of the static system. Static kernels are useful for parallel processing and
pre or post-processing for the partial modules. Static kernels are kernels that are not
specified as a PRGroups in the PRManager. Note that a system may provide multiple
reconfigurable regions. To achieve this, a user needs to instantiate multiple instances
of the reconfigurable region provided by our extension.

An example of a PRManager is shown in Figure 3.4 and examples of different
PRGroups of this example are depicted in Figure 3.5. The PRManager is initially split
in static instances and dynamic instances. In Figure 3.4 as parts of the static instances,
a user can instantiate any kernel entirely, as done in the original Maxeler approach. The
two red boxed kernels in Figure 3.4 are instantiated static kernels that will be parts of
the static system. As shown, static kernel 1 is also completely independent from the re-
configurable pipeline and acts as a parallel concurrent processing unit. Additionally, a
user needs to define the connections to and from the reconfigurable region that will also
define and constrain the connections in the PRGroups. Those connections correspond
to the dynamic connections, which are established by the setDynamicStream function,
while the streams from and to the PCIe or memory interface are called static connec-
tions are set by the setStream function. Furthermore, an ICAP module is instantiated
in the figure and it is connected with a dedicated stream to forward the configuration
data of the partial modules to the device for reconfiguration.

In the dynamic instances part of the PRManager in Figure 3.4, a user can define
any number of PRGroups. The PRGroups are independent amongst each other and
they follow the same interface specifications as the reconfigurable region. An exam-
ple of three PRGroups is illustrated in Figure 3.5. In the static Maxeler approach, the
3.3. DYNAMIC MAXELER MODEL

kernels and the connections of a manager are described by a Java constructor. Subsequently, a PRGroup in our approach can contain anything that an original manager Java constructor can contain. This way, we guarantee that a PRGroup can include different connections and one or more instantiated kernels.

A PRGroup must contain at least one kernel, as shown for PRGroup_N. This was the model when introducing the dynamic approach where a single kernel was assumed as a PRGroup to be reconfigurable. In addition, a PRGroup can contain a pipeline of kernels like PRGroup_0 in Figure 3.5. In that example, we can see that the kernels can have different interfaces between them, as shown in PRGroup_0 between kernel_0 to kernel_1 and between kernel_1 and kernel_2. Moreover, a user can instantiate the same kernels in more than one PRGroup, as done for kernel_2 in PRGroup0 and PRGroup_1. Last but not least, a domain expert can create more complex PRGroups, like PRGroup_1, using kernels with different interfaces and streams to create more complex and resource-intensive PRGroups.

3.3 Dynamic Maxeler model

After the programming of the PRManager, the PRGroups and the existing accelerators, a user can run an external flow that processes the extended version of MaxJ. This will be based on a preprocessor that will be introduced in the following subsection. The preprocessor works as a text processing engine in which the functions described in the previous section (e.g. PRGroup, setDynamicStream) will be used as predefined keywords that relate to different functions. The functions will readapt the given MaxJ code accordingly and orchestrate the extended MaxJ described application into multiple dynamic components. The included Java functions have no functionality in the original Maxeler flow, but when preprocessed they will control the generation of the system components used by the dynamically reconfigurable system. The preprocessor is important to solve the problem of not being able to force the tool to generate subcomponents in the original Maxeler approach which are essential for a dynamic application. In addition, this section describes how the preprocessor is implemented into the front-end flow, which includes both the preprocessor, the language extension, as well as the existing Maxeler toolflow.
3.3.1 MaxJ preprocessor

The here proposed language extension readapts the idea behind the Maxeler manager and introduces a PRManager that contains the extended functionality of a dynamic design. A PRManager allows multiple managers to generate different kernels and connections. To achieve that, we start from a main project and automatically split it into multiple smaller projects (for static and all the PRGroups).

This is performed by a preprocessor that starts by detecting the PRGroups of the specified project, as shown in Figure 3.6. For each one of the PRGroups, the tool creates a Maxeler project directory. Those Maxeler projects are only used by the tool to generate the RTL files of the corresponding kernels. The projects will use the connections defined in the manager interface with the extended function setDynamicStream (see Section 3.2.2), in order to describe the interface of all the PRGroups. The setStream functions will set the connections with the static system.

![Figure 3.6: Front-end implementation in a flow diagram. The input is the initial project with a prefix name of PROJECT_NAME and the preprocessor will generate the corresponding projects (project directory outputs are depicted as 3d boxes). After the project generation, the MaxCompiler will process the code and generate an RTL project version of each project generated by the MaxJ preprocessor. Last, the preprocessor will generate the final RTL top level entities of each existing group (left side of the Figure). The outputs of the front-end processing are several top level VHDL files for each group and several RTL projects generated by the MaxCompiler.](image)

In addition, the static part of the PRManager will be used to generate a static
3.3. DYNAMIC MAXELER MODEL

project, which will be the project which will later control the dynamic system. This ensures that the programmer’s code remains unchanged. The preprocessor concept allows for specifying the static system and all the kernels in one project that can then be automatically compiled without touching the original Maxeler compiler or the FPGA vendor tools.

3.3.2 Front-end description

After the project generation, the preprocessor runs the MaxCompiler for all the projects until the end of the RTL generation step in the Maxeler toolflow, as shown in Figure 3.6. During the RTL generation of the MaxCompiler there may be errors in the design. The error detection remains intact as in the original Maxeler flow. The reason for this is that the sub-projects of a dynamic project, that are generated by the preprocessor, are in practice multiple static projects. The main difference is that in our flow all the generated sub-projects are connected in our flow and they will output possible error messages in one log file in the initial project directory in case of an error. At the same time, the static project will act as the root project as the user can manage the inputs and outputs of the application from the static system and the loading of modules through ICAP. The output of the MaxCompiler will be several RTL projects from the sub-projects, generated by the preprocessor (see Figure 3.6).

The last step performed by the preprocessor is to generate the internal connections for the PRGroups in a VHDL top level file. This subprocess is illustrated in the left side of Figure 3.6. The preprocessor will read the manager files in MaxJ. Then, it will create a list of the connections of the kernels and it will generate connections between the RTL generated kernels in VHDL. This will create the final PRGroup top level entities in VHDL code. All code transformations that are performed though our toolflow happen on the RTL code that is generated by the MaxCompiler and the entire process does not require any user interaction. A user of this flow will only interact through an HLS specification of the system (MaxJ).

Figure 3.6 summarizes the front-end preprocessing flow. The tool starts by processing the corresponding manager and it generates dummy PRGroup projects containing all the necessary files and their corresponding managers as well as a static Maxeler project. The initial project, which contains the PRManager, is being kept at the same main workspace for possible future modifications. In Figure 3.6, the static project does not require an external top level description and it will only be processed in the next stages of this toolflow. The next stages will be described in the following Chapter 4.
and Chapter 5 and the static part will generate the corresponding static bitstream of the
dynamic project depicted in Figure 3.6.

3.4 Design choices

Through the here proposed language extension, a user can generate not only partial
modules, but groups of kernels that constitute a partial module (see Figure 3.5). How-
ever, there may be kernels constituting PRGroups that can perform computations in
any permutation order. Thus, if an application contains $N$ kernels in different groups,
generating every permutation will result in $N!$ groups of kernels to be implemented.
This solution is not scalable and will in some cases be too time consuming for the
FPGA implementation.

Therefore, it should be foreseen, to generate an individual chaining of modules
that requires only $N$ implementations, if the configuration bitstreams of those kernels
are relocatable. This section describes the implementation of individual kernels from
PRGroups. The generation of each individual kernel is called software design choices,
while the relocation of the generated accelerator bitstreams will be described in Sec-
tion 5.2.

It was decided to provide the users with more flexibility in terms of kernel combi-
nations. The approach which was followed in this work is to allow the generation of
all the kernels instantiated in PRGroups as individual partial modules for future usage
(assuming relocatable bitstreams). The above occurs on top of the PRGroup genera-
tion. The implemented partial modules can be chained directly by the run-time system.
Those kernels will be called unique kernels. The here proposed approach foresees to
have only one physical implementation of a kernel (e.g. given multiple instances of the
same kernel in the same or in different PRGroups) and by using module relocation, a
kernel can be executed at different locations on the FPGA.

With this, the total maximum amount of partial modules is given by the following
function where the variable $PM$ corresponds to the number of partial modules, $U$ is
the number of unique kernels to be implemented and $G$ the number of PRGroups,
including the single kernel groups.

$$PM = G + U$$

The ability to reuse a physical implementation at different locations will not only
improve CAD tool time but allows a system to compose stream processing pipelines (from daisy-chained relocatable kernel bitstreams), provided that they are available at runtime. Through this flow, the generated partial modules, both kernel groups and unique kernels, will act as individual load-and-run modules. However in hardware, we can allow additional design choices in terms of placement, that will be transparent for the designer. This will be introduced in Chapter 5.

3.5 Summary

This section described how we extended the functionality of the Maxeler tool for generating dynamically reconfigurable projects, while preserving the Maxeler user experience. To extend this functionality, we needed to detect the limiting factors for generating dynamic systems in the current Maxeler toolflow and introduce a language extension to provide solutions. For parsing the extended version of MaxJ, a preprocessor parses the MaxJ code to generate different Maxeler projects and the MaxCompiler is used to generate the corresponding VHDL code for each one of those generated projects. The next chapter will focus on the mid-level processing of the generated VHDL code.
Chapter 4

RTL-level Specifications

A domain expert that uses an HLS tool that allows partial reconfiguration should only describe the desired dynamic system through a high-level approach. However, to allow this level of abstraction from an HLS approach, this thesis proposes a mid-level process to modify the generated code that will constitute the dynamic system accordingly, in the background. This chapter focuses on the adaptation of the Maxeler-generated RTL code that is done in the background, in order to create code that can be used by the vendor tool for physical implementation.

The first section describes the post MaxCompiler flow. Then, we analyze the modifications necessary to adapt the generated code to our dynamic approach. The modifications include both the static system, the PRGroups and the individual kernels.

4.1 Internal Maxeler architecture

When the Maxeler HLS compiler generates RTL code, it provides clear hierarchies with well-defined communication interfaces. We use this observation by our automated compilation flow that splits the generated project into a static part and accelerators. A general project can be split in two parts: the Maxeler surrounding system and the manager, which contains the corresponding code generated by the MaxJ manager file. The manager contains all the kernels that are instantiated in the MaxJ code. Each one of the kernels belongs to a sub-hierarchy of the manager that also includes the internal connections between the kernels. The connections that are declared in the manager file as external connections are generated as part of the Maxeler surrounding system. Our flow extracts the hierarchy that belongs to the static system (which will contain all
4.1. INTERNAL MAXELER ARCHITECTURE

Figure 4.1: High level view of Maxeler generated system, before modifications. The example contains the Maxeler surrounding system and a MaxJ kernel.

the I/Os of the system) and the hierarchies that belong to the manager, including the kernels.

Figure 4.1 depicts a generated example system. Its architecture consists of the Maxeler surrounding system, which contains memory controllers, the controls of the data I/O and the main interface that transfers data from CPU to FPGA and vice versa though PCIe or an existing DDR memory interface. The example depicted in Figure 4.1 contains a manager hierarchy, which is the RTL product of the MaxCompiler for a specific project. This RTL manager can contain one or more kernels that are connected to the Maxeler surrounding system via internal streams for input and output. In relation with the language extension, presented in Chapter 3, the manager of static part will contain all the static kernels, including the partial region and the ICAP module. For the PRGroups, the manager will contain the RTL description of its corresponding kernels, as generated by the MaxCompiler.

In Section 3.3.1, we described how we split the PRManager into distinct parts using the MaxJ language. After the preprocessor finishes its internal process for each one of the generated parts (i.e. static part and PRGroups), a mid-level code generator takes action. This code generator modifies and generates RTL code depending on the
Figure 4.2: High-level view of the static part of a dynamic design. The reconfigurable region acts as a dummy kernel to define the connections to the dynamic system and it will be replaced by a project-specific reconfigurable region with location constraints and in-region connections. The custom HDL entity contains the ICAP and its control.

contents of the initial dynamic project. The code generator will modify the static part, the PRGroups and the kernels differently. The next sections will describe the changes on the static part, the accelerators and the PRGroups.

4.2 Static system

The static part is the core of a dynamic design. As MaxCompiler outputs the static design through the static MaxJ project, the manager is clearly encapsulated in VHDL entities. The generated manager will contain all the static kernels, the reconfigurable region and the ICAP module. However, reconfigurable region needs to be adapted at the RTL level before progressing to the implementation phase. The processor applies location constraints for the reconfigurable region and instantiates a new RTL description for this region. The reconfigurable region, as described in Section 3.2.2 is initially
generated empty. The above is necessary due to the low-level design of the reconfigurable region that requires strict location constraints and specific netlist files that cannot directly be instantiated in the MaxJ programming model. Additionally, we should take into consideration the size of the region that will host the partial modules. In order to calculate and instantiate the physical instance of the region, we need a way to calculate the resources needed by the partial modules.

Figure 4.2 shows a generated static system. Its architecture consists of the Maxeler surrounding system that forwards the data from CPU to FPGA and vice versa through PCIe or DDR memory interfaces. For simplicity, the example depicted in Figure 4.2 contains only one partial region and a custom HDL instance in the manager. The example could have had more kernels instantiated statically in the manager, as described in Chapter 3.

Partial reconfiguration is performed through the Maxeler memory controller towards the ICAP, which is included in Maxeler’s surrounding system and which sends partial configuration bitstreams to the ICAP ports of the FPGA. The ICAP is instantiated inside a Custom HDL wrapper that forwards input data from the Maxeler interface to the ICAP. Within the Custom HDL wrapper we use a control to handle the I/O of the ICAP stream. Note, that the Custom HDL ICAP component should be already included from the design phase. This control is responsible for little-Endian bitshifting, which is necessary for the ICAP component. The control module follows the Maxeler stream protocol, as it is instantiated in the MaxJ version of the project.

The reconfigurable region acts a placeholder module to define the interface with the dynamic accelerators. As described in the previous chapter, the reconfigurable region file is an empty loopback MaxJ file and it is used to generate the surrounding connections. The wrapper will later be used to host the reconfigurable region itself, as part of the static implementation. The MaxJ version of the reconfigurable region is automatically replaced with a project-specific reconfigurable region which is performed by the code generator. The reason that we can not define the region itself within the MaxJ code is due to the high-level of abstraction from the Maxeler framework. Thus, low-level constraints, for example location constraints for the reconfigurable region needs or specific netlist files that describe the implementation of the reconfigurable region, cannot be instantiated directly in MaxJ code.

The reconfigurable region is implemented as a loopback device. By loopback device, we mean that the inputs of the reconfigurable region are routed across the fabric (i.e. the instantiated location of the reconfigurable region) that it occupies, then this
input performs a U-turn and follows a backward path towards the Maxeler surrounding system. As a default in this PhD thesis, the region interface is predefined at a 512-bit wide datapath, plus some extra handshaking signals. However, a user can originally define the desirable bit-width of the system, with a maximum of 512 bits for input data and 512-bits for output data. A datapath of up to 512-bit was selected as this allows for saturating a DDR memory channel. In case that a user does not need all the bandwidth, our tool will automatically ground the remaining signals. In the case of wider datapath (multiple of 512), the region would have to be wider in term of FPGA clock regions or the congestion would be significantly higher that would affect the timing of the implementation.

Our code generator instantiates the region in the MaxJ generated RTL design. The instantiation occurs by coping the necessary files in the project directory and by setting location primitives for the region itself in the User Constraint File (ucf). Section 5.1 describes how the partial region is designed and created in more detail. Each stream towards the reconfigurable region contains an I/O control, as shown in Figure 4.4. This I/O control is shared amongst all the kernels that will be placed in the region for the specific stream (as in Figure 4.4), instead of using an I/O control between each kernel (as shown in Figure 4.3).

### 4.3 Kernels and Internal Communication

In general, the reconfiguration of modules can save resources on mutually exclusive applications, improve the implementation time and enhance parallelism. However, MaxCompiler does not output RTL code for the accelerators that can be implemented as reconfigurable modules. Thus, after the MaxCompiler outputs the RTL code, an additional preprocessing mechanism is needed for the generated kernels.

The reconfigurable accelerators need to communicate both with the rest of the reconfigurable accelerators as well as the static part. Hence, a predefined interface should be set both on the input and the output side of each implementation. In addition, the interface in the input and the output of each accelerator should match, to allow interchangeable placement in the reconfigurable region. However, this task has to incorporate the existing interface from Maxeler, while taking into consideration the Maxeler communication protocol between kernels. In order to produce dynamic accelerators from the current statically generated RTL code produced by Maxeler, our process needs to support:
• A generic interface for the accelerators allowing the predefined Maxeler handshaking protocols on both the input and the output to communicate with the static part.

• Matching interfaces on the input and the output for multi-placement of accelerators. (e.g., for daisy-chaining of accelerators)

• Automatically creating a top level VHDL entity that connects the predefined PRGroups by the users, as specified by a user through the MaxJ description.

• Allow design alternatives of accelerators that belong to PRGroups to be implemented as individual blocks.

The kernel communication from Maxeler is implemented based on a handshaking protocol, as shown by the depicted connections in Figure 4.3. The interface protocol of the modules on each input stream contain a data, empty, an almost empty and a read bit, while the output uses a data, stall and a valid bit. In order to guarantee independence and relocatability amongst modules, we need to define and create a generic interface that will be followed by all kernels. To understand the basic communication principals of the Maxeler streaming protocol, we observed how Maxeler handles the chaining of the implemented accelerators. We found that Maxeler uses a FIFO semantic component (chaining component) to connect kernels using the same handshaking and flow control mechanisms as used for the communication with the Maxeler interface. Thus, it was decided to follow the same approach in order to directly comply with the Maxeler communication protocol. Figure 4.3 shows the chaining of two MaxJ generated kernels using a chaining component, which is generated by the Maxeler compiler. This component contains a small FIFO and an I/O control block.

By separating the chaining component into two parts, we can integrate the FIFO in the kernel and keep the I/O control as part of the static system. An overview of the proposed architecture for the module is shown in Figure 4.4. As we can see, the handshaking interface of the inputs and the outputs is exactly the same. This is essential because we consider that basic blocks with the same interface can now be placed arbitrarily within the reconfigurable region to create a pipeline to form more complicated accelerators. Figure 4.4 also depicts an example with two individual Maxeler kernels that can be concatenated together in one shared reconfigurable region.

The control block of the initial chaining component from Maxeler is used in the static part to handle the I/O of the reconfigurable region. The proposed design offers
Figure 4.3: Initial module chaining in Maxeler using a chaining component. The chaining component is part of each connection in the Maxeler generated RTL description. As it is a fundamental component in the generated RTL code by Maxeler, its interface is used to set a template for our RTL-level connections.

Figure 4.4: Modified kernels after the chaining module split. Kernels contain an integrated FIFO that existed in the initial chaining module while its I/O control is placed in the static system. The yellow border defines a PRGroup or a unique kernel.

lower resource utilization, as the original Maxeler approach used a chaining component at each input of all the implemented kernels that were included in the initial manager file. We observed that only the FIFO is required for each kernel and not the whole chaining component. Thus, our design replicates just the FIFO inside the modules, as we need only one control block for each input of the reconfigurable region and share the control block amongst multiple kernels. Note that using only one dedicated control block for the design, there is a slightly less resource utilization in the design without impacting the functionality.

After this step, the tool outputs all the existing kernels as individual accelerators, with the same handshaking interface on the input and the output. Initially, each input of a Maxeler generated kernel had FIFO state signals empty, almost empty and read as its handshaking protocol. After our modifications for each kernel, the handshaking
4.3. KERNELS AND INTERNAL COMMUNICATION

Figure 4.5: PR code generator flow diagram. The tool starts by extracting the generated code from Maxeler and creates a project directory for each one of the existing Maxeler projects (including dummies). The tool automatically detects the kernels in each group and performs the necessary changes before grouping them back into the final RTL version of the group. In the static system, the tool adds an I/O control on each input and the corresponding reconfigurable region. Each unique kernel of the groups is stored in a directory (bottom right corner) to be used as external partial modules, that act as building blocks for future use.

signals of those inputs will work with **valid and stall**, as it is on the output side of each Maxeler accelerator. Those accelerators can now be used as individual partial modules and will be generated by our flow automatically, if the user desires to have them as partial modules, even if they belong to PRGroup.

The last step is to connect the kernels to form a PRGroup with the new interfaces between them. This is done through the MaxJ language by reading the PRGroups description in the initial manager file. The subprocess starts by recording the kernels, their inputs and their connection from the MaxJ preprocessor. The values are recorded in lists (in Python) containing all kernels with a unique ID, each signal connection (ID_connection_ID) and the interfaces of each accelerator in a PRGroup. Then, the
mid-level processing tool parses the connections and generates the corresponding signals that are necessary in VHDL. Each recorded connection signal starts from one input and can send data to multiple other components. We use this principal to write the PRGroup’s VHDL top-level with its interface entity and its functionality, containing all the components and connections for each kernel, including the connection signal declaration that is necessary in VHDL language. The top-level entity has again the same interface protocol, consisting of data, valid and stall signals for both input and output.

Figure 4.5 illustrates the mid level processing that occurs, following the same example, depicted in Figure 3.5 and Figure 3.6. In those figures, the PRGroups contain the same number of kernels. After the RTL code extraction from the dummy Maxeler projects, the PR code generator identifies and extracts the whole hierarchy of the existing kernels. The rest of the PRGroup project is not necessary after this step. The tool also applies the modifications to all the detected kernels and creates a partial module version of the kernel (i.e. PM Kernels in the figure). After the modifications, the unique kernels and their hierarchies are saved in a dedicated project directory (called Unique kernel directory in this implementation) of the project for making extra design alternatives available to the user. Additionally, the tool generates the PRGroup partial modules in their corresponding RTL version, by generating the top-level HDL code from the detected connections of the front-end processing (i.e. the MaxJ preprocessor).

The mid-level process ends by outputing the resources needed for each entity, either for a PRGroup or a unique kernel. We use those resource requirements to calculate the minimum resources needed for the reconfigurable region. Note that the resources of the largest PRGroup alone may not define the minimum resources needed for the reconfigurable region. For example, a PRGroup that needs most of the LUTs is the largest and, there may be, for example, a PRGroup that needs more DSPs than the largest PRGroup in terms of LUTs. Thus, the minimum requirements of a reconfigurable region are the maximum number of resources in LUTs, DSPs and BRAMs in all the existing kernels and PRGroups. For the PRGroups and kernels, the resources of each entity will be used in the back-end processing to calculate certain location primitives and constraints.

The primitive-wise counting of resources defines the smallest reconfigurable region that suits all PRGroups. For allowing future extensions, the designer may choose a
larger reconfigurable region, while recompilation and reimplementation of the whole project it is not required, in case of lack of space.

4.4 From MaxJ towards implementation

As, we mentioned in Section 3.1, in order to generate a static design with the original Maxeler flow, the Maxeler tool needs a manager file and the instantiated kernels. The above entities are written in MaxJ, as Figure 4.6 depicts. The implementation starts by running the MaxCompiler, that converts MaxJ to the corresponding RTL code. As Figure 4.6 illustrates, the generated RTL code follows the ISE toolchain until the bitfile generation, that will later be a part of the final Maxfile.

The here proposed toolflow is shown in Figure 4.7 compared to the existing static approach by Maxeler in Figure 4.6. In this figure, each level of processing (front, mid, back-end) is drawn in a blue cycle, while their corresponding output is marked by a dashed-line box. This figure presents a detailed overview of the here proposed implementation, by depicting the inputs and outputs of each processing stage. This section summarizes Chapter 3 and Chapter 4 before focusing on physical implementation practices and details in Chapter 5.

In this flow, the HDL generation toll is written in Python that automatically generates the dynamic design. As in the static approach, the user also programs the manager and the kernel files in MaxJ. The tool starts by generating individual projects based on the PRManager, which is an extended version of the Maxeler manager file. This is done by the front-end process and its internal process is described in greater detail in Chapter 3. MaxCompiler is used in each one of the generated projects to output the RTL code of every project. The RTL code including the IP cores given as netlists and constraint files, are extracted and they are used to proceed to the next level of processing, which is the mid-level. The mid-level processing is responsible for the modifying the RTL code and to include the additional template files that generates, as described in this chapter. As mentioned in the previous chapter, the output of the process are implementation-ready entities of partial modules and the static system.

As a final step, our toolflow implements each one of the entities in parallel to generate the final bitstreams. This is done by the back-end processing, which uses academic tools and Xilinx tools to implement the existing RTL code. The reason that we use external academic tools is to achieve box constrained implementations of partial modules, which will allow more flexibility on the dynamic system, while
preserving clear interfaces for the communication between the static system and the partial modules. The back-end process will output and generate the static bitstream and the partial bitstreams, as depicted in Figure 4.7. The static bitstream is a full bitstream, which is used to describe the full implementation of the static system. As it is done in the original static approach by the Maxeler toolflow, the full bitstream will be used to generate the final Maxfile of the project. The partial bitstreams will be stored in a directory that will later constitute the partial bitstream library of the project. The partial bitstream library will contain all the partial bitfiles for all the PRGroups and unique kernels. As a conclusion, the original Maxeler approach has only one output bitstream, while our approach has one full bitstream and a library of partial ones. Chapter 5 will describe the back-end processing works in depth and analyzes how we transit from the RTL code that is generated by the mid-level processing to the final bitstreams.
4.4 FROM MAXJ TOWARDS IMPLEMENTATION

Figure 4.7: MaxJ to MaxFile in the here proposed dynamic approach. The figure depicts a more complex implementation that the existing static Maxeler implementation depicted in Figure 4.6, as we have more entities to process in the dynamic implementation. The two figures have the same initial implementation. Both approaches start from MaxJ and include programming the manager and the kernel files, before being processed by the front-end tool. The output of each additional process is marked by a dashed line box. The outputs of the front-end process are being processed by the MaxCompiler until the RTL generation. The generated RTL will be modified by the mid-level processing tool to create the implementation-ready entities to be sent to the back-end process. Finally, the output of the back-end process is a library of partial modules and a static bitstream, which, as in the static approach, will complete the final Maxfile of the project.
Chapter 5

Low-level aspects

This chapter covers the low-level engineering aspects of a partially reconfigurable design, that is usually done by an FPGA expert. In order to introduce a flow usable by application domain experts, automatic tools have to be used on the background with extra scripts to modify the code running on the tools accordingly. This chapter focuses mostly on the implementation after generating the necessary RTL files and projects (which is described in Chapter 4). For the implementation phase, the here presented flow uses academic and the Xilinx vendor tools (i.e. ISE).

This chapter describes the toolflow for creating dynamically reconfigurable systems to implement the modified components (see Chapter 4) to build a partially reconfigurable system. This chapter provides an analysis on the implementation flow and the background tools that we use to create our dynamic system. In addition, we explain the low-level details of our approach and the benefits that rise from the back-end processing to the application domain expert.

5.1 Static system

This section focuses on the implementation of the core static system architecture of the dynamic design. We start the introduction by describing the architecture of the reconfigurable region and how it fits to the final static part of our design. After this, the background process which generates the static system will be presented.
5.1. Static System

5.1.1 Reconfigurable region

The first step to create a reconfigurable region is to decide the position that it should be placed on the FPGA fabric. In order to take this decision for multiple Maxeler projects, we observed how Maxeler maps and implements designs in its original flow. From this, it was noticed that Maxeler is almost entirely not using the top and bottom corners of the device. This occurs because in the device we use all the communication specific primitives (e.g. for DDR3 and PCIe) are located in the center of the chip. Thus, we decided to place the reconfigurable regions on the corners of the device. The designer can generalize this approach by determining, how many regions are required based on the utilization of the static and the dynamic component. Those locations are important also for our approach of partial reconfiguration, as they can be isolated from the static accelerators and the I/O infrastructure. The here presented approach, focuses on accelerators that function independently amongst them. The reconfigurable regions that will only be placed in the top corners of the device will share the same interface protocol for allowing relocation amongst regions. The above characteristic can fully utilize the existing partial modules, considering that none of the accelerators is implemented as a design specific block (in terms of location or specific reconfigurable region), but rather as a load-and-run FPGA function. By load-and-run, we mean that a component will be loaded, executed until the process terminates by any interface and resource matching project. At any point as we will describe later in this Chapter, the designer can decide to switch between applications.

The partial region represents the physical implementation of the dummy reconfigurable region that is instantiated by a user as described in Section 3.3.1. The wrapper will be replaced by the physically implemented region, as shown in Figure 4.2. To generate the region or regions for a project, we used the GoAhead tool. The tool will be analyzed in more detail in Section 5.5.

The whole process is automated for domain experts. In order to create reconfigurable region for given resource constraints, we automated the region generation in GoAhead via tool specific scripts. In Chapter 4, we mentioned that we calculate the minimum resources needed based on the existing PRGroups. In the case that we need more resources than available on the FPGA, the maximum number of resources (as mentioned in Section 4.3) is used to define the script implementing the reconfigurable region. This script will generate location specific primitives to be instantiated in the RTL implementation, as well as the region RTL description.

To constrain the data routing, we are using routing blockers. Routing blockers
function as preoccupied routing resources that constrain the interface into a predefined fashion. In other words, in a specified region GoAhead can set and name the amount of wires that will be used and connect those wires from a point A to a point B in the chip. This guarantees relocation with the region and regions. Then, we can generate blockers on the remaining wires, to guarantee that the routing of the static system does not cross any reconfigurable region. The blockers also guarantee a specific wiring template within the region across the full width of the chip. Additionally, macros constrain the interface of the static wires of a predefined reconfigurable region. The interface of those wires will be the final interface of the region itself. Furthermore, the clock signals are constrained such that all the designs (i.e. partial region and accelerators) use the same routed clock routing resources. To force the Xilinx tools to generate the desired clock connections, GoAhead generates a clock blocker, or multiple in case of multiple clock signals. The clock blocker will occupy a predefined clock spine and block the rest of the clock spines.

To achieve high clock rates of the static implementation, internal pipeline stages are instantiated within the region. The pipeline stages will also constrain the wires that are required by the predefined routing path for the reconfigurable region. Those wires will be marked as used on the GoAhead tool for generating the routing and clock blockers. Those pipeline stages are the only used primitives within the region. The rest of the region left is entirely empty and the pipeline stages can be overwritten, in the case that we need the resources for hosting an accelerator. Finally, the region is implemented as a loopback device. By loopback device, we mean that each one of the wires routes to the pipeline stages is routed in a regular pattern all the way across the side length of the region. Then, the wires perform a U-turn and route all the way back, while routing to backward pipeline stages as well. Similarly to the routing blockers, the clock blockers guarantee that in every subpart of the design (i.e. static part or accelerator) the same clock signals will be used for all the clocks. By constraining clock and internal routing, it is clear that relocation after the generation of the subparts will be based entirely on the resource available on any region.

An example of a reconfigurable region is depicted in Figure 5.1. In the figure, we have drawn wires (yellow wires) on the top and in the bottom that route from the static system located on the left beside the reconfigurable region all the way to the end of the chip (right border) and return in the same fashion back to the static system infrastructure. Each of the pipeline stages is marked by a red box and they are distributed equally on the width of the reconfigurable region. Note that each box
Figure 5.1: Fully placed and routed reconfigurable region, as shown by the FPGA editor tool. The region interface constitutes of 512-bits for the input and 512-bits for the output, plus 16-bits of input and 16-bits of output for implementing handshaking signals for the communication of the region and the rest of the system. The signal path is regularly routed across the span of the reconfigurable region. Additionally, pipeline stages are instantiated in the region to improve timing and constrain the routing within the region (red boxed components in figure). Those pipeline stages are double, which means that the pipelining occurs both on the front and backwards path.

corresponds to 2 pipeline stages, one for the forward path and one for the backward. However the pipeline stages can be overwritten while loading a reconfigurable module in the reconfigurable region. The interface of the region is 512-bits for the input and 512 bits for the output. In addition, the region offers some handshaking signals that control the flow of data through the reconfigurable region.

### 5.1.2 Static implementation

After generating the reconfigurable region that is instantiated in the static system, we can run the combined toolflow of GoAhead and Xilinx commands to create the final bitfile of the static system. Figure 5.2 illustrates a very high-level view of the static part physical implementation flow. After all the placement constraints and blocker files are
CHAPTER 5. LOW-LEVEL ASPECTS

Figure 5.2: Low-level implementation of the static system. The toolflow starts from the static RTL part that was generated by the mid-level processing. GoAhead generates a VHDL version of the reconfigurable region and its corresponding area constraints. Additionally, routing blockers are used to constrain routing in a predefined fashion to match the routing of the generated accelerators. The Xilinx vendor tools are used to map and place and route the final design and ultimately generate the configuration bitfile.

As shown in Figure 5.2, the flow remains the same with the Xilinx flow until after the mapping phase. The region and the location constraints are pre-instantiated through the mid-level processing. After mapping of the design, design-specific routing blockers and clock blockers are added. This is done by the tool GoAhead that is controlled through its scripting interface. Note that for Virtex 6 FPGAs, as available on the used Max3 system, we use the XDL language to perform netlist modifications. After adding the blockers, the first step is to route the clock signals one by one, in the case the system provides multiple clock signals to the module hosted in the reconfigurable region. This is performed through scripting commands for the Xilinx FPGA Editor tool. After routing all the signals, the toolflow continues with running the Place and Route command. When we have the final design, we can delete the blockers and

generated, the ISE flow is used to generate a full bitfile of the static part of the dynamic system. A fully placed and routed region is depicted in Figure 5.3.
we can then generate the full static bitstream. We keep the routed netlist for future placement options that are presented in the next sections.

Figure 5.3 shows a simple routed static system with a reconfigurable region placed in the top-right corner of the device. Most of the surrounding system is placed and routed without any extra constraints in the back-end flow. It is placed in the center of the chip, as this provides direct access to the I/O pins used for PCIe and DDR memory, as well as communicating with the region/regions that will be placed in the corners.
Figure 5.4: Low-level implementation of MaxJ generated accelerators. The toolflow starts by using the modified RTL accelerators that the mid-level produces. The Bounding Box Generator (see Section 5.5) is used to bound the modules into rectangular boxes and to calculate possible placement positions for those boxes. This occurs by the given resources for each accelerator or PRGroup. GoAhead is used to generate the constraints for a predefined bounding box. Alongside the placement constraints, GoAhead generates the routing and clock blockers of the accelerators. After placing and routing the design, its netlist can be saved in a library containing all the netlists of the accelerators of a project.

5.2 Partial module implementation

In Chapter 4, we have introduced the RTL modifications that are necessary for the implementation of partially reconfigurable modules, occurring on the back-end process. The output of the mid-level process are multiple projects containing PRGroups providing an RTL description of one accelerator or one PRGroup each and their corresponding resource requirements. Those projects can be directly implemented without any further modifications.

Figure 5.4 illustrates the automatic flow of the module generation. Starting from the modified generated RTL code of the kernel and the pre-calculated resources for each one of the kernels, Bounding Box Generator (see Section 5.5) calculates the exact module bounding boxes given utilization (i.e. the number of LUTs, DSPs, and BRAMs used). Note that the Bounding Box Generator tool that searches 1) for multiple bounding boxes for each one of the kernels and 2) for multiple placement positions
for each one of the calculated bounding boxes. Then, the tool decides the most suitable bounding box based on internal heuristics (such as the minimum fragmentation or the maximum number of placement/routing). After defining a kernel’s bounding box, the toolflow automatically generates placement constraints and blockers around a predefined position, using the GoAhead tool. The generated blockers ensure that the module routing does not violate the bounding box borders.

One of our main challenges solved in this flow is to route through and backward of our module, precisely as we did in our reconfigurable region. This would allow in some cases configuring a kernel into the reconfigurable region without interfering with other modules that are executed in that reconfigurable region. This can occur to enable more design options for a user, without any requirement of reimplementing the design. In order to ensure this routing, we place a vertical series of registers before and after the module, that we call connection macros and act as the interface of our partial module. The connection macros have the same interface with the pipeline stages used in the static implementation (as previously described in Section 5.1). In order to implement the interface routing of the module, we leave "holes" in the blockers to allow routing both paths through the module. The aforementioned holes match the wires between the connection macros that are placed before and after the module. It should be mentioned that the clock signals are accordingly constrained to ensure that the clock routing will match exactly the routing used in the reconfigurable region. This is done by the GoAhead flow that ensures the routing through the same clock nets for both the kernel and the reconfigurable region.

Having the generated constraints in place, we can now run the Xilinx toolchain to fully map and place and route the module. The implementation here is very similar to the implementation of the static system. We start the implementation by using the Xilinx toolflow until after the mapping process. Then, we can add the routing and clock blockers, route the clocks before routing the rest of the design in the FPGA Editor tool and finally running Place and Route to fully route the design. After deleting the blockers, we need to perform an additional step compared to the static implementation. This additional step corresponds to the extraction of the routed netlist in a GoAhead netlist file. This extraction occurs only in the predefined bounding box and we do not include the connection macros before and after the module. Additionally, we can also generate a full bitstream from each module. The netlist and the bitstream of the module are saved and they going to be used in the final phase of the back-end process.

Figure 5.5 shows a fully placed and routed kernel in the FPGA Editor tool. This
Figure 5.5: Fully placed and routed PRGroup depicting the connection macros and the interface connections. The connection macros substitute surrounding static system based on the routing within the reconfigurable region. The zoomed figure shows that the routing is constrained to strictly stay inside the module bounding box (in this case the top border). However, within the modules, there is no constraint for the routing itself Furthermore, the interface wires on the left and the right side of the module physically placed design are implemented in such fashion that allows direct streaming to other modules or back to the static surrounding system.

fully implemented kernel can be extracted by GoAhead and for generating a full bit-stream of the implementation.
5.3 **Bitstream generation**

The aforementioned process results in several implementations of fully physically implemented accelerators (includes PRGroups and unique kernels). Those accelerators can be placed at different positions within the reconfigurable region that are initially calculated. Thus, we can place and route module netlists individually and independently from the static system (based on a predefined interface in terms of kernel I/O wires). The result of the process is a final static configuration bitstream and we can generate partial bitstreams using either bitstream manipulation with the BitMan tool [86] (see Section 5.5) or by using the differential bitstream methodology available by Xilinx [105]. The following subsections are describing both methodologies that cover the same function from two different angles, while each one of them can offer distinct features. Both techniques allow module relocation and replication (also proposed in [106]).

### 5.3.1 Bitstream extraction at netlist level

Having the placement positions of each kernel and the netlists of the static system and the kernels, we can place each kernel into the static system. As depicted in Figure 5.6, for each one of the kernels, we need to place its netlist (including place and route information) to the target position of our static system, with the help of the GoAhead tool. In order for the tool to process the Xilinx generated code, we need to make modifications to the XDL files. We use GoAhead to modify the Xilinx Design Language (XDL) file by instantiating the module in the position returned in the previous step. After the modifications we can revert the output XDL code to the NCD file (Xilinx netlist file description). The functions to convert between the file types are included in the Xilinx toolflow.

To generate the partial bitstream for the module, we first generate the static bitstream and the bitstream that contains the static system and the placed accelerator module in the reconfigurable region, generated by the placed netlist. During the differential bitstream generation process the two full bitstreams are compared between them, using the differential bitstream command from the Xilinx toolchain. The differences are saved in a bitfile, which corresponds to the partial bitfile of the specific module. This function is repeated for all the modules to generate multiple partial bitstreams.

A visual representation of this process is shown in Figure 5.7 that depicts three reconfigurable modules (marked with red boxes), placed in the reconfigurable region.
Figure 5.6: Extracting partial bitstreams using the netlist level approach. Here, the main tool is GoAHead that stitches netlists that were extracted, as described in Section 5.2. GoHAhead can place those netlists in the reconfigurable region. The tool works in the background in a "copy and paste" fashion in order to apply the changes to the static netlist. Having the new netlist with the placed module, we can extract a partial bitstream by using the Xilinx differential bitstream generation.

during the placement phase of our toolflow. The placed kernels are generated from different processes, but they can still be used in a stitched pipeline fashion to create a more complicated accelerator. The main advantage of the netlist approach is that static timing analysis using Xilinx vendor tool can only be achieved by the netlist level approach in the case of replication and relocation. With this feature, we can guarantee that any system created through bitstream manipulation at run-time will actually meet timing (including both setup and hold times). Thus, the timing analysis of this approach can provide the user the necessary timing check in case it is required.

5.3.2 Bit-level bitstream extraction

The bit-level approach is useful for online stitching of partial modules. In the here developed flow, the BitMan tool extracts the bitfile data in a specific physical location A that corresponds to the bounding box in the initial accelerator bitstream. This bitstream is then cut out and written to a physical location B of the second bitstream. The extracted bitstream can be saved as an individual partial bitstream that can be loaded in the second bitfile and program the device. Similarly, we can cut out a revert bitstream from a full bitstream. This bitstream corresponds to a piece of the reconfigurable region before placing any accelerators. This is implemented with BitMan in an initially
5.3. **BITSTREAM GENERATION**

Figure 5.7: Stitching of reconfigurable modules in a reconfigurable region in a pipelined fashion. The kernels placed in the reconfigurable region are marked with red boxes. Those kernels have been copied in a GoAhead netlist format to the previously routed static system. The kernels correspond to independent building blocks to be combined for solving more complex problems.

...empty region to extract the region as a partial bitfile. This bitfile can be loaded in order to clear out the region at any point during the processing or between two different processes to set back the design.

The process is depicted in Figure 5.8. The files needed for this process are two full bitstreams of the placed kernel and the static system. This uses the positions returned by the Bounding Box Generator (see Section 5.2) to place the module full bitstream in its placed position to one of the available in the region of the static full bitstream. The result of the process is a full bitstream and multiple partial bitstreams that constitute a library of partial modules in the initial project directory in the Maxeler user workspace.

The bitstream manipulation on BitMan allows for rapid stitching of modules at runtime without running any of the tools of the vendor Xilinx. That is the main benefit of the bit-level compared to the netlist level approach and that it allows online relocation. Moreover, this process is substantially faster than generating bitstreams from netlist level. More details will be given in Section 5.5 and Chapter 6.
CHAPTER 5. LOW-LEVEL ASPECTS

Figure 5.8: An approach for generating partial bitfiles from full module bitstreams. The tool extracts the module specific information given by its module bounding box from the full bitstream of the module and overwrites the placement location in the full bitstream. The tool that handles the above process is called BitMan and it is presented in Section 5.5.

As a summary to the proposed approaches, bit-level bitstream relocation will provide the desired result in the realm of milliseconds, while the netlist approach will guarantee timing in critical relocations and replications. Time critical relocations (e.g. in high clock rate designs) should occur with the netlist level approach and the bit-level approach would be preferred for functional testing and lower complexity designs. However, both choices can be used in any scenario. The initial bitfile generation occurs with the netlist approach to detect possible timing violations.

5.4 Back-end overview

Our flow can automatically generate multiple physical implementations of the same module to incorporate the heterogeneous resource layout of logic, memory and arithmetic block columns (CLBs, BRAMs and DSPs). This allows for a tighter placing of modules. When all bitfiles and placement positions are generated, we load the kernels in the requested order into the reconfigurable region. This occurs through the C interface code of Maxeler and by sending the partial configuration bitstreams to the ICAP stream. The C code is used to read the bitfiles store them in memory and program the device when needed. The programming sequence for the applications should be orchestrated by the user. This is done by enabling the ICAP stream to write the device until a dedicated function is done. At any point, the reconfigurable region can be
5.4. BACK-END OVERVIEW

Figure 5.9: Two programming phases of a reconfigurable region. Initially the empty reconfigurable region (see Figure 5.3) is programmed with 2 kernels, before resetting the region back to its empty state with a reverting bitstream and loading another individual module. The reverting occurs between the transition from one configuration to the other (not obvious in the figures), with the resetting bitstream.

set back to the initial empty state by loading the reverting bitstream and load another function. An example is depicted in the subfigures of Figure 5.9, where initially the user loads two individual modules, sets back the static system to the original empty region design and then configures a new module. In another case, the user is also able to replicate or relocate existing modules. For that purpose, we save the initially calculated placement positions (in the design phase) in order to incorporate the placement of implementation alternatives. This is automated by BitMan that adds automatically corresponding metadata to the original Xilinx configuration bitstream.

One of the main contributions of the here proposed flow as compared to full static implementations, is that the flow can be parallelized. Figure 5.10 depicts an overview of the back-end process, parallelized amongst the individual entities of the Maxeler project. This is an important characteristic because the mapping and the routing tools need significantly more time with rising complexity. Thus, splitting the design into distinct parts (partial modules and static part) can save substantially CAD tool time when executed in parallel. Moreover, with our flow, every modification of the static system or a kernel will be done independently, without the need of recompiling and re-implementing (including place and route) the whole system. This occurs due to the fact that each part of the dynamic architecture is mapped and placed and routed using the same interface macros, but they are done independently from the other components or static system. In addition, each one of the generated kernels can be used in different
projects, as long as the module fits the region and matches the interface. All the above properties target to improve design productivity, by shortening the design time.

A flow diagram of the back-end process is shown in Figure 5.11. This figure is a concatenation of the subprocesses that constitute the back-end process. In the left hand side of the figure and right sides of the figure, static and module processing are depicted, respectively. On the bottom, a summed up version of the bitfile and partial bitstream generation is depicted. Having introduced the implementation flow, we will now take a closer look into the tools that had been used in the background to hide the low-level FPGA-specific details.

5.5 Customized Tools

The FPGA vendor Xilinx provides tools that allow for implementing run-time reconfigurable systems. However, the vendor flow implements partially reconfigurable modules as an increment to a static system. This means that a module can only work
Figure 5.11: Toolflow diagram depicting the steps that are performed after we extract the RTL code from the Maxeler Compiler to enable moving from a static implementation flow to a dynamic implementation flow. The left half of the figure illustrates the steps occurring while implementing the static part, while the right half focuses on the partial module generation. External tools are marked in blue, while Xilinx tools are marked with grey.
in its particular static system and that any change to the static system requires a reimplemen-
tation of the corresponding modules. In order to provide systems with more flexibility, where modules can be implemented independently to the static system and where modules can be relocated and used across multiple different static systems, we are using a chain of academic tools on top of the vendor Xilinx tools.

5.5.1 GoAhead

*GoAhead* is a tool that is used to create all the components for the system’s reconfigurable design. GoAhead provides floorplanning capabilities, communication infrastructure, and constraints generation that are required for the physical implementation. As referred in the previous sections, GoAhead is used for constraining the partial region as well as for the defining bounding boxes of the modules. It also generates routing and clock blockers that the user can define by constraining routing paths, as well as the RTL description of the reconfigurable region for a design. The tool can be controlled and automated by inputting external parameters from previous processes or tools, such as the Bounding Box Generator.

GoAhead can be controlled by either a GUI or through scripting. In this work, the scripting interface was used in order to generate module bounding boxes or reconfigurable regions, including all VHDL code templates and physical constraints (for controlling the place and route step to comply with the partial reconfiguration rules). The GUI can be used as a visual representation of the design and also allows debugging mechanisms and an illustration of the current selections in the device. The tool can also load and visually represent Xilinx netlists as they are implemented on the chip. An example of the GUI is depicted in Figure 5.12 and the same interface can be used for debugging, floorplanning purposes, and module placement (at netlist level). Furthermore, GoAhead was used in this project to modify netlists by cutting out parts of it or combine/replicate more netlists to create a new design.

Finally, GoAhead supports all recent Xilinx FPGAs, including Virtex 6 (as used in this project), all Virtex 7 series FPGAs (through an XDL API) as well as the new UltraScale devices (though a TCL API). Thus, GoAhead can be used in a flow for different devices using the approaches (i.e. placement and routing constraint generation, clock blockers) used within this PhD for independent reconfigurable module generation that support relocation and replication. Some projects that use the GoAhead tool with the Virtex 7, the UltraScale and the UltraScale+ devices can be found in [107], [108], [7] and [109].
Figure 5.12: GoAhead GUI displaying device primitives and an example of a resource footprint. The resource footprint is a string of symbols, depending on the resource that it occupies. The resources are depicted with a yellow box on the top half of the figure, while an example resource footprint is depicted in the bottom, containing a slice-L, a slice-M, slice-L, a slice-M, slice-L and a BRAM.

5.5.2 Bounding Box Generator

The tool Bounding Box Generator supports the implementation of partial modules by defining rectangular boxes for accelerators, as well as finding possible placement positions for those accelerators. The algorithm of Bounding Box Generator generates bounding boxes for entities, based on an FPGA resource string model of the available resources in the reconfigurable region. This model denotes the device specific primitive allocation to slices, and module primitive requirements.

During the bounding box generation phase, the tool adds more and more resource columns to the module string specification from the FPGA representation until all primitive requirements have been met. With this, the tool ensures to only implement modules for feasible module bounding boxes and that it identifies all possible minimal design alternatives. The tool also takes into consideration multiple clock regions. The generated bounding boxes can span anywhere between one clock region and the entire height of the device (or reconfigurable region). This adds even further flexibility to the placement phase, providing modules with more possible placement positions, for allowing a much tighter overall packing.
Figure 5.13: Example of a reconfigurable region, spanning 3 clock regions, and some module requirements. The region is modeled with the shown alphabet, and a module is presented as a set of primitive requirements.

### 5.5.2.1 Implementation

To explain the operation of the bounding box computation, let us assume that we assign a task to the tool to find placement positions for the example module, with resources as depicted in Figure 5.13, inside the shown reconfigurable region. First, the tool needs to know the number of primitives provided in each column. These values are device specific and must be input into the bounding box generator. In the example region, we have the following number of resources for each column:

- **SliceL**: 40*8 LUTs
- **SliceM**: 40*8 LUTs
- **BRAM**: 20 RAM primitives (10 36 Kbit or 20 18 Kbit)
- **DSP**: 20 DSP primitives

Note that the number reflects the occupied resources in the demonstrated system implemented on Virtex-6. The algorithm (illustrated in Figure 5.14) starts scanning from the first available resource in the reconfigurable region. It checks if this resource contains primitives needed by the module. If so, it adds the slice to the module string and updates the module requirements to reflect that the primitive in the added slice has already been taken into consideration. This step is repeated until all primitive requirements have been met, and the resulting module string represents a design alternative.

In order to give the user as much choice as possible and to allow for fine grained and flexible module placement, Bounding Box Generator looks for all bounding boxes...
spanning from one to as many clock regions as the reconfigurable area has available. This will occur without any input from the user. As such the steps above are repeated using incrementally more clock regions (i.e. increasing the height of the modules). Considering our example, the algorithm looks at the bounding boxes starting with the first resource slice. The tool identifies 3 placement positions as can be seen in Figure 5.15. However, in this project the number of clock regions is defined by the height of the reconfigurable region.

The bounding box generator exploits the fact that the smallest module (i.e. the module variant with the shortest resource string) that fulfills the resource requirements will result in the lowest internal fragmentation. As such, only the smallest design alternatives are considered at the end of this computation. This allows for the reduction of the run-time search space, whilst still providing high placement flexibility. For example, the tool will consider only two of the three module design alternatives for the first position in the reconfigurable region provided:

- (LMBDMB) * 1 row
- (LMB) * 2 rows
- (LMB) * 3 rows (discarded as the two row variant has lower internal fragmentation)

All module bounding boxes generated will be continuous and rectangular. This means that unnecessary resources cannot be skipped. In the example, we can see that the 1 row module generated contains a DSP resource slice even though DSPs were not necessary for the correct run of the module. Similarly the 3 row implementation uses more resources than the 2 row one, even though it only needs just as many. This means that there is a need for a step after the bounding box generation to determine which bounding boxes should be used for physical implementation. Note that during the experiments of this PhD, we used a parallelized 512-bits interface. Thus, each module was constrained to use the full height of the region.

In order to further reduce the search space and still provide the user with flexibility, the algorithm employs a heuristic. Once the total number of design alternatives is computed, the tool sorts the resulting list of bounding boxes in such a way that the alternatives with the most possible placement positions are at the start. Typically, a relatively small number of alternatives is sufficient to allow placement with little external fragmentation (i.e. unused resources between placed modules). This heuristic increases the chance that run-time placement results in better resource utilization.
CHAPTER 5. LOW-LEVEL ASPECTS

Figure 5.14: Algorithm applied to each module specification. Transition 1 detects the number of regions. Note that the algorithm supports different number of clock region height, however in our case, the module height is defined by the reconfigurable region (height of 2 clock regions). Transition 2 happens at every step until all resource requirements for the module are met. Transition 3 occurs once the bounding boxes starting at the current start position were found. Transition 4 allows for mitigation strategies to be employed if routing fails. This is the major contribution of this PhD thesis in [6], alongside a case study for the tool.

One problem that can occur when creating bounding boxes is that, if they are defined aggressively small, there might not be enough resources left over for routing. Because the resulted bounding boxes are rectangles and because the tool uses resource columns as our placement atoms, the bounding box will likely leave some resources
Figure 5.15: Bounding boxes for the defined module in the reconfigurable region, starting only at the first resource slice. Same with the example in Figure 5.13, the tool has detected 3 possible placement positions. Those positions for the module can be applied by the used as design alternatives, should they are needed.

As an extra precaution, the tool contains a method by which the module string can be updated to contain more resource columns as needed. Since routing requires the switch matrix only within a column, extra resources (which is referred by slack variable) can be seen as a wild card (meaning any resource type can be used to ensure routing). This can be added before a placement method is applied by the tool. Finally, if timing still is not met, the algorithm also allow for a “fail” message to be fed back to the generator in order to further increase the number of resources assigned for a module (i.e. one resource column will in most cases solve the routing congestion problem, at the expense of larger modules).

Furthermore, the toolflow implements mitigation strategies that apply physical constraints that will be tried out to improve routability and performance (i.e. the clock frequency achieved). This includes not using the primitives in the corners of module bounding boxes, in order to obtain a better ratio of routing resources to primitives only at places of high possible congestion. This strategy mitigates the same bounding box constraints and the effect is more obvious in the corners. However, all those mitigation strategies are applied automatically as needed without any human intervention, in case of failure to route.
5.5.2.2 Mitigation Strategies

As an additional function, our physical implementation flow is able to handle failures during mapping, placing or routing. For example, if a module does not get mapped, we will extend the bounding box left or right by an additional column. In the case of a routing failure, the tool relaxes the routing inside the bounding box, using three different strategies, that are tried out in the following consecutive order.

- **Strategy 1:** The tool can block the placement in the corners of the bounding box, because the design tends to be heavily congested in the corners. Consequently, by leaving primitives unused but by taking advantage of their routing resources (switch matrices), we provide locally a higher ratio of routing resources to primitives (e.g., LUTs, DSPs, or BRAMs).

- **Strategy 2:** Leaving the top and bottom rows unmapped, as shown in Figure 5.16. Figure 5.16 left shows the fully routed module, while on the right side of Figure 5.16, the unmapped CLBs on the top row are depicted, inside the gray box.

- **Strategy 3:** The last, most effective but also expensive strategy, only uses the
5.5. CUSTOMIZED TOOLS

Figure 5.17: Mitigation Strategy 3: Unused primitives and used switch matrices frame around the module, to relax routing if necessary. The primitives on the sides of those kernels are completely unused and some routing is occupied to relax the internal wiring.

frame around the module in which only the routing resources will be used. Figure 5.17 depicts this situation. In this case, the mapping will be done entirely in the inner side of the frame, while we will use only the routing resources of the frame. In the left side of Figure 5.17, the full partial module is presented, while on the right, details of the implemented partial module are shown.

Implementing modules in bounding boxes includes more constraints on the physical implementation. However, in some cases, routing is not possible for a module with the given constraints. To circumvent this issue, BBG allows hiding the low-level details from the user, while offering mitigation strategies to solve routing issues. In our toolflow, this tool automatically enables a flow usable by application domain experts even for difficult implementation problems.

As a summary, Bounding Box Generator addresses the challenge of automatically defining placement positions in a predefined region, without the need of interaction by the user. The tool works directly with the GoAhead tool to generate the required constraints. This interaction allows the here presented mitigation strategies to be performed transparently to a user (i.e. a Maxeler application expert). This and this case
study on generated accelerators on a Maxeler platform is the author’s contribution to the the Bounding Box Generator tool.

5.5.3 BitMan

The tool BitMan is a component used in the back-end processing. This tool can modify different designs directly at the bitstream level. BitMan supports functions that include module placement, replication, and relocation. The tool also offers an API to place, relocate, and replicate modules directly when running applications on a Max3 system. BitMan supports Virtex-6, all 7-Series, and Ultrascale FPGAs from the vendor Xilinx.

In this PhD project, the tool is used only in the final step of our toolflow in the bit-level approach. The tool can be used, as shown Figure 5.8, to extract the partial bitstream of a fully placed module. In addition to the offline use, the tool can relocate and replicate modules at run-time if necessary. As described previously, the tool can use the placement positions returned by BBG to relocate modules. The module positions match the resource footprint such that the footprint of the reconfigurable module and the footprint inside the reconfigurable region are the same at the placement position. Should a different box is needed by the tool, then BitMan cannot relocate the module, and it is mandatory to run the back-end process again, with a different bounding the box as generated by BBG. However, the benefit now is that the user will need to start from the Xilinx mapping phase until the partial bitfile generation, and not run the fully toolflow from the beginning. it is important to highlight that this is commonly all performed at system design time to build an accelerator library. When using the system for acceleration, BitMan is only needed to stitch modules together to form pipelines.
Chapter 6

Results

The previous chapters presented an implementation flow from HLS to the resulting static and partial bitfiles. This chapter presents an evaluation of the toolflow and its benefits compared to a full static approach. Those benefits correspond to design time and resource usage benefits of the presented dynamic toolflow in Maxeler. Those benefits come on top of the already existing flexibility and abstraction that are enabled by our implementation flow, due to the enabled design choices offered by this approach.

The chapter starts by discussing the experimental setup used in this thesis. In addition, this chapter presents two case studies. The first case study corresponds to a dynamic image/video processing application, generated through the Maxeler MaxJ programming model. The second case study showcases a stream processing architecture for SQL query execution on FPGAs. The latter application was implemented through our toolflow, using RTL level accelerators.

6.1 Experimental setup

For the experiments, a Max3 Workstation was used which provides a large Xilinx Virtex-6 XC6VSX475T FPGA, which is connected to the host computer via PCIe. The FPGA is surrounded by 24 GB (upgradable to 48 GB) of DDR-3 memory and the host CPU is an Intel(R) Core(TM) i7-2600S clocked at 2.80 GHz. The most recent system from Maxeler is Max5, which uses a Xilinx UltraScale+ device. The Max5 platforms can also support the newest Xilinx Alveo boards [110]. As the Maxeler’s toolflow has not changed significantly to the newest Max5 platform and all the external tools that we use have been tested to work with all the Xilinx devices, it should be possible to migrate the here presented results to Max5 with the availability of the
corresponding hardware platform. Of course, this would require modifications on the way that GoAhead interprets the Vivado toolchain compared to the ISE.

The here proposed automatic flow starts with generating the RTL code from the Maxeler compiler and modifying it as described in Chapter 4. As we cannot interfere with the Maxeler Compiler during the translation from MaxJ to RTL, we decided to introduce the MaxJ language extension to split the dynamic application in distinct parts. Additionally, the flow modifies the generated RTL code after the code is generated by the Maxeler compiler. Those two processing phases require insignificant amount of time to occur, as the required time is in the realm of milliseconds while the vendor CAD tools take several minutes (depending on the size of the application) to generate the RTL code.

6.2 Image processing application

A video/image stream processing application was implemented and presented in [111], where various video stream processing modules can be arbitrarily chained to form more complex acceleration pipelines. This application is a classic example of mutually exclusive sub-functions that can also be used in several combinations, in order to present the benefits of our flow. A video of the flow and the system in action is available on [112]. The video illustrates different example cases and various filter chaining examples of multiple independent accelerators. The video also presents the results after each reconfiguration, starting from an empty reconfigurable region or loopback routing and changing to different configuration scenarios.

The implemented module library consists of 8 image processing functions. Those are Brightness correction, Sobel edge detection, RGB-to-Greyscale, Skin Color Detection, Gaussian blur, a Mean, a Minimum and a Maximum value filter. In addition, we have implemented 3 PRGroups, containing a Mean with an RGB filter, Brightness filter with a Skin detection filter, and a Min and Max filter. All of those functions are generated entirely by the Maxeler compiler from MaxJ code. Figure 6.1 and Figure 6.2 depict FPGA-editor screenshots showing implemented unique kernels and implemented PRGroups. It can be seen, as in Figure 5.5, that the interfaces on both sides of each module use the same equivalent wires, as previously depicted in Figure 5.5, which is the key property for module chaining.
6.2. IMAGE PROCESSING APPLICATION

Figure 6.1: Fully implemented PRGroups, as shown in Xilinx’s FPGA editor tool. The illustrated PRGroups are (l.t.r.) Mean with an RGB filter, Brightness filter with a Skin detection filter, and a Min and Max filter.

6.2.1 Resource Utilization

The resource usage of the generated PRGroups and unique kernels is shown in Table 6.1. The overhead that occurred due to the modifications, which are described in Section 4.1 is included in those numbers. The additional implementation cost of each accelerator, compared to the initial version generated by the MaxCompiler, is about 441 LUTs (used for FIFOs which are implemented using distributed memory) for each kernel, as a 512-bit wide and 32 positions deep FIFO is added. Those FIFOs are generated by the Xilinx ISE Memory Generator tool. Note that in Table 6.1, the modules do not contain DSP primitives. However, the here proposed flow fully supports accelerators containing DSP blocks.

The static system consists of 9375 LUTs, 22 BRAMs and has one input and one output of 512 bits to the image operator’s I/O and one input for the ICAP configuration port. In the implemented example, the reconfigurable region offers 28800 CLBs, 128 BRAMs, and 112 DSPs and it is placed in the top right corner of the chip. This placement was chosen because in the static implementation the connections are placed in the center of the device next to the PCIe interface.

As a reference, we implemented a non-reconfigurable full static (FS) design that provides all the aforementioned kernels in parallel. In this system, the input could
be streamed through all possible kernels and back to the host machine, if needed. However, the static system with the largest accelerator (Sobel ED) will require a total of 35857 LUTs. Compared to the example using a full static implementation in Table 6.1, the dynamic system requires only a maximum of 38% of its resources.

As mentioned before for a general case, an alternative approach to this example FS for video processing, a user could implement multiple different projects and reprogram the device with the project that contains only the filters needed which, firstly takes time and secondly, in case of switching accelerators in most real-world streaming applications (e.g. a video processing platform), the operation of the system would be interrupted for a significant amount of time (i.e. at least tens of seconds required to reprogram the device). Additionally, there are cases where a user may need to run a combination of kernels in a pipelined fashion. However, as the number of possible kernel choices grows, it is not feasible to create a design for every combination of kernels that can be combined. Moreover, some filters could be mutually exclusive to each other (e.g. Mean filter versus Gaussian filter), which implies that hosting them both in a static solution would result in an underutilized FPGA implementation.

Additionally, our framework offers the freedom to load any kernel or any number of the available kernels, as long as they fit into the reconfigurable region, and allows loading and resetting ((2 full clock regions of XC6VSX475T)) at a maximum of 10
Table 6.1: Resource utilization of generated PRGroups and unique kernels. The added FIFOs introduced in Section 3 are included in those number.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>LUTs</th>
<th>BRAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brightness correction</td>
<td>4444</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RGB to Grayscale</td>
<td>6814</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Sobel ED filter</td>
<td>26482</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>Gaussian blur filter</td>
<td>12659</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>Mean filter</td>
<td>12678</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Skin Color Detection</td>
<td>9821</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Minimum</td>
<td>4582</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Maximum</td>
<td>4556</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Mean + RGB</td>
<td>19292</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>Brightness + Skin</td>
<td>14358</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Min + Max</td>
<td>9138</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Example full static</td>
<td>92405</td>
<td>112</td>
<td>0</td>
</tr>
</tbody>
</table>

ms for each subsequent modification. This is possible while keeping memory active during reconfiguration and idling only the reconfigurable region during the process. The reconfigurable region used in our case study provides 10% of the FPGA resources and much more resources can be allocated for the reconfigurable region, if needed. Please note that a user can decide to only make parts of the run-time reconfigurable system, while leaving other accelerator kernels static (e.g., if those kernels are being used constantly for preprocessing).

### 6.2.2 Compilation and Configuration Time

The Maxeler Compiler translates MaxJ to VHDL and works relatively fast. For example, the tool provides the RTL code description for all the example cases, including the full static implementation (FS) mentioned in the previous subsection in 10 to 20 minutes. After that step, the FPGA vendor tools carry out the entire physical implementation until the final configuration bitstream. The results on the compilation time of our approach are shown in Table 6.2. The time metrics include the time needed for RTL generation from Maxeler, BBG, GoAhead and the whole toolflow from RTL synthesis to bitfile generation. Thus, the tools have to implement 11 individual (unique kernels plus PRGroups) and significantly smaller designs and another one for the static part. All 6 distinct parts (i.e. static system and 5 PRGroups of which 3 PRGroups contain 2 kernels each and 2 PRGroups contain 1 kernel each) together contain about the
same logic, as compared with the FS design, if they are combined. On the contrary, the FS design needs 74 minutes from the RTL generation to the final MaxFile, or more than 25% additional time in CAD tools for this experiment. However, the larger the full static design, the larger the CAD time that is going to be saved. Please note that the compilation time for the partial modules and the static system includes a maximum cost of 10 seconds, needed for the VHDL code processing and project generation.

The reconfiguration time of all our modules is listed in Table 6.2. As listed, each kernel takes from 3 to 6 ms for configuration by the ICAP instance in the device. After the programming is done, we can execute the function loaded by sending the image data from the DDR memory or PCIe. This execution step remains the same as in the static Maxeler approach. Each reconfigurable region can be resetted individually by generating a resetting bitstream for each region and loading the file through the interface C-code of the Maxeler project. This will ultimately send partial configuration bitstream to the ICAP configuration port of the FPGA.

Finally, module relocation at run-time is done by the BitMan tool. BitMan can generate relocated partial and full bitstreams arbitrarily just by providing the full bitstream of the static system, the partial bitstream that we want to relocate and the new placement position. Using BitMan only requires running a single command containing the aforementioned files and physical locations. The relocation positions will be provided by the BBG tool, during the implementation phase. For the stitching of the aforementioned kernels, BitMan needs a maximum of 10 seconds to generate the final relocated bitstream for each possible kernel. In many practical systems, this process could be carried out once and stored for future configuration processes. The benefit of fast relocation and replication can provide an out-of-the-box solution to improve flexibility and productivity, by providing a more dynamic aspect and on-the-spot changes or extensions to a design.

The current system is clocked at 100 MHz, which is the default clock frequency set by Maxeler. With this, the dynamic implementations provide a peak performance of 6.4 GB/s. The clock frequency was not pushed higher for this experiment in order to directly compare with the existing work in [103] in terms of throughput. This performance can be achieved by all pixel operators implemented for our example case. Similarly, the static system without any accelerator provides the same throughput. The system does include a minor penalty in latency for the single kernel designs, as the reconfigurable region itself includes a pipeline latency of a maximum of 5 clock cycles.
6.3. ADDITIONAL FLEXIBILITY

Table 6.2: Time needed for reconfigurable module generation, the static part of our reconfigurable module and the full static design. Compilation time does not include the RTL generation time as it generates the same VHDL code. The FS design includes only the Maxeler compilation time, as it is full static. The total modification and implementation time for a dynamic project is the maximum compilation time (i.e. 74 minutes).

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Compilation Time (min)</th>
<th>Reconfig. Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brightness correction</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>RGB to Greyscale</td>
<td>27</td>
<td>5</td>
</tr>
<tr>
<td>Mean filter</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>Gaussian blur filter</td>
<td>26</td>
<td>5</td>
</tr>
<tr>
<td>Sobel ED filter</td>
<td>43</td>
<td>5</td>
</tr>
<tr>
<td>Skin Color Detection</td>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>Minimum</td>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>Maximum</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>Mean + RGB</td>
<td>39</td>
<td>4</td>
</tr>
<tr>
<td>Brightness + Skin</td>
<td>38</td>
<td>4</td>
</tr>
<tr>
<td>Min + Max</td>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>Static Part</td>
<td>56</td>
<td>NA</td>
</tr>
<tr>
<td>Resetting bitstream</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Example full static (FS)</td>
<td>74</td>
<td>NA</td>
</tr>
</tbody>
</table>

For even higher throughput, multiple reconfigurable regions may be used in parallel (if permitted by the application and the existing inputs to the dynamic system).

6.3 Additional flexibility

The unique functionality of our flow in Maxeler platforms, this section proves the addition of flexibility of PR in the Maxeler Max3 platforms, in terms of accelerator configurations, testing and design choices. As discussed in Chapter 3, the new additions on the software side of the implementation allows designers to create multiple static kernels but also dynamic PRGroups and unique kernels (through our front and mid-level approach). This means that in the image processing example, there exist 8 accelerators and 3 PRGroups in Table 6.2. Every possible extension of this design can occur through parallelizing or relocating the generated accelerators or even combining kernels not in PRGroups.
Existing kernel combinations could provide testing of unforeseen functions, without going through the implementations phases which are considered time consuming. This benefit also applies to kernels and PRGroups that are used in different project and can be introduced as additional functions to project that match the requirements of region size and interface. Skipping implementation phases for accelerator testing could prove vital for application domain experts, who are certainly not used to extensive implementation times when using applications on CPUs or GPUs.

In addition, allowing instantiation of reconfigurable regions directly from the HLS side is important, as the designer can define the number of regions based on the corresponding project. Those regions can be used by the same streams as an extended pipeline or, depending on the size of the project, even by different stream for different parallel computations. Thus, a designer can provide a description of reconfigurable and static pipelines in the same project based on the project requirements. All this is done as described in Chapter 3. The region input width can be configured automatically by our PR code generator introduced in Chapter 4, based on the architecture described in the Maxeler PRManager. This can be proved by the similar regions used for our different application case studies in Section 6.2 and Section 6.4.

6.4 Scalable Filtering Modules for Database Acceleration

This test case implements a dynamic stream processing architecture for SQL query execution on FGPAs. This is achieved by building pipelines based on scalable database accelerator primitives and partial reconfiguration. The modules used for this case study are provided from project [7] as RTL code. This project took three modules to automatically implement a library of reconfigurable and stitchable accelerator kernels. In addition, a surrounding static Maxeler system was developed to provide an infrastructure with PCIe and extend DDR memory.

To achieve this, we synthesized and implemented the design targeting the existing Maxeler Max3 containing a Virtex-6 FPGA, using the here presented flow from Maxeler’s Custom HDL wrapper. This design is a scalable filtering module for SQL query acceleration and its functionality is described in Section 6.4.1. This section also focuses on the architecture and the design decisions taken during implementation. In order to assess the system aggressively we implemented a case study for restriction module for query Q19 from TPC-H benchmark [113], which is the query with the
### 6.4 Scalable Filtering Modules for Database Acceleration

**Figure 6.3:** Architecture of the provided module for N 32-bit data elements and K DNF clauses from [7].

**most complex restriction and Boolean evaluation in the benchmark.** In Section 6.4.2, this test case provides a proof that the here presented flow can support not only HLS generated code, but also the existing Maxeler extension that supports CustomHDL. The very same accelerators were implemented in UltraScale+ devices in [7] as well, however that test case is out of the scope of this PhD thesis, as it could not be tested in the Max3 run-time system implementation (Max3 uses a Virtex 6 device).

#### 6.4.1 Filtering module implementation

A stream processing module that applies restriction would need to compare or match the data elements to a set of pre-initialized reference values. Arithmetic compare requires the support of six main operations: \( c \in \{<, \leq, =, \neq, \geq, >\} \). Previous works put forward modules that each has a single hardwired operation to execute. Although this suits well a low throughput system as proposed in [114], it would not be practical for high throughput, as we aim for larger datapath sizes and provide optional computation on every 32-bit data element in a record. Therefore the provided filter modules included two hardwired compares (see Figure 6.3(b)) and use their results to evaluate a pre-initialized selection of compare operations.

Furthermore for the implementation of the case study, we propose the use of distributed memory (DRAM) to hold the reference values as well as the operations to be performed for each data element in each functional ID position (see Figure 6.3(a)). Additionally, the modules can implement multiple compare operations for every data element by replicating the compare elements in Figure 6.3(d).
The compare operations produce a true or false response, but in the cases of complex WHERE expressions using multiple compares, we have to evaluate multiple true/false responses through a boolean expression. Different methods for boolean evaluation in restriction operations have been proposed, such as using look-up-tables [115], hardwired boolean operator trees with programmable nodes [116], or providing dedicated modules for the boolean operators [114]. These approaches are not designed to deal with many input literals (which are common for complex boolean expressions) and/or literals that are evaluated over multiple clock cycles (as needed for larger records). For example, using big LUTs for evaluation is limited as it requires \(2^N\) bits to store the look-up table and a method for splitting the problem into multiple smaller LUTs has not been discussed in [115]. A boolean programmable tree of operations limits the flexibility of the enabled queries to be accelerated, as it is unable to process boolean results that are generated in different clock cycles (from data in different chunks of a record) [116].

To extend support for larger data types than 64 bits, the corresponding data types were split into multiple 64-bit data types before the generation of DNF clauses and incorporating the relation of the sub-types into the logical expression. For example, 128-bit compare \(X < Y\) can be implemented by using the 64-bit higher(H) and lower(L) parts of the values similar to the support of 64-bit values. Strings can be of very large sizes, but most operations would be comparing for an exact match, thus utilizing a == operator, which implements a logical AND of the compared subparts. Consequently AND boolean operations result in a single DNF clause, and hence does not increase the complexity of the DNF logical evaluation.

As a solution to this problem, the filter modules use a design that adopts DNF [117]. DNF comprises of clauses that are aggregated with an OR operation where each clause consists of AND-ed positive or negative propositional variables that result from the undertaken compare and match operations.

A variable in a clause has three possible programmable states for each of the clauses: positive variable, negative variable, not existing in this clause. Thus, this requires parameter storage of at least two bits per functional ID to encode the programmability of the variable in a clause and then logic to evaluate the result from a compare PE. We propose the use of a single LUT-M as a look-up table with inputs the function identifier (5 bits) and the resulting bit from the compare PE, and producing a 1-bit result that states whether the particular variable satisfies the particular clause as shown in Figure 6.3(c). These variables are then evaluated using static AND and OR
6.4. SCALABLE FILTERING MODULES FOR DATABASE ACCELERATION

Figure 6.4: Fully implemented filter modules targeting a Xilinx Virtex-6 device with capacity of 8 a, 16 b, or 32 c DNF clauses (see Table 6.3).

trees as shown on Figure 6.3(c-d). When the variable does not exist in the particular DNF clause, it is initialized to produce a boolean result of 1 for both result states from the compare. With these optimizations, we shrink the logic utilization of the DNF structure by requiring only 1 LUTM per variable and an optimized hardwired boolean reduction.

In addition to the image processing pipeline case study in Section 6.2, this filter case study requires module replication (e.g. for complex Where clauses) and an implementation of the filter modules. This, for example, allows setting the exact compare function from the C interface (see Section 3.1) of the application in the Maxeler I/O infrastructure. Thus, the filter accelerators are instantiated in a Maxeler and followed the same flow for physical implementation, similarly to the one described in Chapter 5. This project is used to verify that the flow presented in this PhD thesis functions with Custom HDL accelerators, as well as with MaxJ generated kernels (or PRGroups).
6.4.2 Implementation Results

We implemented a generic version of the proposed filter module including parameters, compare PEs per data element (J), and the number of DNF clauses (K). The module applies restriction and boolean evaluation by modifying the stream state bits. In this implementation, a 4-bit value can indicate reserved commands (e.g., for module initialization with reference values). Since DNF implements in a OR reduction, only one clause is required to evaluate a record. By providing modules for a different number of clauses, partial reconfiguration can be used to configure the cheapest solution on the FPGA. Moreover, more complex problems evaluating even more clauses can be implemented by daisy-chaining of multiple smaller modules.

The case study on the Maxeler PR system operates on a 512-bit datapath, as in the test case presented in Section 6.2. We reserved a 32-bit word to propagate stream and chunk enumeration, leaving the effective datapath to 480 bits. We synthesized, placed and routed three versions of the module as shown in Table 6.3 and Figure 6.4. Three configurations have been evaluated. As shown in and Table 6.3, the three configurations implement 1, 2 and 4 compare PEs per data element and respectively 8 in Figure 6.4(a), 16 in Figure 6.4(b) and 32 Figure 6.4(c) DNF compare clauses capacity. Thus in Table 6.3 we report in brackets how many of the columns are logic resources. This case study uses Maxeler’s default frequency. Thus the targeted frequency is 100MHz, however the tools report higher frequencies achieved.

The utilization for the PEs in the proposed design is deterministic. The highest relative utilization is for distributed memory, which is used to hold the initializing data for the compares and the DNF variables boolean evaluation. For a module implementing $N$ 32-bit data elements, $J$ compare PEs per data element and handling $K$ DNF clauses, the LUT-M utilization can be calculated as:

$$LUTM = \text{InfraOverhead} + N + N \times J \times (16 + 2 + K)$$

where the infrastructure overhead is a constant that depends on the implementation and target device. We observe that the module utilization scales linearly with datapath width.

For the implementation, the frequency for the Max3 FPGA to 100 MHz and the result after place and route was up to 245 MHz. This leads to an effective throughput of between 12.5 and 14.7 GB/s. This throughput is sufficient to utilize the fastest available
The more clauses executed per module the better the speedup over a software only solution. All six implemented variants (see Table 6.3) can be placed alone to execute the restriction and boolean evaluation operations in all queries from the TPC-H for benchmark except Q19 [113]. Q19 is the most complex in terms of compare operations and boolean evaluation. It implements 28 different compare operations, including a text field that is compared to 12 reference values. A single operation per module approach requires 28 modules to implement the compare operations of this query and 27 2:1 boolean modules for evaluating the Boolean expression.

The Boolean expression of Q19 requires to compute to 24 DNF clauses. Our modules are flexible and can be placed in a daisy-chained fashion to solve larger DNF problems. A single module can implement 1, 2 or 4 compares for a particular field and evaluate 8, 16 or 32 clauses from a DNF. To implement a large DNF expression, we can place multiple modules in the PR region. Each module implements a certain amount of clauses (8/16/32) from the total clauses needed. Thus, DNF-clause-wise, we

<table>
<thead>
<tr>
<th>Datapath width</th>
<th>Virtex6</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP PEs</td>
<td>1(15)</td>
</tr>
<tr>
<td></td>
<td>2(30)</td>
</tr>
<tr>
<td></td>
<td>4(60)</td>
</tr>
<tr>
<td>DNF Clauses</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Bitstream (Fig. 6.4)</td>
<td>(a)</td>
</tr>
<tr>
<td></td>
<td>(b)</td>
</tr>
<tr>
<td></td>
<td>(c)</td>
</tr>
<tr>
<td>LUT-Ms</td>
<td>1236</td>
</tr>
<tr>
<td></td>
<td>1896</td>
</tr>
<tr>
<td></td>
<td>3936</td>
</tr>
<tr>
<td>LUTs</td>
<td>3624</td>
</tr>
<tr>
<td></td>
<td>5051</td>
</tr>
<tr>
<td></td>
<td>8320</td>
</tr>
<tr>
<td>Flip-flops</td>
<td>2793</td>
</tr>
<tr>
<td></td>
<td>3020</td>
</tr>
<tr>
<td></td>
<td>3474</td>
</tr>
<tr>
<td>Resource Columns</td>
<td>4(3)</td>
</tr>
<tr>
<td></td>
<td>5(4)</td>
</tr>
<tr>
<td></td>
<td>9(7)</td>
</tr>
<tr>
<td>Freq Target, MHz</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Freq Report, MHz</td>
<td>245</td>
</tr>
<tr>
<td></td>
<td>205</td>
</tr>
<tr>
<td></td>
<td>210</td>
</tr>
<tr>
<td>Throughput, GB/s</td>
<td>14.7</td>
</tr>
<tr>
<td></td>
<td>12.3</td>
</tr>
<tr>
<td></td>
<td>12.5</td>
</tr>
</tbody>
</table>

Table 6.3: Utilization and throughput of the evaluated module configurations. Note resource columns in brackets represent only logic columns.

simple channel DDR memory speed. Targeting higher throughput, the proposed design can be implemented on a wider datapath and additional pipeline stages can be inserted.
can implement Q19 by placing at least one 32-clause module or two 16-clause modules etc. Since Q19 requires 12 compares for one field, we can replicate the module in the PR region or replicate the field in the stream, or any combination of both. For example, in the Maxeler case study we use the module from Figure 6.4(c) and replicate the challenging field three times.

Our system can perform the restrictions in all the TPC-H benchmark, including query 19. We use the implementation with 32 DNF clauses (Table 6.3(c) and Figure 6.4(c)) and run the restriction and boolean evaluation from TPC-H Q19. For this, TPC-H database was generated with a scale factor of 1, which results in a 1 GB database. Also, the field P_CONTAINER was replicated in order to accommodate for its large amount of compares with different reference values. The experimental run was performed from DDR memory through the FPGA PR region and results were streamed back in DDR memory. The runtime reconfiguration for the module took from 5 to 10 ms. The initialization of the module with the compare and clause parameters took less than 1 microsecond. All restriction and boolean evaluations from the query are executed by a single module with 480-bit effective datapath running with the default frequency of the Maxeler system, which is 100 MHz, resulting in a throughput of 6 GB/s through the PR region. The DDR-FPGA-DDR run of the restriction takes 121 milliseconds to execute, which corresponds to filtering at a rate of 50 million records per second.
Chapter 7

Conclusions

Following the rise of HLS tools, this PhD offers support for partial reconfiguration that exists on the Maxeler platforms, we proposed a dynamic Maxeler implementation toolflow, covering distinct features that comply with an HLS paradigm. This toolflow is fully automated and without any user intervention, it compiles an extended version of the MaxJ code in a dynamic design. The RTL code is modified directly from the generated from the MaxCompiler, before implementation. During the physical implementation phase, the here presented toolflow hides all low-level details and allows design choices for application domain experts, in order to generate an independent static system and independent dynamically reconfigurable accelerators.

This toolflow demonstrated implementation time benefits that reached 25% of the time to bitfile compared to a static implementation. In addition, a minimum of 62% of the resources can be saved as shown in Section 6.2, compared to a static implementation of all the accelerators in a single design. Reconfiguration occurs in the realm of ones of ms (in more extensive modules, it can reach tens), while the clock frequency is not impacted and there is a maximum latency of 5 clock cycles. The here presented project as also compared with all the existing works enabling partial reconfiguration on the Maxeler platforms. A total of four publications were resulted from this PhD project.

This chapter presents the contributions of this thesis in Section 7.1. Finally, Section 7.2 presents possible extensions for this work.
7.1 Discussion and Contributions

This PhD thesis proposed a complete framework around the Maxeler HLS compilation flow that extends the current full static to a dynamic dataflow approach. This extended compiler functionality enables partial reconfiguration for Maxeler developers, who can describe the dynamic system directly in a high-level language (i.e. MaxJ). After the programming of the application, the FPGA implementation can be implemented from the HLS application all the way to the configuration bitstreams without any intervention from the application-domain expert.

The work proposes a top-down approach, starting from a language extension that models the dynamic aspects of a system. The automatic compilation processes the Maxeler’s generated code and automatically performs the changes needed to implement a reconfigurable system. The modified RTL code is initially split in accelerators and the static system that correspond to the subparts of the dynamic design. Those subparts will be physically implemented in parallel to generate a full bitfile for a static system and partial bitfiles for the accelerators. The entire physical implementation is automated and the flow incorporates mitigation strategies in the case physical implementation fails.

The default clock speed of a Maxeler system is 100 MHz, without any optimizations through the HLS language. On the other hand, latency is only impacted by a maximum of 5 clock cycles, due to the internal pipelining within the reconfigurable region. All implemented accelerators met the default clock frequency of 100 MHz, as defined by Maxeler, at a 512-bit wide datapath. This allowed dynamic stream processing at 6.4 GB/sec with only a respectable increase in latency of 5 clock cycles due to the internal pipeline stages in the region. However, the pipeline stages can be overwritten and thus, the core functionality may not be impacted by latency, should one or more accelerators are loaded in the device (or an accelerator that fully utilizes the region).

The reconfiguration time when using partial reconfiguration is significantly lower than the time to reprogram the device with a different full bitstream. The time for reconfiguration, including internal latency of the Maxeler system during initialization, can occur in less than 10 ms for the full reconfigurable region configuration. The largest implemented accelerators took 5 ms to be reconfigured in the static system. Moreover in our case study, the static application occupies significantly higher resources compared to the dynamic design, that requires only 38% (or save 62% of the
utilization) of the static design’s logic, without losing any of the existing functionality (i.e. the accelerators that can be loaded in the static system).

The toolflow covers general PR challenges, such as placement of a module and creation of a partial region. Module bounding and the search for possible positions occurs through Bounding Box Generator, which is introduced in the context of this work. In addition, other academic tools were used by our toolflow, such as GoAhead, to automate the generation of placement constraints and RTL files for the reconfigurable region, and BitMan to relocate and replicate accelerators in the form of partial modules within a reconfigurable region. Those tools enable flexibility of the design by allowing design choices from software (see Section 3.4) and by allowing multiple placement positions for each of the possible bounding boxes for reconfigurable accelerators. In addition, changes in a subpart of the design will affect only the subpart itself (either the static part or the modified accelerator).

Furthermore, compared to the Xilinx vendor PR tools, the proposed flow allows for multiple kernels/functions to be placed in the same reconfigurable region. This complies with the stream processing paradigm, in which a user can create a processing pipeline of smaller accelerators to implement a more complex application. This concept is usable by users with a software background, as it essentially reassembles the concept of pipeline into hardware. The proposed flow guarantees independency not only within the same project, but also across multiple projects, given that we enable reusability of the generated modules through a common physical interface. Matching this interface and fitting the reconfigurable region is the only requirement of an accelerator to be used in a different project or in a different reconfigurable region in the same project. This matching, however, comes automatically when using the streaming protocol by Maxeler. The independence that allows implementing the kernels and the surrounding static system is in particular useful for debugging as small changes can be accommodated in a fraction of the time.

The here presented flow can be used with other applications that can benefit from our dynamic HLS approach. Every application that contains run-time mutually exclusive applications can potentially be implemented and optimized using our approach. If partial reconfiguration is used, mutually exclusive applications can take full advantage of the resources available in a reconfigurable region. In addition, applications that cannot fit in one device can also benefit from partial reconfiguration. For example, instead of cascading an application across different FPGAs using Maxeler’s MaxRing protocol, our approach enables a time-multiplexing of a single FPGA. In this scenario,
an application that needs to be split into different FPGA partitions could be executed on one physical FPGA using our dynamic approach for time multiplexing (e.g. by using external RAM to buffer data between partitions). In an I/O bound application, this could be even faster as the memory throughput is about an order of magnitude higher than what is available for PCIe or MaxRing. In summary, this PhD project empowers application domain experts familiar with the Maxeler framework for the very first time to benefit from a partially reconfigurable approach. This permits exploiting the dynamic programmability of FPGAs to improve performance, user experience and cost, depending on the application implemented.

7.2 Future work

This PhD project demonstrated that building a partially reconfigurable system could be performed by domain experts using only MaxJ and C code. The here proposed flow implemented on a Xilinx Virtex 6 device (as used in Maxeler Max3 systems). The most recent compiler release from Maxeler is Max5 based on Xilinx UltraScale+ devices. Thus, due to the lack of the latest workstation and FPGA, implementation and experiments had not been performed in this thesis.

Maxeler’s toolflow has not changed significantly from Max3 until the latest Max5 platform. Thus, given that all the external tools used in this PhD project (GoAhead, BBG and BitMan) have been tested to work with UltraScale+ devices, porting the here presented approach should be straightforward. However, this would require new floorplanning of the static FPGA design and of the reconfigurable region. This process is needed once for the new Max5 platform and it is the only time an FPGA expert is needed.

Finally, changes of the automation mechanisms (front and possibly mid-end processing) will be required for a new version of the Maxeler system. However the existing automation of generating dynamic projects from Maxeler, which is created in the time of this work, could be slightly modified on the new platform to create a similar flow. The remainder of our approach, presented in this thesis, could be inherited by the dynamic flow for a Max5 platform. This is planned as future work, assuming the availability of Max5 hardware. It can also be considered to generalize this flow from MaxJ to general HLS languages to support the FPGA cloud computing community.
Bibliography


[56] Intel SDK. https://www.intel.co.uk/content/www/uk/en/software/programmable/sdk-for-opencl/overview.html.


