Management and Analysis of Bitstreams Generators for Xilinx FPGAs

BY

DAVIDE CANDILORO
M.Sc. Computer Engineering, Politecnico di Milano, Milan, Italy, 2008

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2008

Chicago, Illinois

TABLE OF CONTENTS

CHAPTER **PAGE**

1 FPGA TECHNOLOGY AND DYNAMIC RECONFIGURATION 1
- 1.1 FPGA overview 1
- 1.2 FPGA architecture 4
- 1.2.1 Logic blocks 6
- 1.2.2 IO blocks 7
- 1.2.3 Communication resources 7
- 1.2.4 Additional resources 11
- 1.3 The configuration bitstream 12
- 1.3.1 Overall bitstream structure 13
- 1.3.2 Packet headers 15
- 1.3.3 Configuration registers 16
- 1.3.4 Frame indexing 22
- 1.4 Reconfigurable computing 23
- 1.5 Dynamic reconfiguration 25
- 1.5.1 Dynamic reconfiguration schemes 26
- 1.6 Partial dynamic reconfiguration issues 29
- 1.7 Design flows for Partially Dynamic Reconfigurable Architectures 30
- 1.7.1 Module based flow 31
- 1.7.2 Difference based flow 35
- 1.7.3 Early access partial reconfiguration flow 36
- 1.8 Definitions 37

2 PARTIAL DYNAMIC RECONFIGURATION APPLICATIONS 39
- 2.1 Partial dynamic reconfiguration advantages 39
- 2.2 Dynamic reconfiguration applications 40
- 2.2.1 A network controller application 41
- 2.2.2 Cryptography applications 41
- 2.2.3 Adaptive control 42
- 2.2.4 Video streaming 44
- 2.2.5 Other applications 46

3 DEVICE CHARACTERIZATION 48
- 3.1 FPGA families and models 48
- 3.1.1 Spartan 3 49
- 3.1.2 Virtex II Pro 50
- 3.1.3 Virtex 4 51
- 3.2 Configuration conventions and file formats 53

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

3.2.1 Configuration resources numbering scheme 53
3.2.2 The RPM grid 55
3.2.3 UCF file format 57
3.3 Available tools 58
3.3.1 ISE - Integrated Software Environment 58
3.3.2 FPGA editor 59
3.3.3 Floorplanner 61
3.3.4 PlanAhead 62
3.3.5 Jbits 63

4 **METHODOLOGY** 66
4.1 The Earendil flow 66
4.2 Application issues 69
4.2.1 PDR constraint satisfaction 70
4.2.2 PAR programs constraint interpretation 71
4.3 Exploring bitstream relocation 74
4.4 Area constraint modification 76
4.5 The proposed approach 77
4.5.1 Input data 77
4.5.2 The overall system 79
4.5.3 Advantages of the proposed approach 81

5 **IMPLEMENTATION** 87
5.1 Data organization and class definition 87
5.1.1 The Project class 87
5.1.2 The Bitstream class 89
5.1.3 Bitstream FAR progression 94
5.1.4 Far progression tables and occupation data retrieval 99
5.1.5 The UCF class 106
5.1.6 DB class 107
5.1.7 RPM class 108
5.2 Canvas class 111
5.3 Feature implementation 111
5.3.1 Validation of the flow constraints 112
5.3.2 Validation of area constraints 112
5.3.3 Area conflict retrieval 113
5.3.4 Conflict display in the evolution of the application 114
5.3.5 Bitstream relocation 116
5.3.6 UCF view 123

6 **CASE STUDY** 124
6.1 The chosen architecture 124

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

	6.2	Validation of the UCF file	128
	6.3	Area conflicts display	135
	6.4	Exploration of the relocation possibilities	137
	6.5	Conclusions	143
7	**CONCLUDING REMARKS AND FUTURE WORK**		145
	BIBLIOGRAPHY		147

LIST OF TABLES

TABLE **PAGE**

I FOUR INPUT LUT EXAMPLE. 6

II TYPE 1 PACKET HEADER WORD FORMAT 15

III TYPE 2 PACKET HEADER WORD FORMAT 16

IV VIRTEX 4 CONFIGURATION REGISTERS. 17

V SPARTAN 3 AND VIRTEX II PRO FAR FORMAT. 19

VI VIRTEX 4 FAR FORMAT. 19

VII FPGA FAMILIES, MODELS AND REFERENCE MANUALS USED IN THIS WORK 48

LIST OF FIGURES

FIGURE		PAGE
1	Generic architecture of an FPGA	5
2	Virtex II Pro slice architecture - top half.	8
3	Direct interconnection.	9
4	Segmented interconnection	10
5	The general structure of a full configuration bitstream	14
6	FAR addressing in Xilinx FPGAs	20
7	Virtex II Pro FPGA configuration memory mapping	23
8	1D (left) versus 2D (right) reconfiguration model	28
9	A sample design conforming to the module based flow for partial reconfiguration	33
10	The hierarchy proposed in the early access partial reconfiguration flow	37
11	The logic view of an adaptive control application	43
12	XCS200 internals as shown in XIlinx Floorplanner	50
13	XCVP7 (left) and XCVP20 (right) internals as shown in XIlinx Floorplanner	51
14	XCVFX12 internals as shown in XIlinx Floorplanner	52
15	An example of the resource numbering system used to refer to resources across Xilinx devices.	54
16	An example of the numbering style adopted in the RPM grid.	56
17	A screenshot of a placed and routed module in FPGA editor.	60

LIST OF FIGURES (Continued)

FIGURE **PAGE**

18 A slice block as shown in FPGA editor. The description of the selected resource is outlined in red in the console of the program. 61

19 Earendil flow overview 67

20 A representation of a possible error caused by the PAR process. The reconfigurable region is defined by the red outline, the allocated logic is pictured in yellow. 73

21 Three instants in an application life cycle. RFU 1 must be relocated to be configured together with RFU 3 in the third instant. 75

22 Four possible placements of a RFU considering the position of reconfigurable resources. 76

23 The overall logic structure of the Rebit framework 79

24 Rebit project panel 88

25 The automaton that parses a bitstream file 91

26 Spartan 3 XC3S200 far progression and configuration memory map table 100

27 Virtex II Pro XC2VP7 and XC2VP20 far progression and configuration memory map tables 100

28 Virtex 4 XC4VFX12 far progression and configuration memory map table 101

29 FAR progression vector coloring according to bitstream content 103

30 The representation of the resources of a Spartan 3 XC3S200 FPGA as the result of the interpretation of RPM grid data 109

31 A sample conflict graph between 6 different RFUs 114

32 The conflict sub-matrix corresponding to the example in Figure 31 . . . 115

33 The conflict sub-matrix filtered to display only a subset of the RFUs. . 116

34 The 3D view realized to browse the different moments of the application evolution showing the appropriate RFUs. 117

LIST OF FIGURES (Continued)

FIGURE **PAGE**

35 A sample of the sliding window algorithm used to compute the possible placement of partial bitstreams 120

36 A screenshot of the program showing the positions computed for the partial bitstream relocation. From the original placement in the top left corner, four other different placements have been found. 122

37 The representation given by the FPGA editor of the configuration of the XC4VFX12 FPGA with the static part of the evolvable hardware case and the edge detection module configured. 126

38 Screenshot of the Rebit project pane with the case study files loaded and the results of bitstream parsing. 127

39 Screenshot of the Rebit UCF View with the initial slice bounds defined in the UCF file displayed. 129

40 Screenshot of the Rebit Validation panel reporting the results of the validation of the UCF reconfigurable region and of bitstreams inclusion checks. 130

41 Overlaying of two screenshots from the Relocation View and from the UCF View panels showing how the partial bitstream (red) overflows a reconfigurable region (green). 133

42 Screenshot of the editing of the reconfigurable region (in blue). 134

43 Screenshot of the validation results after the alteration of the UCF constraints for the RR of the case study. 136

44 Screenshot of the Area Conflicts pane after the computation of the conflicts between the two partial bitstreams of the project. 137

45 Screenshot of the Relocation View pane after the computation of the feasible positions for the edge detection partial bitstream. 138

46 Screenshot of the UCF View pane with the representation of the two reconfigurable regions proposed. 140

LIST OF FIGURES (Continued)

FIGURE **PAGE**

47 Three screenshots of the program showing three phases of the framework utilization in the relocation of partial bitstreams: the conflict graph prior to relocation (a), the relocation view showing how the placement is changed from top to bottom half (b) and the resulting conflict matrix (c). 142

48 The 3D browser for the static photos implemented in the program showing the filtered conflicts on the basis of the four static photos supplied. Each layer in the 3D space has the incidence matrix of the conflict graph filtered by a particular static photo. 143

SUMMARY

Reconfigurable computing is a research topic in constant growth, as the possibilities and the technology offered by the reconfigurable devices improve year after year, both in terms of raw available configurable logic resources and in terms of the possibilities offered to exploit the logic resources themselves.

The state of the art of reconfigurable devices is currently represented by FPGAs. These chips offer a large array of configurable logic blocks, able to implement arbitrary hardware functionalities. This array is usually enriched by hardware cores embedded on the chip itself in order to offer even greater performance when implementing well-defined hardware functionalities such as DSP functions or multiplications. Recently even general purpose processors have made their way into the FPGA arrays, pairing the flexibility of software and the flexibility of custom hardware and broadening even more the set of applications that can be implemented as a System on Chip taking advantage of custom hardware.

FPGAs are thus not used only for rapid hardware prototyping, but also as components of deployable real-world systems, with a broad and growing range of applications that exploit the unique performance speed-ups given by hardware acceleration and flexibility in the customization of this hardware. Their strict relation to ASIC technology has allowed the manufacturers to adapt existing VLSI development frameworks to FPGAs, allowing developers to synthesize their own custom hardware at a fraction of the cost required for a full ASIC production cycle.

This work will focus on a particular set of methodologies for the development of FPGA-based

systems, that is *partial dynamic reconfiguration* of those devices. These methodologies fully exploit the reconfigurable nature of FPGAs by allowing a portion of the hardware to be changed at runtime to support a new functionality while the rest of the system keeps running. These techniques are rather new due to the fast advances in FPGA technology. In particular, what has made them possible was the introduction by FPGA manufacturers of the possibility to configure portions of the devices at runtime.

Partial dynamic reconfiguration is one of the key features that make FPGAs unique devices, offering a degree of freedom not available in other akin technologies and in some cases pushing FPGA-based solutions towards the standard application platform.

However the available methodologies for the development of systems based on partial dynamic reconfiguration are currently based on many manual steps, do not always yield a deployable result and their overall application is error prone, shifting the development efforts towards the resolution of numerous subtleties related to the reconfiguration techniques and not towards the real objective of the development process itself.

This work thus aims at possibly relieving part of the development process from these issues, and to give the system designers a novel framework targeted at debugging the results of their work rather than having to actually download it and test it on the FPGA.

Since many errors can be spotted at low level, i.e. from the actual bitstreams that will configure the device, a debugging part of the framework will be devoted to parse these configuration files and match them against the constraints imposed by the design.

Another portion of the framework will instead be focused on aiding the system designer in the

definition of the application constraints for a partial dynamic reconfiguration solution, giving the opportunity to visually plan the phases of the application execution life and spot possible problems arising from a particular combination of partial functionalities configured in the same moment on the device.

Chapter 1 provides an introduction on how the technology in FPGAs work, showing how the concept of reconfigurable hardware is implemented via the use of Configurable Logic Blocks, Look Up Tables and interconnection resources within a generic FPGA. The last part of the chapter is then devoted to the explanation of the internal format of bitstream configuration files in Xilinx FPGAs, i.e. the files produced by the FPGA vendor toolchain to configure the chip and deploy the system into a functional application on a custom board, and how this format is actually interpreted by the FPGA configuration logic to initialize or overwrite its configuration memory space. The second part of this chapter introduces the model of partial dynamic reconfiguration in FPGAs, with an explanation of the flows proposed by the system vendor to actually implement this procedure. The taxonomy of the available reconfiguration techniques is also explored in this chapter, in order to understand what are the alternatives and the techniques akin to partial dynamic reconfiguration. A hint of the issues involved in the application of these flows from a system designer's perspective will also be given. At the end of the chapter a list of definitions and acronyms used throughout the work will be provided.

Chapter 2 is devoted to the analysis of a set of applications taken from the current literature that exploit partial dynamic reconfiguration to boost performance and gain flexibility. The two fundamental advantages of partial dynamic reconfiguration are introduced, and some design

trade offs are explored in the end, showing how partial dynamic reconfiguration is not always the best possible solution in a particular application domain.

Chapter 3 delves deeper than the analysis performed in chapter 1 into the architectural details of the FPGAs used in this work. Moreover, some conventions used by the chip vendors in referring to internal resources of the FPGA for reference and constraints definition purposes are introduced in the last part of the chapter. Finally, the existing toolset available to the system designer for the analysis of a reconfigurable system will be introduced. These programs actually represent the available tools for the debugging and verification of partial dynamic reconfiguration systems, and for the planning of such systems.

Chapter 4 states the motivations behind the present work, namely to support the system designer in the debugging of a reconfigurable system and giving a representation of the evolution of the reconfigurable functionalities over time, in an application-dependent fashion.

The integration of this framework in a broader context - the EARENDIL flow developed in our lab - is also discussed.

Chapter 5 describes the choices and techniques used to realize the proposed framework, with the description of the Rebit program that implements the concepts introduced in chapter 4. Detailed descriptions of the code used to parse and interpret bitstream files, the graphical user interface and of the controls over the available data set are introduced.

Chapter 6 introduces a case study in which the framework developed in this master's thesis has been tested and put to use.

Chapter 7 summarizes the work done in the direction of aiding system designers in the imple-

mentation of partial dynamic reconfiguration systems, showing how the proposed framework can effectively be used in the process of developing such systems. Future work in the improvement of this framework are eventually discussed.

CHAPTER 1

FPGA TECHNOLOGY AND DYNAMIC RECONFIGURATION

This chapter will introduce a family of silicon devices, FPGAs, exploring their features and applications, while giving some basic hints at their architecture. This work will deal with the low level configuration details of these devices, so the last part of this chapter will be devoted to explore the file type that contains the configuration information to initialize the state of these reconfigurable chips, the bitstream file. For describing the bitstream composition, the details have been aggregated from the three different families of FPGAs examined in this work, namely the Spartan-3, Virtex-II Pro and Virtex 4 families manufactured by Xilinx Inc.

The chapter will then illustrate a particular technique that has been viable with most recent FPGA devices, called Partial Dynamic Reconfiguration. To fully understand what this technique is, the concepts of reconfigurable computing, static and dynamic reconfiguration and the taxonomy of dynamic reconfiguration itself must be analyzed. In this way partial dynamic reconfiguration can be correctly placed in the set of system development techniques that it is possible to implement on a modern FPGA chip.

1.1 FPGA overview

FPGAs (*Field Programmable Gate Array*) are a particular family of integrated circuits destined to custom hardware implementation, with the key property of being capable of re-

configuration for an infinite number of times. Currently FPGAS are the state of the art of Programmable Logic Devices (PLD), hence this work has been focused on these particular chips. Reconfiguring an FPGA means changing its functionality to support a new application, and it is equal to have some new piece of hardware - mapped on the FPGA chip - to implement a new functionality. In other words FPGAs make it possible to have custom designed high-density hardware in an electronic circuit, with the added bonus of having the possibility of changing it whenever there is the need of, even while the whole application is still running. An HDL language such as VHDL or Verilog is used to describe the functionality to be implemented on the device, then the design software of the device manufacturer translates the description of the hardware into a configuration file for the device that can be downloaded on it. These software tools are analogous to those employed in ASIC chip design, in the sense that they convert a hardware specification into an actual netlist that can be synthesized, placed and routed on an actual piece of hardware such as an ASIC or a FPGA.

The flexibility of having custom, changeable hardware in an application is the factor that has determined the popularity of FPGA devices in a broad range of fields. If the FPGA is paired with a general purpose processor, for example, the most demanding sections of the software can be translated into hardware cores that accelerate program execution, yielding notable speedups in the overall execution, especially when software sections executed serially on the processor can be translated into hardware that can exploit the parallelism of the algorithm. This is a viable technique in high performance computing, where software kernels such as FFT or image

processing algorithms are implemented on a FPGA, leaving the rest of the program to execute on a general purpose processor. In (Karanam et al., 2006) the authors show the results of executing the Smith-Waterman algorithm - a sequence alignment algorithm in bioinformatics - on a hybrid platform composed of a general purpose processor and an FPGA. The original program is profiled to locate the most computationally intensive functions, then these functions are translated into hardware using an HDL language and finally communication issues between the processor and the hardware on FPGA are taken into account trying to eliminate bottlenecks. The result is a significant speedup in the execution time of the algorithm. This methodology is broadly used whenever there is the need of accelerating software running over large sets of data, and is commonly referred to as *hardware-software codesign*.

FPGAs can also be used without a paired microprocessor, by implementing all of the application functionalities in hardware. In this case the hardware implemented on the FPGA covers all the data path from the inputs to the outputs of the application. The advantage in this is that the hardware is easily replaceable by downloading an appropriate configuration file onto the chip, rather than having the circuit physically replaced, and this is a key factor in applications such as network appliances, where a node of the network can be easily reconfigured off-site for network upgrades or maintenance. This is one of the reasons why FPGAs can be chosen over ASICs (*Application Specific Integrated Circuits*) in designing custom hardware.

The number of logic gates on the most recent FPGA models has also opened the way for the implementation of complete systems on chip (SoC) on these devices, given the fact that many FPGAs now have one or more processors directly embedded into the silicon wafer, and that even if those processors are not available, soft core processors can be used. Soft core processors are complete processors IP-Cores described in an HDL and synthesizable on FPGAs. Xilinx provides the *Microblaze* and the *Picoblaze* soft core processors to be used in architectures for their FPGAs.

1.2 FPGA architecture

This section will briefly hint at the distinctive features of FPGAs, in particular to clarify how they can implement custom hardware via their reconfiguration. The architecture of a generic FPGA is illustrated in Figure 1. The FPGAs used throughout this work are all manufactured by Xilinx, so the following architectural details apply to the chips produced by this company. However the general principles on which FPGAs are made are the same even across different chip brands. The three main building blocks of an FPGA are:

1. Logic blocks (CLB)
2. IO blocks (IOB)
3. Communication resources

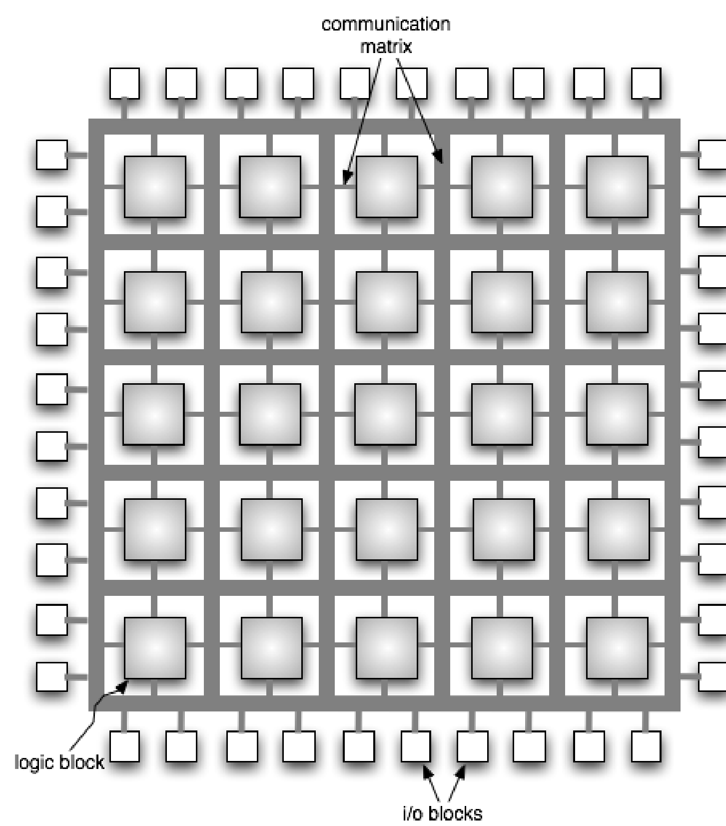

Figure 1. Generic architecture of an FPGA

1.2.1 Logic blocks

Configurable logic blocks (CLBs) are the main component of an FPGA. They can have one or more function generators realized with look up tables (LUT), that can implement an arbitrary logic function according to their configuration. In these components, the result of the function is stored for every possible combination of the inputs, such that a 4-bit LUT will require 16 memory cells to store the function, no matter its complexity. Table I is an example of a four input LUT. Around the LUT there is the interconnect logic that routes signals to and

in1	in2	in3	in4	out
0	0	0	0	0
1	0	0	1	1
0	0	1	0	0
0	0	1	1	1
1	1	0	0	1
1	1	0	1	0
0	1	1	0	1
1	1	1	1	1
1	0	0	0	0
0	0	0	1	1
0	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
0	1	1	0	1
1	1	1	1	1

TABLE I

FOUR INPUT LUT EXAMPLE.

from the LUT, implemented using standard logic gates, multiplexers and latches. Therefore, during the configuration process of an FPGA, the memory inside the look up tables is written to implement a required function, and the logic around it is configured to route the signals correctly in order to build a more complex system around this basic building block. In Xilinx FPGAs a single CLB contains a set of four slices that in turn contain two look up tables and the necessary interconnect hardware. As an example, Figure 2 shows the architecture of the top half (one single LUT) of a slice of the Virtex II pro CLB.

1.2.2 IO blocks

The input output blocks (IOBs) have the function of interconnecting the signals of the internal logic to an output pin of the FPGA package. There is one and only one IOB for every I/O pin of the chip package. The IOB have their own configuration memory, storing the voltage standards to which the pin must comply and configuring the direction of the communication on it, making it possible to establish mono-directional links in either way or also bidirectional ones.

1.2.3 Communication resources

The interconnection resources within an FPGA allow the arbitrary connection of CLBs and IOBs.

The main modes of interconnections are direct and segmented.

- Direct interconnection is made of groups of connections that cross the device in all its dimensions. Logic blocks put data on the most suitable channel according to data des-

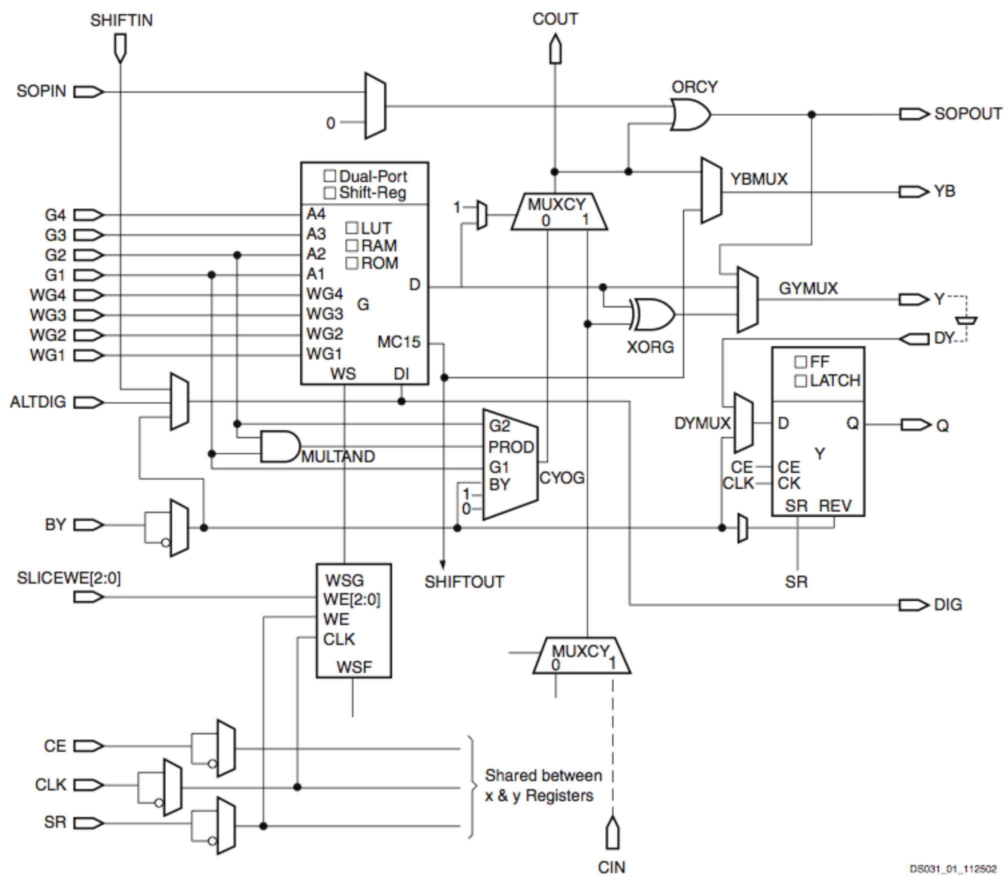

Figure 2. Virtex II Pro slice architecture - top half.

tination. This implementation usually includes some additional short-range connections that link to nearby blocks. An illustration can be seen in Figure 3.

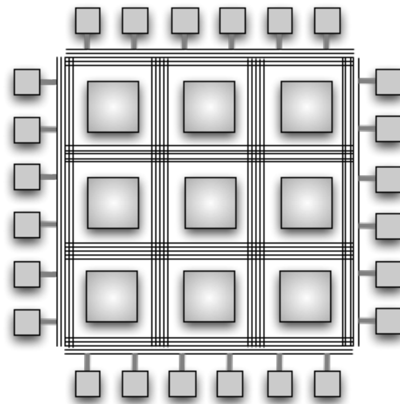

Figure 3. Direct interconnection

- Segmented interconnection is based on lines that can be interconnected using programmable switch matrices. Also in this kind of interconnection there are lines that cross the entire device, in order to maximize the speed of communication and limit signal skew. An illustration can be seen in Figure 4.

The main advantage of direct interconnection is that parasite resistance and capacitance are almost constant, resulting in an improved predictability of the propagation times of the signals. Segmented interconnections offer a reduced power dissipation because resistance and capacity of the interconnection lines are only those of the interconnection length between the blocks.

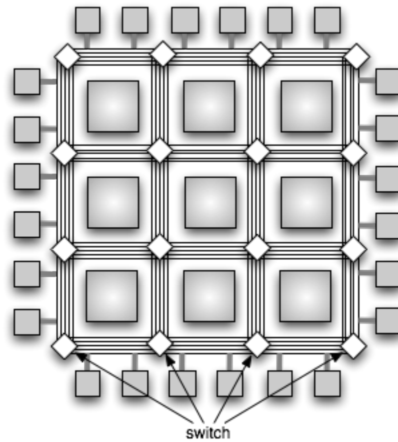

Figure 4. FPGA using the segmented interconnection model

The three building blocks presented so far interconnect together in the device as to create a communication infrastructure composed of the communication lines and IOBs around a bi-dimensional array of CLBs, which covers most of the available die area and represents the true FPGA building block. The memory cells attached to every configurable resource control its key features, in such a way that the IO voltage standards of a IOB are controlled by particular values in its corresponding memory cell, the interconnections among the communication infrastructure are controlled by setting appropriate bits in the configuration memory, and the equations in the LUTS are controlled in the same way. All of this configuration memory is made of SRAM memory elements, and it's therefore volatile: when the device is power-cycled all of its configuration is lost and it must be started fresh with a new configuration. Usually an external

machine downloads the configuration on the FPGA via one of its configuration interfaces, and sends a start command to signal that the configuration has taken place. Some boards also offer a ROM where configuration is stored, so that it can be subsequently downloaded on the FPGA on power up. The file that stores the information that is copied over the configuration SRAM memory of the FPGA is called *bitstream* and can be either full or partial according to the extent of configuration memory addressed in it.

1.2.4 Additional resources

Most FPGAs are not composed only of the three components described in the previous paragraphs, but they have additional resources directly embedded on the die. Such resources are RAM cells - called *block ram* in Xilinx FPGAs - processors, DSPs, multipliers and so on. In this way system designers can take advantage of hardware already present on the chip and integrate it in their designs without the need of having to implement all of the desired functionalities on the configurable resources but exploiting the speed and the functions of pre-made embedded hardware cores. As an example, the resource array of the Xilinx Virtex II FPGA XC2VP20 (Xilinx Inc., 2003) is heterogeneous, because it is composed of a CLB array interleaved with hard-cores such as an embedded PowerPC processor, 88 blocks of RAM each capable of 18Kb of storage, 8 multi gigabit transceivers and so on. A design can therefore be composed of two kinds of hardware: hard-cores and soft-cores. The hard-cores on an FPGA thus enrich its functionalities and improve the overall speed of architectures that make use of them.

1.3 The configuration bitstream

A bitstream file[1] is a binary file in which configuration information for a particular Xilinx device is stored, that is where all the data to be copied on to the configuration SRAM cells are stored, along with the proper commands for controlling the chip functionalities.

Bitstreams can be either partial or full. A full bitstream configures the whole configuration memory of the device, and it's used for static design or at the beginning of the execution of a dynamic reconfiguration system, to define the initial state of the SRAM cells. Partial bitstreams configure only a portion of the device and are one of the end products of any partial reconfiguration flow.

FPGAs provide different means for configuration, under the form of different interfaces to the configuration logic on the chip. There are several modes and interfaces to configure a specific FPGA family, among them the JTAG download cable (which is the method used in this work), the SelectMAP interface, for daisy-chaining the configuration process of multiple FPGAs, configuration loading from PROMs or compact flash cards, microcontroller-based configuration, an internal configuration access port (ICAP) and so on, depending on the specific family. A full explanation of these modes and techniques goes beyond the scope of this work, and the reader is pointed to the user guide of the family of interest to find out more details about the various configuration possibilities.

Whatever the chosen configuration interface and mode, a bitstream file is a necessary prereq-

---

[1] Bitstream files have the .bit extension

uisite for the successful operation of any Xilinx FPGA.

In every FPGA a *configuration logic* is built on the chip, with the purpose of implementing the different interfaces for exchanging configuration data and to interpret the bitstream to configure the device. A set of *configuration registers* defines the state of this configuration logic at a given moment in time. Configuration registers are the memory where the bitstream file has direct access. Actual configuration data is first written by the bitstream into these registers and then copied by the configuration logic on the configuration SRAMs.

The details explained in the following part may vary in some minor aspects across different FPGA families, but the basic principles of configuration and bitstream composition hold true for all of them, that is the bitstream composition and the underlying mechanisms in the configuration logic have been kept the same across different technological families.

The reference documents for understanding the configuration details of the FPGA families taken into consideration in this work are (Tseng, 2004) for the Spartan 3 family, (Xilinx Inc., 2007b) for the Virtex II Pro family and (Xilinx Inc., 2007c) for the Virtex 4 family.

1.3.1 Overall bitstream structure

A common full bitstream file is generally made of five components:

- a comment part, stored mainly as the ASCII representation of the comment, reporting file name, creation date and time and part model number.
- a dummy word (0xFFFFFFFF) and a synchronization word (0xAA995566), with the purpose to align the parser on 32-bit word boundaries and to signal the start of the actual configuration commands.

- a series of packets, with the respective headers and payloads, that are used to perform writes to the configuration registers of the configuration hardware of the device.
- a series of configuration words.
- another sequence of packets to write to the configuration register and issue start up commands.

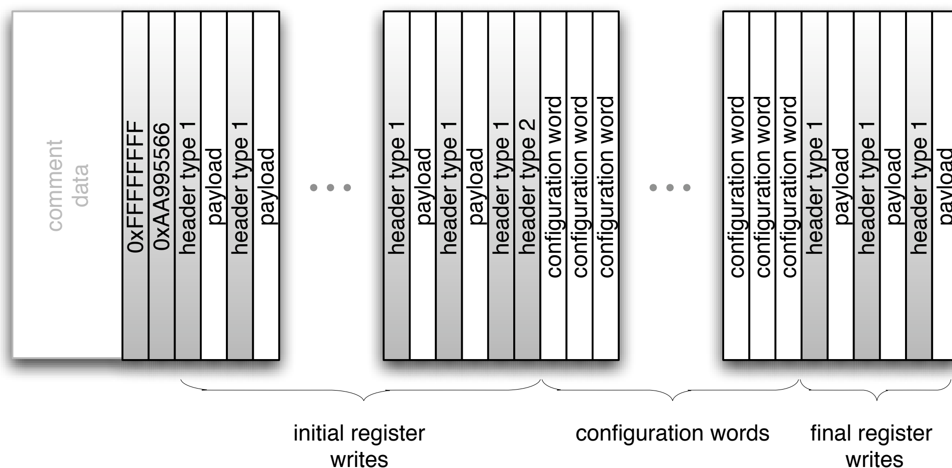

Figure 5. The general structure of a full configuration bitstream

After the configuration logic has been aligned with the special synchronization word, the bitstream is processed and interpreted on 32-bit boundaries, word after word. The register writes at the beginning and at the end of the actual bitstream, i.e. the bitstream part after the

sync word, are realized via a packet structure of a header and a certain number of payload words.

1.3.2 Packet headers

The header word defines the type of operation to perform on the configuration register, the address of the register to write to and the number of subsequent data words destined to be written on that register. Table II illustrates the composition of the 32 bit type 1 header, 'R' stands for a bit position reserved for future use, while an 'x' indicates any bit.

Type 1 packets are the most common packet type in the configuration bitstream. Type 2 packets

Header Type	Opcode	Register Address	Reserved	Word Count
[31 : 29]	[28 : 27]	[26 : 13]	[12 : 11]	[10 : 0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

TABLE II

TYPE 1 PACKET HEADER WORD FORMAT

are only used when the number of bits in the word count field of the type 1 header is insufficient to count a big number of configuration words. The type 2 packet has been created only for the purpose of reducing the overhead caused by the insertion of too many type 1 packets in long writes to the same configuration register. The header for the type 2 packet is defined in Table III: the majority of the header is used to store the count of the subsequent words and

Header Type	Reserved	Word Count
[31 : 29]	[28 : 27]	[26 : 0]
010	RR	xxxxxxxxxxxxxxxxxxxxxxxxxxx

TABLE III

TYPE 2 PACKET HEADER WORD FORMAT

no space is devoted to the register address, which is always defined in the previous word, that must be a type 1 packet header with word count set to zero. Therefore a type 2 packet must always follow a type 1 header, and it can be seen only as an extended word count field for the type 1 packet.

The significant data to understand the bitstream format is therefore in the header of type 1 packets. In these headers, the opcode field is usually set to a null operation or to a write operation in configuration bitstream. When in write mode, the last 5 bits of the Address field select one among the possible configuration registers, while the word count field indicates how many words the configuration logic must expect as a payload of the header.

1.3.3 Configuration registers

The write operation performed by type 1 and type 2 packets have the purpose to write to specific configuration registers. The registers of a Virtex 4 FPGA are listed in Table IV. Each of these registers controls a particular feature of the device. For the purpose of understanding the configuration process, the relevant registers are the CRC, IDCODE, FAR and FDRI registers. All the other registers are used for controlling aspects other than the configuration, and

Register Name	Read/Write	Address	Description
CRC	RW	00000	CRC register
FAR	RW	00001	Frame Address Register
FDRI	W	00010	Frame Data Register, Input
FDRO	R	00011	Frame Data Register, Output
CMD	RW	00100	Command Register
CTL	RW	00101	Control Register
MASK	RW	00110	Masking Register for CTL
STAT	R	00111	Status Register
LOUT	W	01000	Legacy Output Register
COR	RW	01001	Configuration Option Register
MFWR	W	01010	Multiple Frame Write
CBC	W	01011	Initial CBC value register
IDCODE	RW	01100	Device ID register
AXSS	RW	01101	User bitstream access register

TABLE IV

VIRTEX 4 CONFIGURATION REGISTERS.

for their detailed description the reader is pointed again at the documentation of the family of interest.

CRC register.

The CRC (Cyclic Redundancy Check) register is used to check the correct transferring of the configuration data by storing the result of a standard 16-bit checksum algorithm. The bitstream has a precalculated checksum value at the end of configuration words, that is checked against the value calculated by the configuration logic. If the two values don't match an error situation is created and the configuration process must be repeated. The CRC verification can

be disabled for each family by setting the appropriate bit of the COR register.

IDCODE register.

To avoid the downloading of a bitstream created for the wrong device, the IDCODE register must be written with a 32 bit word that is unique for every device. This word is also hard wired into the configuration logic, and the comparison of the contents of the register with the hard wired value can determine the correctness of the bitstream.

FAR register.

The FAR (Frame Address Register) is where the position of the first configuration frame is written, and it is thus important for knowing the physical location in partial bitstreams. In full bitstreams the value written to this register is always zero. This register is automatically incremented by the FPGA's configuration logic when the words that make up a configuration frame have been received, in such a way that only the first FAR value must be present in the bitstream file.

The FAR address register is broken up in different fields, depending on the specific FPGA family taken into consideration. Table V shows the FAR register of the Spartan 3 and Virtex II Pro families, while table VI illustrates the format of the FAR register in Virtex 4 devices. The terminology used in the manufacturer's user guides has been updated to be uniform across every family.

The **Block Type** field of the FAR address indicates in broad strokes what kind of resources the frame with that value will configure. Different types are allowed: configuration of CLBs,

unused	Block Type	Major Address	Minor Address	Frame Byte
[31 : 29]	[28 : 27]	[26 : 13]	[12 : 11]	[10 : 0]
0000	0xx	xxxxxxxx	xxxxxxxx	000000000

TABLE V

SPARTAN 3 AND VIRTEX II PRO FAR FORMAT.

unused	Top-Bottom	Block Type	Row	Major	Minor
[31 : 23]	[22]	[21 : 19]	[18 : 14]	[13 : 6]	[5 : 0]
000000000	x	xxx	xxxxx	xxxxxxxx	xxxxxx

TABLE VI

VIRTEX 4 FAR FORMAT.

IOBs and CLOCK lines has the first block type address, then BRAM values and BRAM interconnect lines have their own block type values.

The **Major address** field indicates what column of resources the frame is currently configuring and the **Minor address** field indicates a frame inside a particular configuration column. The additional fields in the Virtex 4 FAR address have been introduced to address portions of configuration resources other than entire vertical columns of the device array. In particular the **Top-Bottom** bit addresses the top or bottom half of the device and the **Row Address** is used to refer to a particular row of frames, in the bottom or top half, depending on the chosen model. Figure 6 aggregates all the possible information that is contained in a generic FAR word, either for the Virtex 4 family and for the Virtex II Pro and Spartan families, showing

what is addressed in each field; for the families that do not support a bi-dimensional granularity in configuration, only one row of columns must be considered in the figure, while the whole addressing scheme, comprehensive of the Row and Top/Bottom fields holds for the Virtex 4 family. The fields of the FAR address thus divide hierarchically the configuration memory

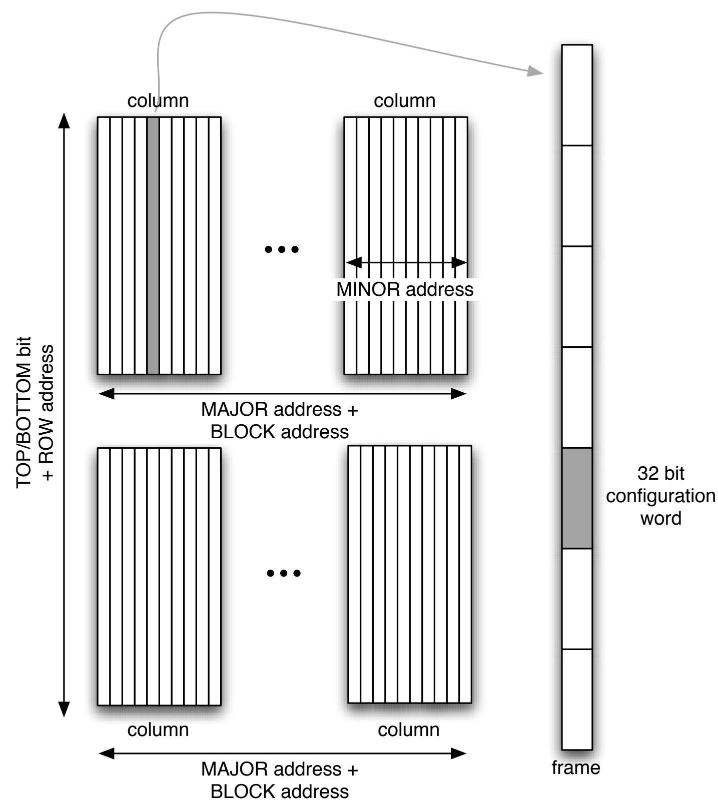

Figure 6. FAR addressing in Xilinx FPGAs

space. The top level component is a row of resources, addressed by the Top/Bottom bit and by the Row address respectively. Spartan 3 and Virtex II Pro can be considered as FPGAs having only one row of resources and thus adhere to this very same model.

The next level in the hierarchy is defined by the Block Address together with the Major Address[1]. These two fields define together which column in a row is configured. A frame within a column is then defined by its Minor Address, and there is no way to address configuration words within a single frame (the Frame Byte field of Table V is unused). The atomic configuration unit in a Xilinx FPGA is thus represented by the frame, as the addressing hierarchy clearly shows.

FDRI register.

The FDRI register is used to write the words that make up a frame. The FPGA configuration logic implements the FDRI register as a shift register, so that a frame is configured while the next one is being shifted in. For this reason in the bitstream there is some padding data in order to make the final writes to the configuration memory. As said before, the FAR address is automatically incremented whenever a whole frame has been written to the FDRI register.

The majority of a bitstream is thus made of the words that are written to the FDRI register, as shown in Figure 5.

---

[1]Xilinx documentation has various names for the Block Address: it is also called Column Address or Block Type field

1.3.4 Frame indexing

The bitstream file structure, as seen in the previous paragraphs, contains a large write to the FDRI register with a sequence of 32 bit words that configure portions of the FPGA, written in a sequence that does not report the particular address (the FAR address) in the configuration memory to which the data is written to. It is thus impossible to read a plain bitstream file and infer the portion of resources that it configures without the exact knowledge of the memory mapping of the particular FPGA to which the bitstream is going to be applied.

In other words, there is the need to understand the logic with which the FAR address is incremented in the internal configuration machine of the FPGA in order to correctly map each frame read in the bitstream with the correct resources.

The information provided in the vendor documentation hints at the addressing scheme used in the devices of the Virtex II Pro and Spartan families, but these details are not found in the documentation for the Virtex 4 family of devices.

Figure 7, as an example of the provided information, reports the configuration memory mapping explained in the user guide for Virtex II Pro FPGAs. The *column type* field in the figure gives information about what particular column of resources in the array is being configured with the frames belonging to a column, and shows the progression of the configured resources as the FAR is automatically incremented by the configuration logic: the first column configures the central clock routing resources, then the next column the left IOB column, then their interconnections, then the columns of CLBs and so on. A change from 0 to 1 in the *Block Address* field means that the frame of that particular column are configuring the contents of the block ram columns

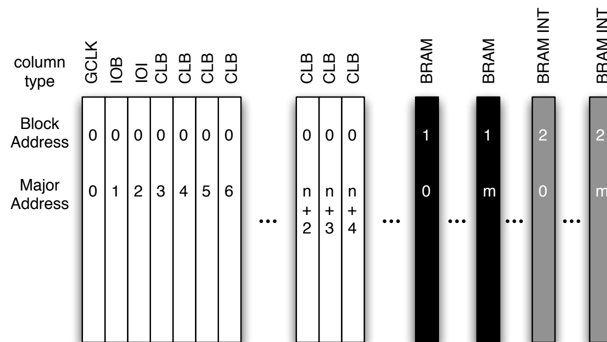

Figure 7. Virtex II Pro FPGA configuration memory mapping, from Xilinx Virtex II Pro User Guide (Xilinx Inc., 2007b)

on the device, and a value of 2 in the *Block Address* means that the frames are configuring the interconnections of the ram cells with the rest of the array.

The variables in this addressing scheme are given by the number of CLB columns and by the number of BRAM columns, marked with the letters 'n' and 'm' respectively in Figure 7.

1.4 Reconfigurable computing

The topic of *reconfigurable computing* has been introduced in 1960 by Gerald Estrin in (Estrin, 1960). That paper proposed a system that could efficiently exploit the resources offered by reconfigurable hardware. The reconfigurable device was paired with a standard processor that would configure a specific task on the resource, wait until the execution of the task and possibly reconfigure the resource for another task. This system, enhanced with reconfigurable hardware, could take advantage of hardware acceleration in the execution time of the configured tasks while not losing in the flexibility in terms of executable applications that the processor could

offer.

In this way it is realized in hardware what is usually realized with processors with a context switch between different processes: the different processes time-share the computational power of the processor, realizing different functionalities with the same resource set. This parallel between software and hardware can be made by comparing the reconfigurable hardware resource (such as an FPGA) to the processor, and the different functionalities mapped on the reconfigurable resource as the software processes. Several hardware functionalities can time-share the resources offered by the reconfigurable device in the same way that the processes are switched in and out for execution on the processor.

The implementation of reconfigurable systems as envisioned by Gerald Estrin has however been effectively put to use only in the last two decades. In fact the technology of reconfigurable hardware has seen significant improvements in the latest years, allowing a boost in both speed and configurable logic amounts in reconfigurable technology, and giving way to computing systems that can follow the reconfigurable model reporting significant speedups in application execution, via the exploitation of custom hardware. The main advantage of reconfigurable computing is therefore the improved flexibility available to designers: hardware can be seen as a dynamic entity rather than some static, hard-wired functionality realized on silicon, and the systems that comprise a reconfigurable resource can exploit hardware acceleration in multiple points, without having to have on the system board all of the required hardware functionalities at the same time.

Moreover, reconfigurable computing offers a way to test hardware blocks without having to

undergo the costly process of ASIC manufacturing, cutting the costs of hardware development by minimizing the impact of errors in the definition of the functionalities: in a reconfigurable system, in fact, the flexibility advantage of the reconfigurable resources allows the reconfiguration of a faulty functionality without any significant cost, even allowing in some architectures the replacement of faulty hardware while the overall system is live.

The flexibility of reconfigurable computing has attracted many researchers to envision a wide range of applications and systems, fostering the production of a variety of scenarios that together add up into the global reconfigurable computing model. The following sections will be devoted to the exploration of dynamic reconfiguration, a particular area of reconfigurable computing in which the reconfiguration process is taken as part of the whole application execution model.

1.5 Dynamic reconfiguration

The reconfiguration capabilities of FPGAs give the designers extended flexibility in terms of hardware maintainability: as seen in section 1.1 FPGAs can change the hardware functionalities mapped on them by taking the application offline, downloading a new configuration on the FPGA (and possibly new software for the processor, if any) and rebooting the system. Reconfiguration in this case is a process independent of the execution of the application and the process has the name of **Static reconfiguration**.

A different approach is the one that considers reconfiguration of the FPGA as part of the application itself, giving it the capability of adapting the hardware configured on the chip resources according to the needs of a particular situation during the execution time. This particular

case is called **Dynamic Reconfiguration**. In the case of dynamic reconfiguration, therefore, the reconfiguration process is seen as part of the application execution, not as a stage prior to it. Dynamic reconfiguration has its own particular taxonomy according to the features of the system, as explained hereafter.

1.5.1 Dynamic reconfiguration schemes

The first distinction is between **total** or **partial** dynamic reconfiguration. In total reconfiguration the entire device is configured with new data that totally replace the old ones. In partial dynamic reconfiguration, on the opposite, only a portion of the device is reconfigured, while the rest of the hardware mapped on the FPGA can possibly continue to operate transparently with respect to the reconfiguration process. It is the latter scheme that will be explored within this work. Complete systems on chip can be developed that employ dynamic partial reconfiguration, with a static part that runs the main control software and the rest of the resources devoted to being reconfigured over time in order to host different functionalities.

Dynamic reconfiguration can be either **internal** or **external**, depending on the entity that performs the reconfiguration: the FPGA can always be configured from an external interface such as JTAG, controlled by a host machine. In this case the reconfiguration is external, because all the commands for reconfiguration are issued from outside the chip. However some devices have an internal configuration port (the ICAP in Xilinx devices) that gives access to the configuration memory from within the FPGA itself. In this case it is possible to have a

processor on the FPGA that uses the ICAP port to reconfigure some resources. This is the case of internal reconfiguration, meaning that the reconfiguration process is self-contained within the FPGA itself, with no need of external entities. Internal reconfiguration can give way to full systems on chip that can adapt their own HW in response to changing application needs.

Dynamic reconfiguration can take place in a 1D or in a 2D space. in **1D reconfiguration** the resources affected by the process always span the whole height of the device. In other words it is impossible to configure one set of resources without having to reconfigure the whole columns they span as well. This reconfiguration scheme is imposed by the architecture of some devices such as those in Xilinx Spartan 3 and Virtex II families, because the smallest addressable configurable unit in these devices is the frame, a vertical line of resources.
On the opposite, **2D reconfiguration** allows the partitioning of the configurable resources in a 2D space, allowing the definition of different reconfigurable regions one on top of the other. The 1D limitation has been overcome in more recent FPGA families such as Virtex 4 and Virtex 5, which can reconfigure specific portions of the device without affecting the entire columns of corresponding resources. In this way more sophisticated architectures can be designed, exploiting the added dimension to the freedom in the choice of reconfigurable areas. Figure 8 illustrates these two different schemas: on the left the 1D scheme is pictured, showing how the whole FPGA height is occupied by a reconfigurable region (RR[1]). On the right the constraint

---

[1] A Reconfigurable Region - or RR - is a portion of the device destined to be reconfigured with different functionalities and is the place where the actual partial dynamic reconfiguration process takes place.

on vertical occupation is relaxed, thus the reconfigurable areas do not span the entire height of the device anymore.

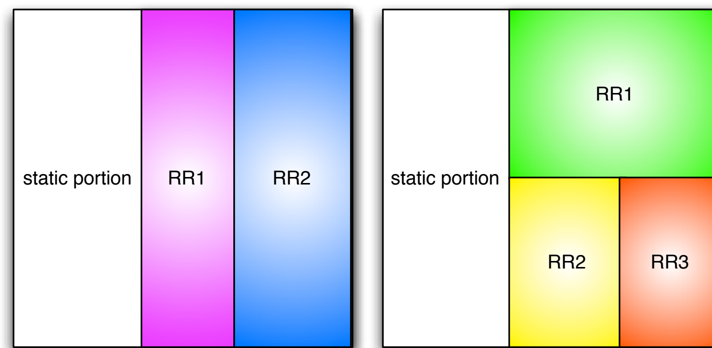

Figure 8. 1D (left) versus 2D (right) reconfiguration model

Dynamic reconfiguration can have a different granularity, distinguishing between **smallbits** and **module based** dynamic reconfiguration. The first approach consists in modifying a small portion of the device configuration bitstream, possibly even a single bit, in order to modify the functionality of a particular LUT in a CLB, or to change the parameters of a single IOB. The second approach, instead, addresses larger areas of the FPGA by creating HW components that can be switched back and forth into the system. The two techniques are addressed in (Xilinx Inc., 2004).

1.6 Partial dynamic reconfiguration issues

While the benefits in certain application fields of partial dynamic reconfiguration have been extensively demonstrated[1], some issues remain in the methodologies used to achieve a functional architecture.

The issue that is avoiding a wider diffusion of such architectures is certainly the lack of a complete software toolchain capable of taking into account partial dynamic reconfiguration in a sound manner, even if now novel methodologies have been proposed (Rana et al., 2007).

The current methodologies for Xilinx FPGAs (Xilinx Inc., 2004)(Xilinx Inc., 2006) , as an example, comprise a long series of steps that the developer has to undertake in order to convert the product of conventional CAD tools into the final full and partial bitstreams necessary to deploy a partial dynamic reconfigurable architecture. During the various steps, moreover, the system designer has to keep clearly in mind partial reconfiguration constraints and use the software to enforce them. Due to the relative novelty of partial reconfiguration techniques these procedures are not yet included into the manufacturer's software, and many operations have to be done manually. This factor may deviate the focus of the development process away from the real application towards these (minor) details, thus resulting in an extended time-to-market of the developed application. Another issue related to the lack of support for partial reconfiguration in the manufacturer's design flow is that some tools that make up the development flow are not reconfiguration aware. For example, the PAR (place and route)

[1] refer to chapter 2 for benefits and applications.

tool which is responsible of translating the design netlist into placed and routed components interprets some area constraints given by the user in a non stringent way, as those constraints were only hints to the actual placement and routing. While this is acceptable in a fully static design, when partial reconfiguration is involved this fact becomes a real issue, since a resource placed outside the boundaries of a reconfigurable area can overwrite the configuration of other regions that the designer has not meant to reconfigure, causing issues and often making the whole design to fail correct execution. As for the time being the only possible thing that the developer can do to check whether the design does not satisfy any of the given constraints is to manually inspect the design for flaws and constraint violations, with little or no possibility to repair the errors.

The aim of this work is to give the system developer a way to automatically inspect the resulting files of the project, as to perform high and low level checks that are aware of the partial dynamic reconfiguration techniques and issues.

1.7 Design flows for Partially Dynamic Reconfigurable Architectures

The process of designing a PDR architecture is often supported by design flows that aid the developer to go from the specification of the architecture to the final synthesis stage. This section will illustrate the flows proposed by the vendor of the FPGAs employed in this work to develop PDR systems on their FPGA models.

Xilinx provides detailed user guides and application notes to support partial reconfiguration on their products. This section will explore the three flows proposed for implementing such systems.

1.7.1 Module based flow

The module based flow for partial reconfiguration, described in (Xilinx Inc., 2004), provides the designer with a methodology for implementing a dynamic reconfiguration system provided that a module based design methodology is employed throughout the project. The module based methodology for the development of static designs is described in (Xilinx Inc., 2005), and basically consists in a set of guidelines that allow a project coordinator and a set of team members to work on a single project, given that the modules in the design can be developed separately, possibly by different people. The principle upon which this methodology relies is that the project coordinator creates a top-level design, in which each module (an arbitrary hardware functionality) is instantiated as a black box, i.e. as an entity of which only communication ports and functionalities are known. The top-level design is responsible of the cooperation of the different modules and for the creation of the overall system. Developers then implement these black-boxes into working soft-cores by giving an HDL description of their functionalities and by running the mapping, placement and routing tools on their design. Once all the modules are ready, a final assembly phase takes place, resulting in a downloadable bitstream for a particular FPGA, that implements all of the module functionalities comprised in the system described in the top-level design.

In essence three phases are involved in either the static or partial reconfiguration version of the module based flow:

Initial budgeting phase where the floorplan and the constraints of the overall design are defined.

Active module phase where each module is implemented via HDL description and place and route process.

Final assembly phase where all the modules are put together in a final system design.

The importance of the module based flow is clear when applied to the dynamic reconfiguration methodology: the black boxes of the modular approach represent the RFUs of the dynamic reconfigurable architecture, and the top level design is what describes the reconfigurable areas. The modular development flow therefore constitutes the basis for the successful development of a dynamic reconfigurable design. To make this approach feasible, however, some special considerations regarding partial reconfiguration have to be taken into account. These considerations translate into particular constraints that enrich the module based flow to support partial dynamic reconfiguration, basically taking into account the reconfiguration capabilities of FPGAs and inter-module communication. The constraints are thoroughly described in (Xilinx Inc., 2004) and will be explained hereafter. A picture of a sample final design configured on a FPGA is given in Figure 9.

In this sample there are three static areas that will never be reconfigured, two RAs where the modules (or RFUs to adopt the terminology used in this work) will be reconfigured and some communication facilities between the vertical boundaries implemented via bus macros.

Bus macros are pre-placed and pre-routed pieces of logic that make communication among modules feasible, even if the modules themselves are reconfigurable. Thanks to bus macros the system designer can define global communication facilities in the top level design, taking into account dynamic partial reconfiguration. In fact, every RFU will be described with bus

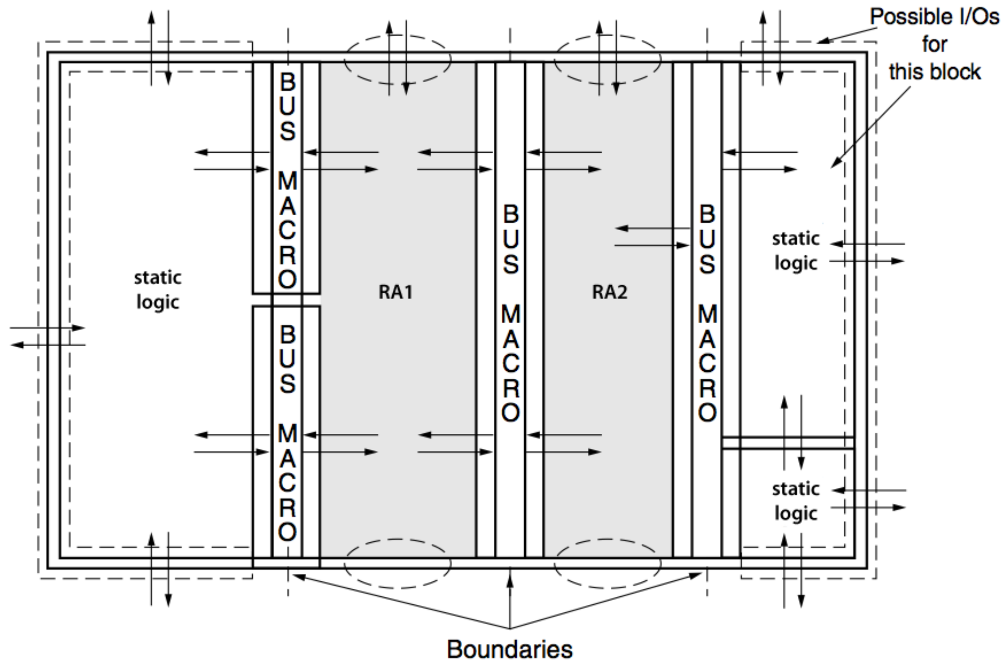

Figure 9. A sample design conforming to the module based flow for partial reconfiguration. Image from (Xilinx Inc., 2004).

macros in the exact same position as in the top level design, so that after reconfiguration the communication resources are still in the same place, changing only the module attached to them. Xilinx provides pre-routed bus macros for the FPGA families used in this work, and every bus macro provides a 4-bit bus over which communication is possible in both directions. Two key considerations must be made when using bus macros: the first one is that if a RFU requires communication with another RFU or static logic not immediately next to it, bus macro pathways have to be envisioned to pass across the logic in the middle of them. The second consideration is that reconfiguration of a RFU will temporarily interrupt communication over

the bus macros that are attached to that area. Due to the process with which reconfiguration takes place in Spartan 3 and Virtex II families of devices, the RAs must be of the same height of the device, since the smallest addressable reconfigurable area is the frame, that in these devices spans the entire height. This fact has another implication: all the IOBs above and under the reconfigurable area are included in it, and if the RA happens to be also on one of the sides of the device, all of the IOBs on that side are part of the area as well. In this way the IOBs belonging to a particular RR cannot be part of the top-level design at the same time. These considerations must be kept in mind when designing a system based on this flow. Moreover, there are other constraints expressed as properties of a particular RR:

RR width: the width of a RR must range from a minimum of four slices to a maximum of the full device width, in four slice increments.

RR placement: the RR must be placed on a four slice boundary, with a minimum horizontal placement in slice 0, other feasible placements given by multiples of 4.

RR resources: the resources encompassed by a RR are all the resources configured by the frames it spans, i.e all of the slices, three state buffers, block rams, multipliers, IOBs and routing resources.

and constraints expressed as properties of the design:

RFU placement: the placement of a RFU must be fixed and inside one RR.

Bus macro placement: bus macros must be placed in order to align the line that divides them in half with the boundary between the areas (static portion or RR) that they connect.

Static portion reconfiguration independence: the static portion of the design must not rely on the state of a particular RFU, that is it must be designed in order to be unaffected by the reconfiguration of some RR.

As stated previously, most of these constraints must be enforced and checked manually.

1.7.2 Difference based flow

While the module-based approach just described is used to configure in and out large blocks of logic, the difference based flow is suitable for small design changes. This flow is described in (Xilinx Inc., 2004) as well. Small changes in a design affect the equations of a LUT, the voltage standards of a IOB and so on. Using the *bitgen* command line utility from Xilinx, a difference bitstream can be produced that alters only the small portion of the configuration required to do the change, while the overall system still remains operational. These sort of changes are quicker with respect to the previous flow, because the exchanged information is much smaller. The changes to the design can be done either at the back-end level of the toolchain or at the front-end. In the former case the FPGA editor tool allows to visually inspect the FPGA and its configuration in order to make changes to LUT equations, I/O standards and BRAM content. In the latter case the HDL of the design is altered and synthesized, then the difference bitstream is produced again invoking the bitgen tool.

This flow is another way to implement dynamic partial reconfiguration, but it does not have all

of the constraints of the module-based flow, and can be applied to alter static designs, hence it is beyond the scope of this work.

1.7.3 Early access partial reconfiguration flow

The methodology proposed in (Xilinx Inc., 2006) is another flow for the dynamic reconfiguration on most recent FPGA families, namely the Virtex 4 family. The flow builds on the principles of the module based one and improves it on three key aspects.

- Most recent FPGAs have an improved configuration granularity, because rows of configuration space can be individually addressed instead of entire columns of resources. This fact makes a 2D reconfiguration scheme viable and is taken into account in the methodology.

- The second important improvement with respect to the module based flow is that signal routes in the base design can now cross the boundaries of the reconfigurable areas and thus do not require the employment of bus macros anymore.

- This flow adds support for the Virtex 4 family of FPGAs.

A strict hierarchical model must be followed to comply to the early access partial reconfiguration flow, and several components must be created in order to form this hierarchy. Figure 10 illustrates the hierarchy guidelines.

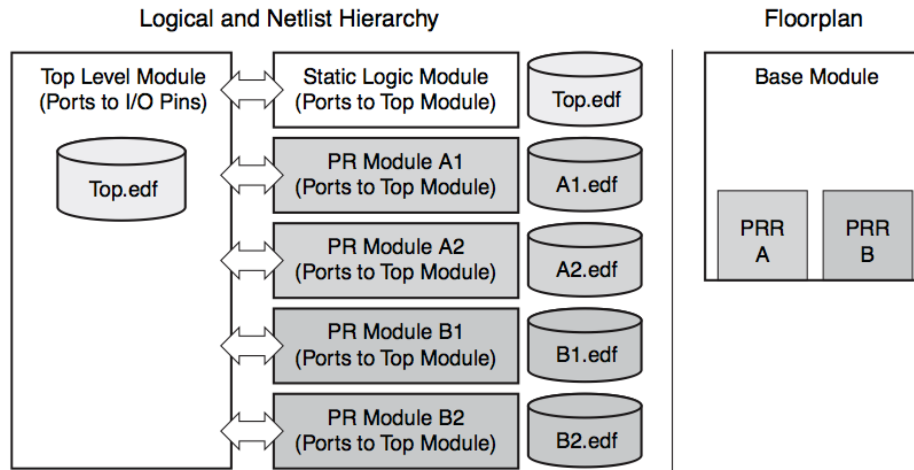

Figure 10. The hierarchy proposed in the early access partial reconfiguration flow. Image from (Xilinx Inc., 2006).

1.8 Definitions

Hardware: with this term we broadly intend functionalities implemented either with hard-cores or soft-cores, as opposed to functionalities implemented in software.

Hard-Core: a physical hardware core embedded on the die of the FPGA, available as a resource in user designs but not reconfigurable.

Soft-Core: a self contained hardware functionality implemented by configuring a particular set of the configurable resources of an FPGA.

IP-Core: Intellectual Property Core, used in hardware design to refer to a soft core implementing a defined functionality.

HDL: Hardware description language.

PAR: Place and router software provided by the FPGA vendor that translates a netlist into a description of the configuration of the device.

PDR system: acronym of Partial Dynamic Reconfiguration system.

RR - reconfigurable region: a portion of the device resources destined to be reconfigured with different RFUs over time. It is an area slot for functionalities to be configured into, and it has hardware macros that allow communication with the rest of the hardware present on the device.

RFU - reconfigurable functional unit: a soft-core developed in order to be configured in one or possibly different RRs.

Dynamic Partial Reconfiguration: Dynamic partial reconfiguration is a reconfiguration technique performed when the device is active.Some areas of the device, the reconfigurable regions, can be reconfigured while other areas remain operational and unaffected by the reprogramming, allowing to switch the hardware functionalities in response to the application needs.

SoC - System on Chip: a complete, self contained architecture composed by one or more processors and coprocessors implemented on a single chip and destined to be autonomous with respect to external hardware.

CHAPTER 2

PARTIAL DYNAMIC RECONFIGURATION APPLICATIONS

In this chapter a brief overview on the state of the art of applications than can possibly take advantage from partial dynamic reconfiguration is presented. The presented list of application doesn't want to be exhaustive by any means, but rather it aims at giving a glance of what it can be done with reconfigurable hardware. The possible advantages of employing partial dynamic reconfiguration in an application scenario will be presented, along with the development efforts in this field that are not necessarily related to a particular application domain.

2.1 Partial dynamic reconfiguration advantages

The advantage of employing a particular technique in hardware design can be expressed as the gain in computational speed over other solutions, the reduced power consumption, the resulting system flexibility, the reduced price and even as the gain in the easiness of the required programming effort, or as a combination of all these metrics.
Therefore there are certain application fields that benefit dramatically from partial dynamic reconfiguration, and other application fields where the effort to be put in implementing a PDR solution is too much with respect to the corresponding gains, or even application areas where a PDR solution would negatively affect the design performance.
The following application scenarios explain the application features that can justify the adoption of a PDR methodology.

Area requirements are too high for a static design.

There are some applications that require a consistent area for the deployment of the system. In this case more reconfigurable resources must be added to the system, or partial dynamic reconfiguration of the available ones can be implemented, in particular if the application can be partitioned into different phases, each building on the previous results, over a certain data set. In this case the different modules of the application can be configured on the device one after the other, to process data and obtain the final result, while keeping the area requirements low.

Some portion of the application must change over time and react to changes in its environment.

With partial dynamic reconfiguration a portion of the hardware can be adapted over time to cope with new application requirements, giving way to a higher flexibility in system design. In this way *adaptive* systems can be generated, overcoming the inherent static nature of HW solutions. While one could implement all of the possible behaviors in a monolithic static design, this would lead to a greater area occupation on the reconfigurable device, having only the section that implements the correct behavior active and the rest of the system idle. Partial dynamic reconfiguration allows switching in the correct functionality, exploiting efficiently the available reconfigurable hardware.

2.2 Dynamic reconfiguration applications

Given the advantages of the partial dynamic reconfiguration approach, this section will introduce some applications that exploit this technique.

2.2.1 A network controller application

In (Chaubal, 2004) the author implements a network controller capable of handling the TCP and UDP protocols by exploiting partial reconfiguration capabilities on Virtex-II FPGAs. In this application the partial reconfiguration process is exploited in order to react to changes at run time in either the number of open channels, the protocol used for a specific channel and the parameters of that protocol. In this way the network controller can support a wide range of situations by appropriately configuring the necessary hardware. Moreover, if one communication channel changes its parameters or the protocol, triggering a partial reconfiguration, other channels can remain unaffected and continue processing.

2.2.2 Cryptography applications

The field of cryptography has seen the development of many algorithms that can enforce the secure transmission of data over unprotected communication channels, and several standards have been implemented both as software components and as intellectual property cores (IP-Cores). Triple-DES, AES, DES, RSA, OpenPGP are some examples of these protocols. In this application domain flexibility in hardware is very important, as it permits to switch from one algorithm to the other at run-time, or to fine-tune an algorithm in its parameters. Moreover, if an algorithm is malfunctioning, reconfiguration offers the capability to replace the corresponding IP-Core with a secure one. The inherent parallelism of the algorithms can be exploited to efficiently configure the reconfigurable hardware resources, obtaining significant speedups with respect to analogous software implementations. (Castillo et al., 2005) presents a

cryptographic system that relies on open source hardware implementations stored as bitstreams in a configuration database, a static architecture portion that implements the parts common to all of the chosen algorithms plus the bitstream loader, and a reconfigurable slot (Reconfigurable Region) where the particular IP-Cores are downloaded.

2.2.3 Adaptive control

A controller has the duty of influencing the behavior of a particular system by sensing some variables of the system itself and generating a vector of control variables accordingly, creating a feedback loop that comprises both the controller and the controlled system. Adaptive controllers deal with the control of such a system in a changing environment, that is the controlled system behavior, and thus the controller, must react to changes in the environment of operation.

With conventional methods there is the possibility to generate a controller capable of reacting to every possible change in the environment, but the size of such a controller could be too big for its actual hardware implementation.

A more convenient approach is thus offered by reconfigurable hardware, in which the different controllers can be switched according to the environmental changes. As in the previous case, a database of the different modules must be kept in order to have them ready as the application requests them. The general structure of such a controller system can be seen in Figure 11. The active controller is chosen among a list of predefined ones and connected with the rest of the system to close the feedback loop. The supervisor component is in charge of sensing both the output of the controller and of the controlled system and switching the controller if necessary.

Its complexity could vary from a simple boolean function to a complex reasoner.

A sampling interval for the sensor variables is defined for the overall system, and the duty of

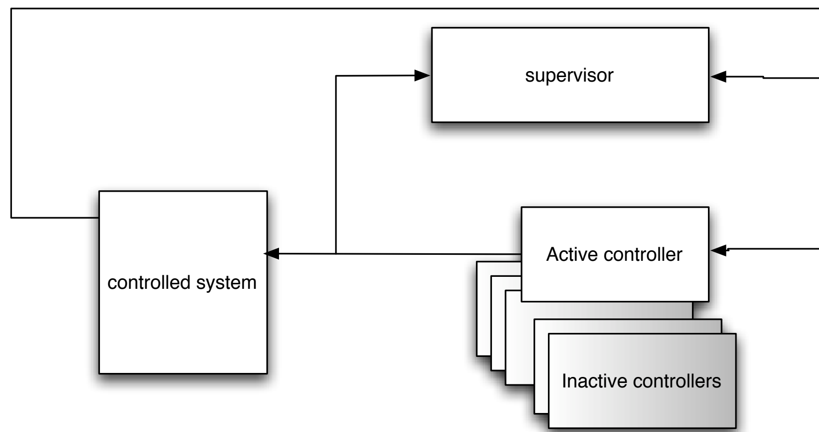

Figure 11. The logic view of an adaptive control application

the controller is to produce the vector of the control variables within every sampling interval. The controller itself could be implemented using different techniques, but the most common is given by:

$$x(k+1) = Ax(k) + Bu(k) \tag{2.1}$$

$$y(k) = Cx(k) + Du(k) \tag{2.2}$$

where A, B, C and D are coefficient matrices, x is the state vector of the controller, u is the input vector, y is the output vector and k is the time sample index.

These equations can be implemented efficiently in hardware by means of a distributed arithmetic, which consists in 'folding' the result of the matrix computation in the hardware.

In the resulting architecture a reconfigurable region is defined to hold the controller module, and a static portion of the design is created in order to have the supervisor module and the communication facilities. However, since reconfiguration time could be longer than the sampling interval, the controlled system could be left in a floating state until the new controller is configured. To solve this problem the architecture illustrated in (Danne et al.,) has been developed: in this approach two reconfigurable regions hold two different controllers, in order to supply the control vector at every sampling interval. In this case the reconfiguration time is added to the reaction time of the supervisor, but does not represent a possible failure of the overall controller in supplying its output in time.

2.2.4 Video streaming

Video streaming is the process of performing computations on image data belonging to a video sequence, picture after picture. Performance is very important in this application, because the results of the computations must be produced in real time, and this is the first reason why the reconfigurable computing approach has attracted many researchers in this field: performance. The second reason is the inherent adaptivity of the platform, which can be used to adjust the overall system by changing parts of it.

The computation on the video frames suits well to a pipelined structure, where each module in

the pipeline performs an operation on the data path. Usually the first module in the pipeline chain is in charge of capturing the input data, while the last is in charge of outputting the processed frames. The modules in between perform a specific image processing task, such as transformations or data extraction.

The hardware IP-Cores that compose the pipeline chain can be exchanged in a PDR architecture, while the rest of the system keeps running. It is thus possible to customize the processing pipeline without having to reconfigure the entire system. In this way for example the first module of the pipeline, the one in charge of collecting data, can be exchanged from a module acquiring frames from a webcam to one receiving a video feed over ethernet, in a transparent way with respect to the rest of the system. Moreover, a module containing a median filter can be replaced by a module that performs a Gauss filter[1], thus affecting the behavior of the overall computation. In this way the system can adapt its pipeline in order to react to changes such as brightness floating in the input stream, increased quantity of noise in the images and so on. A difference based flow for partial dynamic reconfiguration is often not feasible in this case, because of the different implementations on the processing elements of the pipeline: the Gauss and the median filters have different implementations and they cannot be synthesized in a single, customizable IP-Core.

---

[1]Median and Gauss filters are two common operations on a block of pixel data in an image processing application, employed to reduce image noise.

2.2.5 Other applications

The four applications analyzed so far take advantage of the features of partial dynamic reconfiguration introduced in section 2.1, namely the improved flexibility given by dynamically changing hardware modules and a more efficient use of the area of the reconfigurable resource. In the latter case, however, it must be pointed out that often an external total dynamic reconfiguration scheme is preferred over the partial dynamic one, because the whole area of the FPGA is used for the implementation of a portion of the design and then reconfigured when that portion has finished its computation. Unless the application has the stringent requirement of being implemented as a SoC, the most straightforward choice is that of an external processor controlling the reconfiguration of the whole FPGA. Examples of this approach can be found in pattern matching and text searching applications, bioinformatics applications, evolutionary image processing and others. In these cases the hardware flexibility of the FPGA is exploited to quickly process large quantities of data against a relatively small set of reference inputs. These inputs are often directly 'folded' into the hardware design for maximizing speedups, and the reconfiguration is used to partition these inputs into hardware modules that are actually deployable on to the chip.

For example, in a pattern matching application, the set of words to count in the input document(s) can be translated into a finite state automaton that takes advantage of the common prefixes of the words for a more efficient implementation. The transition table of the automaton is translated into a hardware design. In this case the transition table could be too large to be implemented on the FPGA, so it can be partitioned in order to reconfigure the FPGA with the

different subsections of the design. Often automated tools are provided for the HDL generation of the different partitions.

For a thorough list of application fields, the issues arising from each particular topic and for a deeper understanding of how reconfigurable computing is suited to each of these fields the reader is referred to (Bobda, 2007),(Gokhale et al., 2005) and (Voros and Masselos, 2005).

CHAPTER 3

DEVICE CHARACTERIZATION

In this chapter the basic concepts and principles behind this work will be explored. An analysis of the current methodologies for partial dynamic reconfiguration will be performed, the FPGAs used in this context will be introduced, along with their relevant architectural details. The last part of the chapter focuses on the characterization of the UCF file that allows for constraints to be specified in order to realize the floorplanning of the partial dynamic reconfigurable system.

3.1 FPGA families and models

The FPGAs used in this work belong to three distinct families, with different architectures and features. Table VII lists the FPGA model(s) used to explore each of the three families used in this work. Since this work will mostly deal with low-level details of the architecture

Family	Model
Spartan 3 (Xilinx Inc., 2007a)	XCS200
Virtex II Pro (Xilinx Inc., 2003)	XCVP7 XCVP20
Virtex 4 (Xilinx Inc., 2007d)	XC4VFX12

TABLE VII

FPGA FAMILIES, MODELS AND REFERENCE MANUALS USED IN THIS WORK

of the aforementioned FPGA chips, the following subsections will be devoted to illustrate the architecture of the devices in table VII, by showing an internal representation of the resources of each model. As pointed out in section 1.2, the general structure of the internals of an FPGA is a sea of logic gates, grouped in slices and CLBs, into which heterogeneous hard-cores are embedded. This structure will be reflected in the following architectural description of the four models.

3.1.1 Spartan 3

Spartan 3 family is the simplest of the three, and it offers fairly low cost devices. The available resources besides CLBs, block ram, and IOBs are dedicated 18 bit X 18 bit hardware multipliers and digital clock managers (DCM) hard cores. In particular the chosen model, **XC3S200**, has a CLB array composed of 24 rows by 20 columns, interleaved by two columns in which reside 216 Kbits of block ram and 12 hardware multipliers. Four digital clock manager hard-cores are placed on the perimeter of the device in place of IOBs in the position corresponding to BRAM columns.

An internal representation of the device is illustrated in Figure 12, where the resources that break the regular structure of the slice array have been highlighted in dark gray. Configuration of the resources is performed on a columnar basis, i.e. a subset of slices in a vertical column cannot be addressed for reconfiguration without configuring the remainder of the column as well.

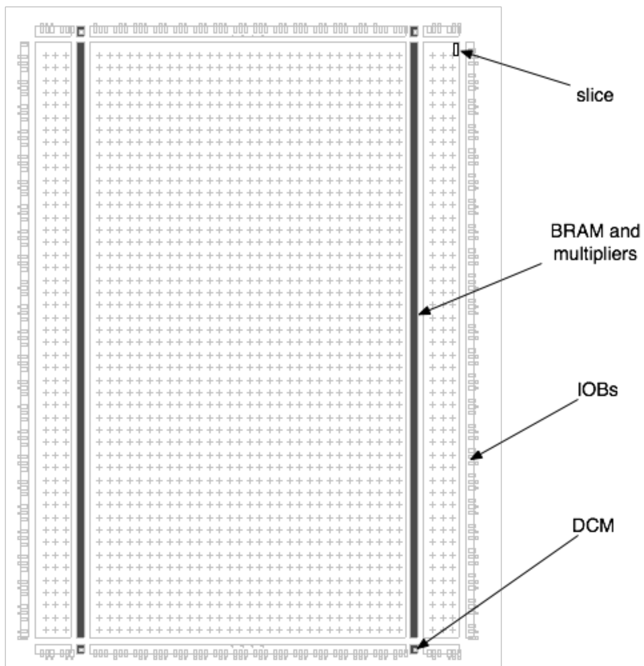

Figure 12. XCS200 internals as shown in XIlinx Floorplanner

3.1.2 Virtex II Pro

Virtex II Pro architecture provides more resources and an improved logic count with respect to the previous family. The two models of this family that have been studied in this work, **XCVP7** and **XCVP20** have hard-core processor(s) on board, in order to exploit the speed and flexibility offered by a general purpose processor in standalone designs. In this way the performance in software execution of an external processor can be achieved directly within the same die of the FPGA. The processor model is the Power-PC 405.

A representation of the internals of the two models is given in Figure 13, with resources other

than CLBs and IOBs highlited in dark gray as before. The available resources are an array of 4,928 and 9,280 slices, 44 and 88 for both 18Kb BRAMS and 18 x 18 bit multipliers and 4 and 8 DCMs for the two devices respectively. Configuration is still performed on a per-column basis as in the Spartan 3 family.

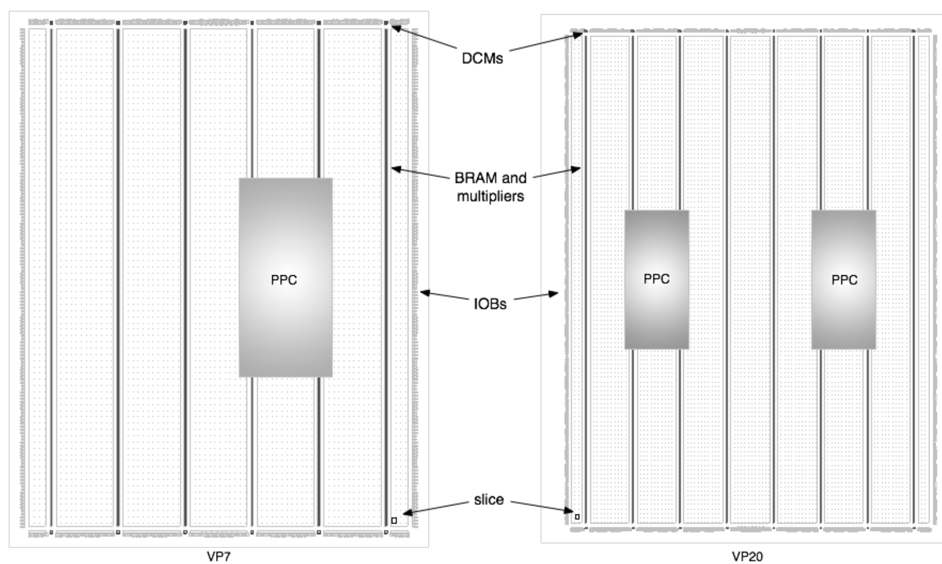

Figure 13. XCVP7 (left) and XCVP20 (right) internals as shown in XIlinx Floorplanner

3.1.3 Virtex 4

The Virtex 4 family is the most recent of the families taken into consideration. The most important feature regarding dynamic reconfiguration is its capability to be configured without the columnar constraint. Configuration frames are now provided with a row address, that

points to a specific row of resources on the device. The single row of CLBs is not addressable individually, as the rows addressed by the frames are groups of CLB rows. This fact, however, makes a 2D reconfiguration scheme feasible. The quantity of hardware embedded on the silicon die has been improved, giving way to an even more heterogeneous array. The number of CLBs of the **XC4VFX12** is 64 by 24. The device internals are shown in Figure 14.

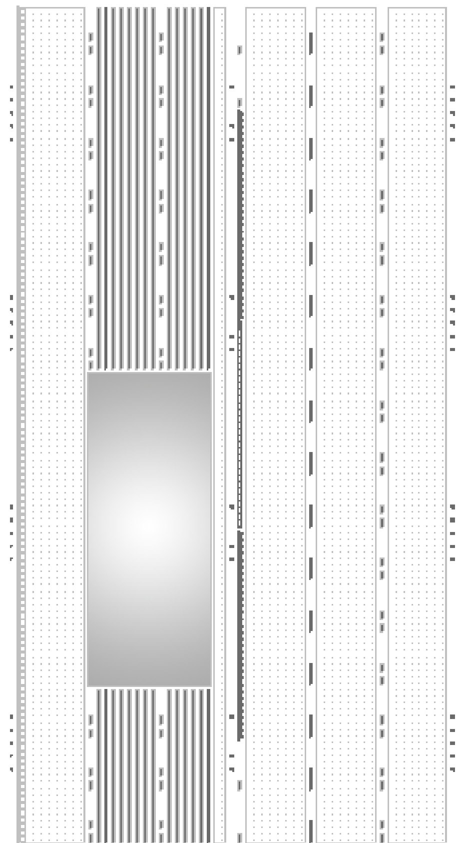

Figure 14. XCVFX12 internals as shown in Xilinx Floorplanner

3.2 Configuration conventions and file formats

The following subsections will illustrate the conventions used by the vendor in numbering and addressing the resources on the array. These conventions are used extensively throughout the ISE [1] tools and some understanding of them is needed in order to correctly follow the steps in the flows for partial reconfiguration of the devices, namely for constraining pieces of logic inside particular regions. Both of the presented numbering schemes are visible by accessing the representation of a FPGA in FPGA editor, a tool described in section 3.3.2.

The two file types on which the analysis of this work has been performed will then be introduced.

3.2.1 Configuration resources numbering scheme

The heterogeneous nature of the FPGA array requires the adoption of a numbering scheme to address the single resources in design tools. The model chosen by Xilinx is that of a separate numbering scheme for each resource category. Therefore there will be distinct numbering schemes for CLBs, BRAMs, multipliers and so on.

The process of numbering the resources is equal for all the types and it is location based: the resource with coordinates X0Y0 is always the one in the lower left corner of the device. The numbering of the subsequent resources in the array is determined just like if they were points in a bi-dimensional cartesian coordinate system. Having the resources numbered separately according to their type permits the correct identification of a particular element inside the set of elements of the same type, but this system does not allow the identification of the location

---

[1] Integrated Software Environment, the set of software tools developed by Xilinx for going from an HDL description of the hardware to the bitstream(s) to be configured on the FPGA device.

of a particular resource on the device unless the exact architectural realization of the array is known.

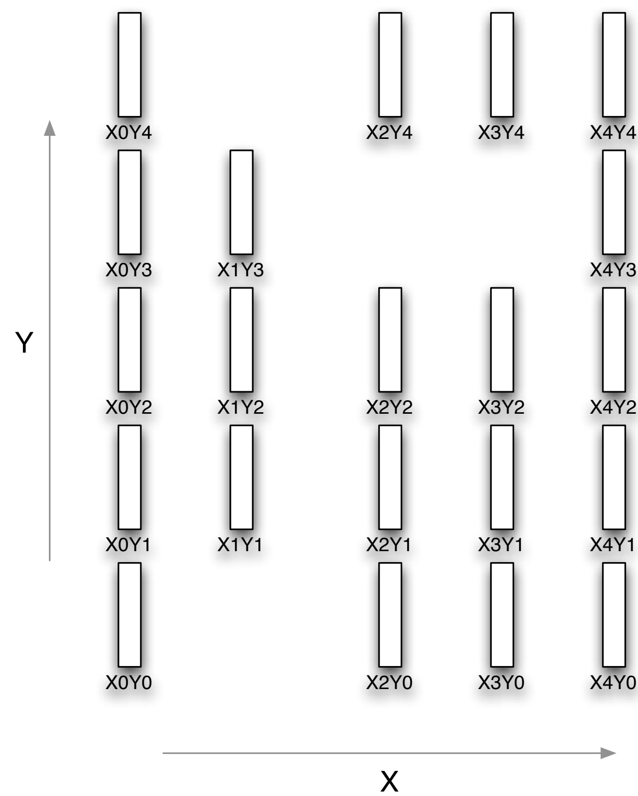

Figure 15. An example of the resource numbering system used to refer to resources across Xilinx devices.

An example of this numbering scheme is shown in Figure 15 for a generic type of resource. As the image shows, the first resource of the numbering scheme is placed in the lowest left corner,

while the coordinates of other resources are incremented going to the right for X coordinates and up for Y coordinates. The maximum value of X is determined by counting the resources in each row and taking the maximum minus one. Analogously the maximum Y coordinate is given by taking the maximum among the column element numbers and subtracting one. Another important thing to notice from Figure 15 is that 'gaps' are allowed in the numbering system, so that, in the example, there will be no such resource as X2Y3 or X3Y3. In place of gaps there could be either another configurable resource, or another, non-configurable FPGA element.

3.2.2 The RPM grid

In addition to the normal numbering scheme described in section 3.2.1, a global coordinate system has been envisioned by the device manufacturer: in (Wade, 2002) this system is exposed as a reference for the realization of heterogeneous hardware macros. Hardware macros are pre-routed and pre-mapped hardware functionalities that can be included in a design. For the placement of the macros on the FPGA, the relative position of a resource with respect to another, independently of the resource type, must be known. For this reason a *global* numbering scheme has been introduced, called the RPM grid [1]. In essence the RPM grid is an array of positions that can be occupied by a particular resource. The position slots of this array are numbered in the exact same way as the numbering of single resource types, so that the exact resource occupation of a relatively placed hardware macro can be known in all of the resource types, and the relative positioning of those resources can be known. It is thus possible to place

---

[1] RPM stands for Relatively Placed Macro

the macro in a new position on the FPGA, given that that position has the same features of the original position where the macro has been developed, namely resources and inter-resource relative placements. This feature of the RPM grid will be useful during a partial bitstream relocation, where it is important to know both the resources that the bitstream configures and their relative placement on the FPGA die.

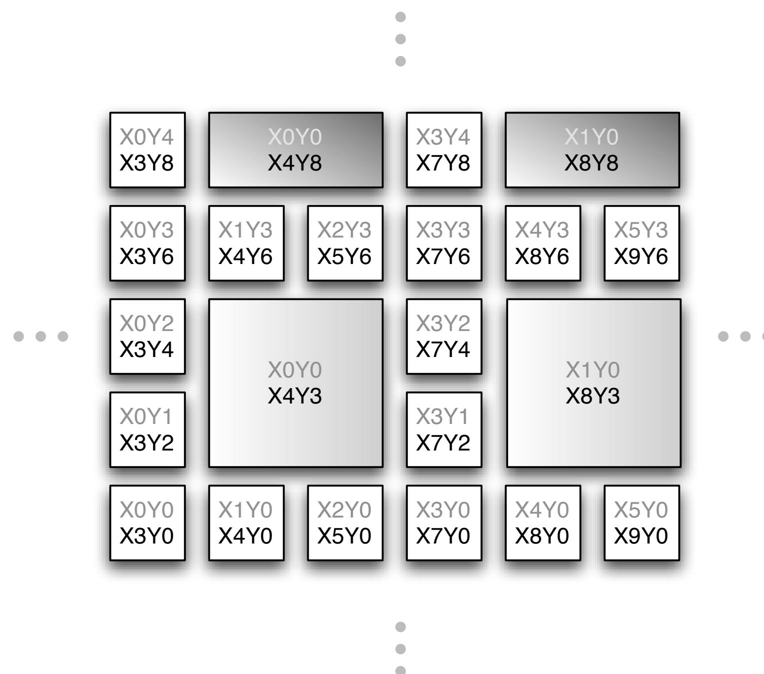

Figure 16. An example of the numbering style adopted in the RPM grid.

Figure 16 shows a small example which illustrates the pairing of the conventional numbering system described in section 3.2.1 and the RPM numbering system described here. Different box sizes represent different kinds of resources, as in real devices where resources can be visually identified as belonging to a particular class according to their shape. In each box, the first line of text represents the coordinates in the conventional numbering system, while the second line has the coordinates of a sample RPM system.

The point of reference of a resource in the RPM grid is its lower left corner, and the main constraint is that if the horizontal and/or vertical position of this point in the global visual representation is smaller than the position of the lowest left corner of another arbitrary resource, the former resource will have an RPM X coordinate and/or an RPM Y coordinate smaller than the latter.

Increments in RPM X or Y coordinates are not always equal to one, as in the conventional numbering system, but can be other greater integer numbers, provided that the positional order of resources is respected as before.

3.2.3 UCF file format

The UCF file format is used by the Xilinx toolchain to express some constraints on the design in order to guide the processes of placement, routing and mapping of the HDL descriptions of the soft-cores given by the developer. What is relevant for the analysis of the dynamic reconfiguration methodologies is the AREA GROUP class of constraints, which allows the definition of a RR on the FPGA. The PAR (Place and Route) program will then read these constraints and reduce the configuration resource space accordingly.

In the following excerpt from one of such UCF files a constraint about a reconfigurable region (or partial reconfigurable module, according to the terminology used by Xilinx) is defined.

```
INST "addsub_B" AREA_GROUP = "AG_PRMB";
AREA_GROUP "AG_PRMB" RANGE = SLICE_X28Y72:SLICE_X41Y127;
AREA_GROUP "AG_PRMB" RANGE = DSP48_X0Y26:DSP48_X0Y27;
AREA_GROUP "AG_PRMB" MODE  = RECONFIG;
```

The first line associates a component instance to a reconfigurable region. Then the physical constraints of the resources comprised in that particular area are given, using the numbering scheme explained in 3.2.1. The second line defines the space of the region in the slice space, while the third defines the constraint in the space of the DSP48 resources. The last line eventually states that the region defined will be affected by the reconfiguration process.

3.3 Available tools

3.3.1 ISE - Integrated Software Environment

ISE is a suite of programs given by Xilinx for the development of hardware modules starting from their description in an HDL language, and constitutes the essential software to develop custom hardware cores with Xilinx FPGAs. It is built around a series of editors - the main one a text editor for modifying the project files - a project navigation tree and a console that reports the results of the programs called by the designer on the project files. With ISE it is possible to apply the module design methodology that allows different people to work at the components of a design separately and integrate them in a top project, working hierarchically.

Starting from the HDL code of a module is then possible to synthesize it, that is translating the specification of the hardware functionalities into a net composed of logic ports and interconnections. This is the basic step in validating the HDL code and determining if it is one of the most computationally intensive tasks performed by the program.
The phases that follow the synthesis transform the logic net generated by the synthesis into a mapping on the resources of the FPGA. The mapped resources are then placed and routed on the physical array of the FPGA by the Place and Route (PAR) programs. This results in a netlist [1] annotated with the results of the PAR programs, that can be eventually translated into the configuration bitstream for the chosen device by the *bitgen* program.

3.3.2 FPGA editor

FPGA Editor is a program comprised in the ISE program suite, which offers a window into the internal architecture of a Xilinx FPGA. It takes as input an .ncd file of a synthesized hardware design and displays the configuration of the FPGA resources on a graphical representation of the chip. Besides being used for visualization or debugging purposes, the FPGA editor can show the configuration and allow small alterations to the FPGA resources. For example, the designer can alter the equations of a LUT of a particular slice, or change a particular routing between resources. A sample of a placed and routed module shown in FPGA editor is shown in figure 17. This tool is used also to have an idea of what is the occupation of the hardware module on the FPGA available area. FPGA editor is a tool that allows to refer to the

[1] The netlist used prior and after the execution of the PAR programs is contained in an .ncd file

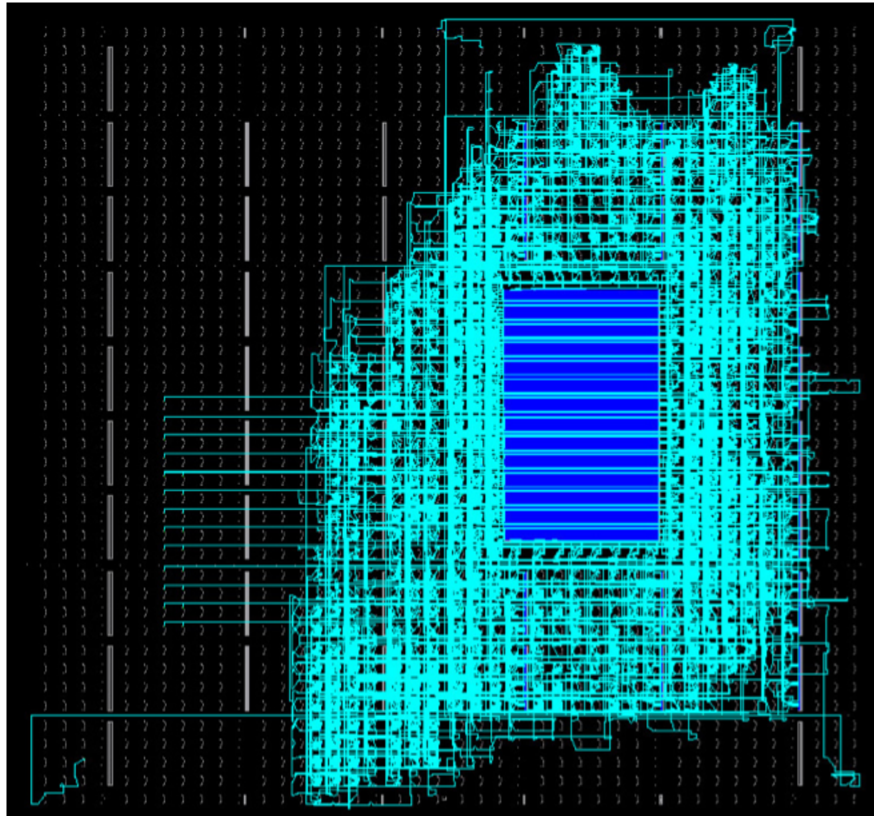

Figure 17. A screenshot of a placed and routed module in FPGA editor.

internal architectural details of a particular FPGA model, by gathering information on each particular resource of the array and inspect the configuration of each resource of the FPGA. As Figure 18 shows, upon selection of a particular resource, its description is output on the program log. Moreover, Figure 18 shows a CLB that has been modified starting from a blank design file in order to modify its LUT functions. The currently selected cell in the figure is the block pointed by the arrow, while the slice with the LUT modifications is the one directly

beneath the pointed slice. Around the slices there are the switch matrices of the CLB and all the interconnections that support the FPGA array.

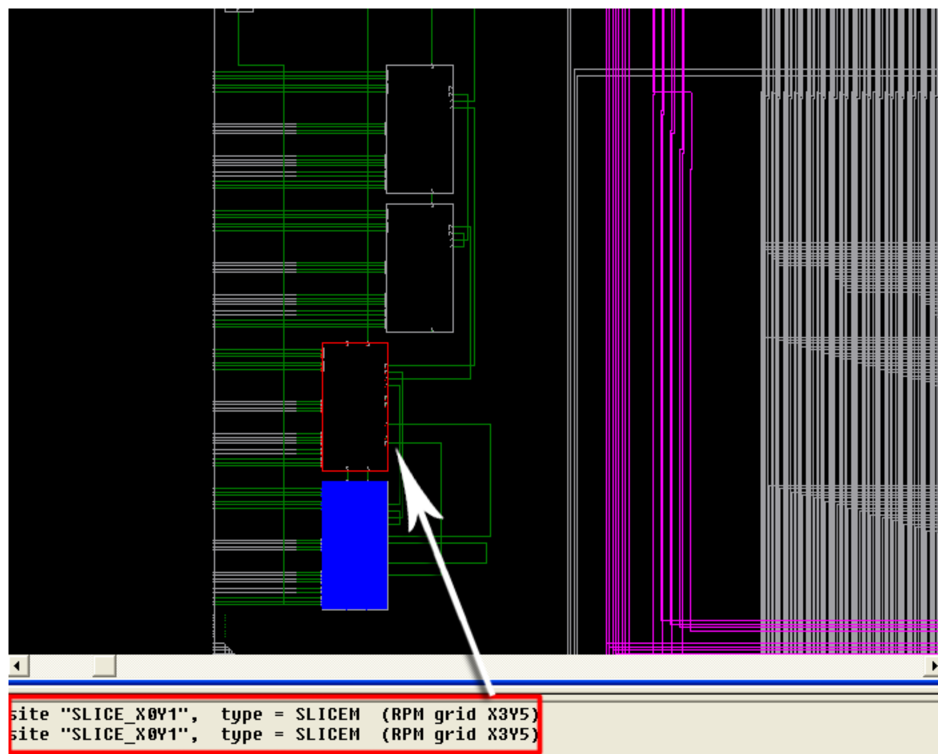

Figure 18. A slice block as shown in FPGA editor. The description of the selected resource is outlined in red in the console of the program.

3.3.3 Floorplanner

Floorplanner is another addition to the ISE suite of programs. As FPGA editor, it can open .ncd files that contain a placed and routed module, and show the contents of this file over a

logical FPGA representation. It then allows the user to refine the area constraints that have been supplied at the beginning to the Place and Route programs and change them according to the results visualized in the editor. In this sense Floorplanner is a graphical constraints editor for refining the content of a UCF file that drives the place and route process. The idea is to have multiple iterations in which the placement and routing of a particular module or design is refined in order to have an end result that is more efficient in the utilization of the resources of the FPGA.

Unfortunately, this tool is not aware of the issues involved in designing a module for a PDR system, and the designer still has to focus on the particular constraints of the chosen flow in order to create consistent area constraints. Floorplanner thus allows the creation of area group constraints that include the necessary resources, other than slices, in the constraint definition, but in the editing of these constraints the user is free to follow any policy, and there is no enforcement of the partial dynamic reconfiguration guidelines in the program, that is Floorplanner is suitable for a modular design static implementation of an architecture on FPGA, but not of particular guidance when this module based design is used to implement a PDR architecture.

3.3.4 PlanAhead

Planahead is an optional tool in the Xilinx development suite that implements a hierarchical floorplanner, that is a program that is able to floorplan different modules of the design by taking into account the hierarchy designed in the standard module based approach. Since this hierarchy can have multiple levels of containment, the floorplanner is able to work at each one of these

levels, changing the constraints for each of the sub-modules that compose a module up in the hierarchy. It allows the optimization of metrics such as the timings of the modules as their position change, the area utilization and the length of interconnections with IOBs. To optimize these metrics, the redefinition of the area constraints is applied to each module and a brute-force algorithm is employed, evaluating the desired metrics every time. The running time of this approach can be of days for complex hierarchies. As with the case of Floorplanner, however, the program has been devised for static designs based on the modular approach and very little is taken into account for the definition of area groups that respect the partial reconfiguration guidelines. Moreover, the software operates at the slice level, not taking into account the resources that are encompassed in a slice area definition, possibly leading to mistakes in a PDR environment, where the Place and Route tools can select to reconfigure a part of the FPGA outside the area constraint defined on the slices, causing the reconfiguration of a part of the architecture not meant to be reconfigured.

3.3.5 Jbits

Besides the tools comprised in the ISE tool suite, Xilinx also released Jbits (Guccione et al., 1998), a Java API which gave an insight into the bitstream configuration files for the Vitrtex family of devices. The purpose of Jbits was to give a low level API to every configuration information stored in the bitstream file, namely every configurable aspect of a CLB had the opportunity to be set or probed by using the appropriate class. In this way custom hardware could be generated by programs other than the ISE tools, or the bitstreams produced with ISE could be modified to change a particular functionality, doing from Java program what is now

done with FPGA editor visually and by hand.

For example, the FPGA editor ability of changing a particular LUT equation in a slice of a CLB would be replaced by the following java code:

```
set(row, col, Slice0_FLUT, XOR);
```

In this case the F lookup table of the slice 0 in the row and column positions specified is changed to provide a XOR function. The corresponding methods that read and return the configuration from the bitstream have also been added by the authors of Jbits, and in this sense the configuration bitstream of Virtex devices has been given a means for analysis. On the top of the layer providing small bits modifications, a custom core library layer was built that could create more complex hardware building blocks with a certain level of customization, thus allowing programs exploiting Jbits to leverage on a higher level of abstraction rather than having to deal exclusively with the low level details of the FPGA architecture.

Unfortunately the Jbits project seems discontinued by Xilinx, and it is no use for inspecting the configuration bitstreams anymore. It would have been of great use having such a tool for the development of systems such as the framework proposed in this work, because the meaning of the configuration words in the bitstream format has not yet been disclosed by Xilinx, forcing the researches to reverse engineer the bitstream format in order to gain an insight of the meaning of its contents.

It is worth mentioning that recently an open source effort to provide an API similar to what Jbits once did is currently under development (Note and Rannaud, 2007). The effort in recovering the configuration information from the bitstream has yielded for now a way to recover the

netlist of the design encoded in the bitstream itself, and hopefully the support for more devices and families will be added in the future, thus allowing the development of tools that can be independent of the Xilinx provided software and implement novel methodologies and flows, even for partial dynamic reconfiguration of Xilinx FPGAs.

CHAPTER 4

METHODOLOGY

This chapter illustrates the motivations that are behind the development of the Rebit framework, created in this master's thesis work, discussing why there has been the need of a tool for the debugging and validation of Partial Dynamic Reconfiguration architectures. The first four sections of the chapter are thus devoted to explain the possible issues or limitations that are involved in the conventional development of a PDR system, while the last part is devoted to illustrate the proposed approach for answering the points made at the beginning of the chapter: it discusses the solutions that have been thought to ease the development and the validation of a PDR system on FPGA, the data on which these solutions are based and the overall organization of the proposed framework, leaving the implementation details to the next chapter.

4.1 The Earendil flow

The framework proposed in this work is part of a much larger one described in (Rana et al., 2007), the *Earendil* framework, developed in our lab at Politecnico di Milano.
The scope of the Earendil framework is to provide a complete flow and model for the definition of applications that exploit the reconfiguration capabilities of FPGAs. The idea behind the proposed methodology is based on the assumption that it is desirable to implement a flow that can output a set of configuration bitstreams used to configure and, if necessary, partially reconfigure a standard FPGA to realize the desired system. A complete model for the application has thus

been envisioned, to be used in every phase of the flow, namely the *High Level Reconfiguration*, *Validation* and *Low Level Reconfiguration* phases.

The first phase - HLR - aims at partitioning the application in sub-modules and resolving the problem of hardware-software codesign in an efficient way, that is to decide which portions of the application will be implemented in SW and which ones as HW modules, taking into consideration reconfiguration capabilities.

The second phase - VAL - drives the refinement cycle of the system design, by altering the decisions taken in the first step of the flow.

The last phase - LLR - is targeted to the automatic generation of a low level implementation of the physical solution, eventually resulting in the configuration bitstreams targeted to a device of choice.

A representation of the Earendil flow is given in figure 19, where these three different phases are highlighted in different colors. By applying this flow to the realization of a particular

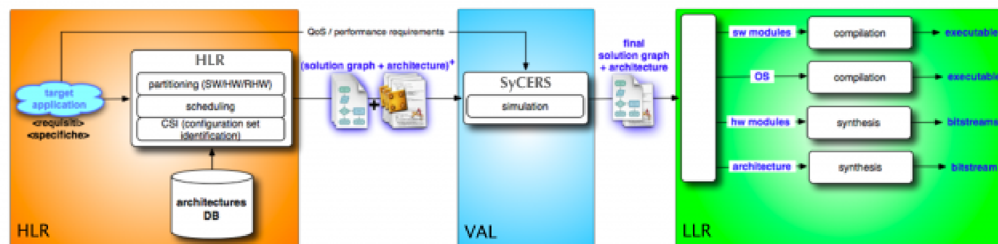

Figure 19. Earendil flow overview

application taking advantage of reconfigurable hardware, the developer is guided from the very first phases of application definition on to the eventual configuration files and the executable files to be used in the overall resulting system.

The Earendil flow necessarily relies upon the execution of a *Bitstream Generation*[1] phase provided by the software of the FPGA vendor, and thus a further analysis of the produced bitstreams is necessary to fully debug - at configuration level - the results of the flow.

The standard methodology in this case is to manually inspect the produced files in order to look for possible issues that would prevent the system to work correctly, or to test the system directly on a FPGA board and figure out what the possible errors could be, if any. Certainly an automated way to check the configuration files for possible problems would be of great support for the final deployment of the produced system.

Moreover, a global vision of the low level configuration architecture produced by the flow, together with the high level area constraints produced during the system development would replace the long process of manual aggregation of the configuration data and the definition of the Reconfigurable Regions, done in order to understand the produced system and for its debugging.

Finally, the relocation of the RFUs can be taken into consideration even at the configuration level, since it is possible to relocate partial configuration files with automatic tools such as BanMat Light (Corbetta et al., 2005 2006). However, the position of relocation must be calculated

[1]the bitstream generation takes place at the end of the 'synthesis' phases in Figure 19

by hand and it must then again be validated against the RR constraints.

All of these reasons, further explored in the following sections, gave origin to the present work, in which a tool for the analysis of the configuration files of PDR architectures has been produced to ease the aforementioned phases at the end of system design.

Rebit, the name of the developed framework, thus is placed at the end of the Earendil chain illustrated in Figure 19, as the debugger of the produced configuration files.

Since the Rebit framework answers the issues of debugging generic bitstreams and bases its analysis on data that is common to every partial reconfigurable system, it is independent of the overall execution of the flow and it can be used in the debugging and validation of any system that makes use of PDR techniques.

In this sense the proposed framework can be seen either as the final debugging step at the end of the Earendil flow, or as a standalone framework for the analysis, debugging and validation of reconfigurable systems. The detailed explanation of the motivations that gave origin to the framework, in the light of a generic reconfigurable application, are given in the following sections.

4.2 Application issues

As pointed out in section 1.6, the application of PDR techniques in developing an arbitrary application is not straightforward and requires a careful analysis of the application domain and of the proposed implementation in the light of the constraints and methodologies involved in the design of such a system. The already cited flows for partial reconfiguration proposed by Xilinx in (Xilinx Inc., 2006) and (Xilinx Inc., 2004) address some of the existing problems by giving a

set of guidelines, but the application of these guidelines and constraints is still a task reserved to the system designer, who has to merge the requirements of a functional PDR architecture with the constraints and issues arising from the chosen application domain.

Two key aspects in particular are to be taken into consideration during the development of such a system:

- PDR constraint satisfaction.
- The interpretation of the given constraints by the PAR programs.

These topics will be discussed in the following subsections.

4.2.1 PDR constraint satisfaction

The constraints exposed in the flows for partial reconfiguration given by the vendor must always be respected for the successful realization of the system. For example the constraints regarding RR width and RR placement must be respected when the reconfigurable regions entries are defined in the UCF file, and the generated bitstreams for each of the RFUs must comply to the specification of their corresponding Reconfigurable Region.

These constraints must be entered by hand into the UCF file, and involve a deep architectural knowledge of the particular FPGA model employed for the system, often making the designer resort to the internal representation of the device in programs such as Floorplanner or FPGA editor, in order to gather the information about the definition of the reconfigurable region with respect to the constraints of the PDR flows.

In this case an automation in the verification of errors in the definition of these constraints,

validated both with respect to the particular FPGA architecture and with respect to the guidelines of the flows for partial reconfiguration would be a significant contribution to the manual development process.

The effect of a mistake in the definition of the RRs would not be noticed immediately, but the Xilinx programs, in particular the PAR program, would operate on erroneous information, producing a system that would not be functional at all. If for example two reconfigurable regions overlap with each other, the generated partial bitstreams would conflict with each other when configured on the device, with one overwriting the other in the region where the overlap takes place. Moreover, if all of the resources on the FPGA would not have been included in their respective RR, the synthesis of the modules of the system would try to use these resources for the realization of other functionalities, i.e. for the static part of the architecture, leading to a failure when the partial bitstreams are configured in a RR not completely specified.

These errors are not easily recognizable in the design phase, and often their dangerous effect only comes up when the system is downloaded and tested on the actual device. It is thus important to have debug functionalities to assist the developer in the UCF editing, trying to make this task less error prone and more automated.

4.2.2 PAR programs constraint interpretation

Even if the constraints for the Reconfigurable Regions are defined correctly, the PAR tools can violate them during the synthesis of a particular RFU, thus giving way to partial bitstreams that actually exceed the boundaries of the RR area in which they are configured. In this way the efforts placed in the definition of such areas are nullified by the violation of the area constraint.

Moreover such an error is not easily discoverable, since no warning about constraint violation is given by the vendor software. These kinds of errors are therefore unpredictable and are caused by the non-deterministic nature of the Place and Route tools: it is in fact impossible to know a priori the exact boundaries of the synthesized hardware unless executing the PAR algorithms themselves. If there is the attempt to place too much logic into a particular RFU, the constraints of the RR will necessarily be violated. Until now the available methods of checking the results of the PAR programs was to open the generated files into the FPGA Editor, and visually inspect the design for possible violations, without the possibility of having the constraints defined in the UCF file displayed in the program for easy reference. Figure 20 gives a representation of the error type that can arise from the violation of the area constraint by the PAR programs. Even if the Reconfigurable Region has been defined with correct bounds, the logic allocated during the Place and Route algorithms overflows these bounds, going into a region that is not destined to that particular RR, but either to another RR or to the static part of the design. This situation is very dangerous and usually leads to faulty systems: given that the atomic reconfiguration unit is the frame, when the RFU will be downloaded on the device it will overwrite the configuration of the two frames corresponding to the logic mistakenly allocated during the Place and Route, seriously compromising the functionalities of the overwritten portions, first of all the communication facilities that straddle the boundaries of reconfigurable regions.

It must be pointed out that the reason why the PAR tools do not warn the user about these issues is that those programs have been developed for the production of static designs, where

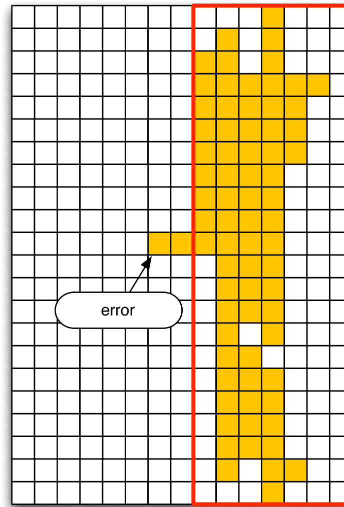

Figure 20. A representation of a possible error caused by the PAR process. The reconfigurable region is defined by the red outline, the allocated logic is pictured in yellow.

the reconfiguration is not going to take place at runtime. In this situation the area constraints given by the user are only a hint of where to realize a determined functionality, and are not mandatory, because the overall system is configured as a whole, without overwriting any of its portions.

It is thus clear how these issues are critic for the success of the design of a reconfigurable system in general and particularly relevant in the realization of a partial reconfigurable one. All of the applications presented in chapter 2 have probably been forced to cope with these issues and in general they lose flexibility in terms of the easiness of expanding their functionalities,

because every time that a RFU must be added to the overall system, all of the aforementioned problems must be taken again into consideration.

For example, if a new controller module must be added to an adaptive control application, the designers must not only concentrate on the algorithm that implements and the subsequent HDL coding, but must also respect the constraints involved in the overall system, having to manually retrieve and check them against their design and its actual synthesized RFU.

Moreover, even in an automated flow as Earendil these problems are still present due to the non-deterministic nature of the PAR tools.

4.3 Exploring bitstream relocation

Besides the verification of the constraints satisfaction, it would also be amenable to be able to explore different design solutions, even after the bitstreams have been produced.

This is made possible by partial bitstream relocation, a technique that allows changing the physical placement of the resources configured by a particular bitstream after its synthesis.

The usefulness of bitstream relocation is found when the system designer wants to alter the scheduling of the RFUs of the PDR design:if a particular functionality has been initially synthesized to be placed in a certain RFU, and it is needed in another point of the application life cycle where the RR initially assigned to it is occupied by another RFU, it is possible to relocate the first functionality in another RR to have both RFUs functioning at the same time. Figure 21 gives an exemplification of this situation: RFU 1 and 2 are configured at the same time on the chip. In the next configuration RFU 3 takes the place of RFU 1, and if RFU 1 is to be

configured on the device, it must be first relocated.

This process causes two problems: the first is that there could not be a RR available to hold

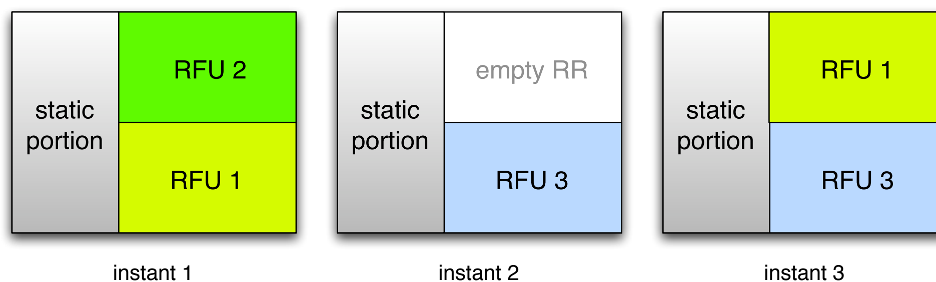

Figure 21. Three instants in an application life cycle. RFU 1 must be relocated to be configured together with RFU 3 in the third instant.

the RFU in another place, because the occupation of the RFU violates the area constraint of the target RR; the second is that there might be a free RR, but it might not contain the configurable resources necessary for the RFU configuration. This last issue is exemplified in Figure 22. The actual RFU position is in the top left corner of the device, and configures the core in its center. The possible positions for this RFU are thus the ones that guarantee keeping the relative positioning between the configured resources in the first place. If the bitstream for that particular RFU would e relocated in the middle of the device, its functionalities would be lost, thus it is important to consider this issue duiring a relocation.

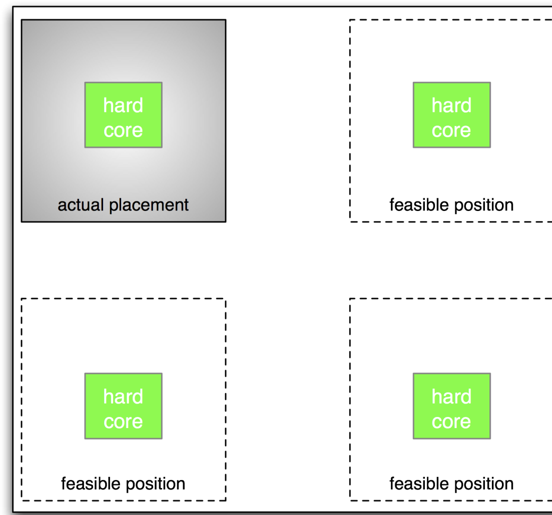

Figure 22. Four possible placements of a RFU considering the position of reconfigurable resources.

The relocation task would be easier if the feasible positions for relocation would be known, with the configuration of the device resources taken into account automatically.

4.4 Area constraint modification

The last feature that is lacking in the current methodologies for the design of PDR systems is the automatic editing of the UCF area constraints. As stated before, these constraints drive the PAR programs in the creation of the routed logic that will eventually be configured on the device. These constraints, as seen in chapter 3 are composed by multiple lines, stating what is the area constraint for each type of resource on the FPGA. As for now the main constraint on the SLICE resource type must be integrated with other lines that state the constraint for the

resources contained in the area defined by the slices on the FPGA array, and the integration is done manually by looking up the other resource indexes on the FPGA representation.

It would thus be an aid to the designer to be able to edit these constraints visually on an FPGA representation.

4.5 The proposed approach

The present work tries to give an answer to the aforementioned issues, by providing an integrated environment that is essentially used for the analysis of the bitstream files involved in the realization of a PDR system, and results in a software tool, Rebit, that tries to ease the system designer efforts to check their configuration files after the output of the chosen flow and debugs possible errors *before* the actual configuration or reconfiguration of the FPGA takes place. The following subsections are thus devoted to illustrate the approach used in answering to the issues presented before, by describing the data model on which the framework operates and the three components that have been envisioned to realize the program implementation.

4.5.1 Input data

As stated before, the input of the present work is constituted by the bitstream files that compose the final result of any of the flows for Partial Dynamic Reconfiguration. In addition to these configuration files, some other design information is needed to successfully operate, namely the UCF file describing the RRs of the system and the information about which RFUs will be configured on the device at a given moment in time. This last information is usually given by the scheduling phase of a PDR flow, and, in the Earendil terminology, is referred to as the set of the *Static Photos* of the system.

In addition to this information, the framework must also have a precise knowledge of the underlying FPGA for which the architecture has been synthesized, in order to correctly interpret the data available in the bitstream files. The following list summarizes the project design input information and explains the information that can be gathered by parsing the relative files.

Bitstream files: the model of FPGA used in the system and the frames that are configured by each particular bitstream.

UCF file: the constraints set by the user to define the RRs of the architecture and the location of the communication bus macros.

Static Photo Description: all of the possible combinations of RFUs that the system will configure at the same time.

From this data and an architecture description it can be possible to automatically check the constraints defined in the Partial Dynamic Reconfiguration Flows and thus relieve the system developer from doing so.

The architecture description available to the reasoner that performs all of these checks is instead independent of a particular design and includes both an architectural description for each of the FPGA models introduced in chapter 3, and the information needed to parse the bitstream files, such as the IDCODE word that univocally identifies a particular device or the length of a configuration frame.

In particular the description of a specific FPGA model is given in terms of its associated RPM grid, as described in section 3.2.2, which contains both the overall relative positioning among the resources and the numbering of these resources as the designer must refer to in writing

the UCF file. The RPM grid is thus the fundamental component for performing all of the architecture-related checks and reasoning, because it gives an adequately detailed low level description of a particular FPGA model and of its internal structure.

4.5.2 The overall system

Figure 23 illustrates the overall structure of the proposed framework. The input data

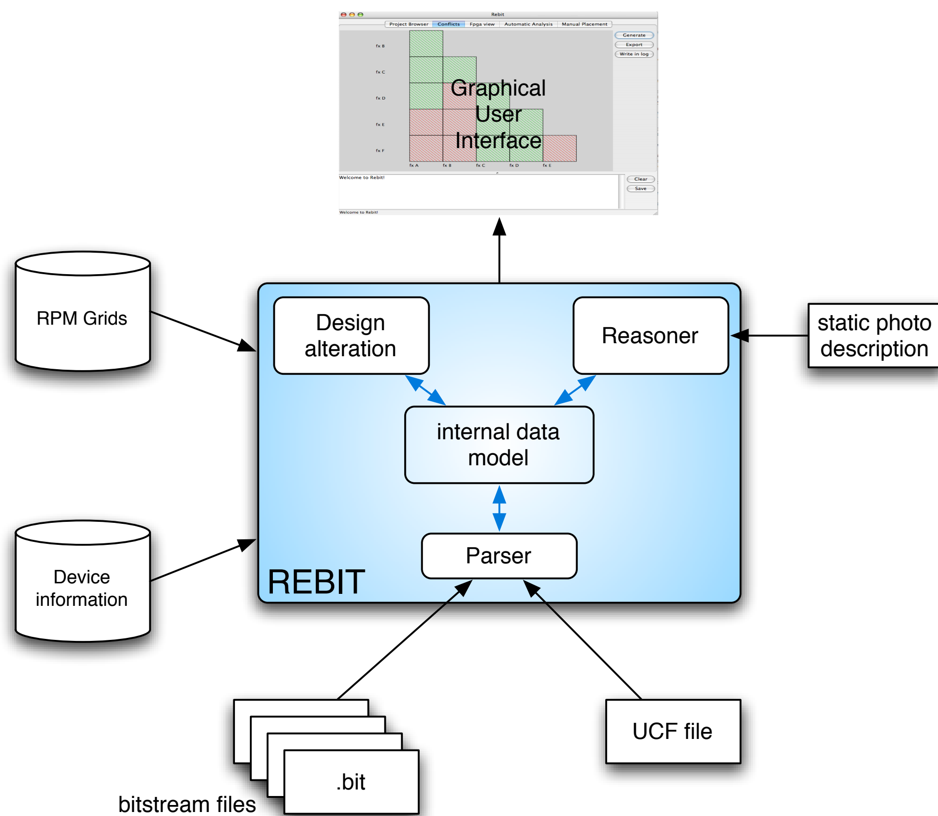

Figure 23. The overall logic structure of the Rebit framework

that are given to the application are first stored in the framework and are available to all the other components of the system. The structure of the framework is then logically split into three different modules, the *parser*, the *reasoner* and the *design alteration* module explained in the following paragraphs. Each module has the ability to work on the internal data representation of the framework, by gathering data, modifying data or by performing checks and displaying the information.

Parser module

A first portion of the proposed approach is a parser module, which is in charge of parsing the input files of the project and gathering all of the relevant data needed by the other two components.
The bitstreams added to the project are immediately parsed by this module and their area occupation is gathered, if the parser of the bitstream did not end in an error state. While parsing the bitstream files, their validation is performed by simulating the behavior of the configuration logic internal to the FPGA. This prevents errors in these files due to alterations in data transfer or errors introduced by the application of other tools in different programs such as bitstream manipulators. This validation phase therefore ensures having correct files to configure the device, while gathering the area occupation data of partial configuration bitstreams.

Reasoner module

The reasoner section of the approach is then the part in charge of applying the checks to the configuration bitstream data and to the constraints entered in the UCF file, in order to verify that every constraint is satisfied. Moreover the reasoner is in charge of determining how

the regions configured by the partial bitstreams overlap, and to construct a conflict graph that shows which RFUs are compatible for allocation in the same moment.

The information given by the static photo description is then used to create sub-instances of the conflict graph to show if there will be area conflicts during the evolution of the system.

Design alteration module

Finally the last module provides a way to alter either the position of the RFUs on the device by simulating a bitstream relocation, or to modify the constraints stated into the UCF file, thus providing a way to make design alterations while respecting the partial reconfiguration constraints. Once the modifications have given an acceptable result, the files can be written back by performing the relocation of the partial bitstreams and the modification of the UCF constraints.

4.5.3 Advantages of the proposed approach

The Rebit framework has been created to answer the issues presented in the first part of this chapter. From the given input and the architectural description of the FPGA models introduced in chapter 3 the methodology illustrated in this chapter tries to ease the development process of PDR systems by addressing each of the issues that may prevent the designers of PDR systems to quickly produce an architecture that does not suffer from errors injected by the mistakes that is possible to make during the definition of the system, and to explore different design possibilities to avoid conflicting situations in during application execution.

The following paragraphs illustrate how each particular issue can be solved by analyzing the available data at the end of the system generation.

The satisfaction of the guidelines of the flows proposed by Xilinx is guaranteed by the reasoner module that systematically checks the input UCF file for possible inconsistencies or errors, and signals the warnings to the designer, thus implementing a validation of the desired system against the constraints imposed by the particular flow and model of FPGA chosen.

The overflowing of the resources placed and routed by the PAR programs is prompted to the user again by the reasoner module, which checks area occupation of the partial bitstreams against the constraints of each Reconfigurable Region. A Reconfigurable module thus must either totally be included in a RR or totally not be included in that region. This ensures that no module crosses the boundary of a particular region. This check is made by the reasoner on each partial bitstream for each Reconfigurable Region, allowing easy error checking without having to manually inspect each RFU in FPGA editor.

The definition of the RRs in the UCF file can be validated against the data of the RPM grid for the chosen device by the reasoner: given the area constraint for the slice resource type, the boundary on the overall system given by the RPM grid can be found, thus allowing to check whether the constraints on other resource types fit inside the slice area.

The automated nature of this check allows an error checking on the constraints manually created by the designer, thus allowing a reliable way to validate these constraints and preventing unsuccessful architecture realizations.

The area requirements of each partial bitstream can be calculated, thus allowing the reasoner to build a representation of the conflicts between them, for showing the system designer which functionalities can be configured at the same time and which ones do not, allowing the elimination of errors due to overlapping RFUs. In this way the designer can know where a relocation might be necessary in order to have two functionalities active at the same time.
This allows the low level validation of the application schedule, to see if the sequence of RFUs to be configured on the system is sound with respect to their possible area conflicts.

From the definition of which functionalities will be configured in the different phases of the application life cycle, it is possible to create a subset of the area conflicts between different RFUs, and present it to the user, thus reducing the set of RFUs that can generate conflicts. The system designer can thus view which of the functionalities are involved in a particular phase of the application, and if these functionalities generate any occupation conflict.
This expands the previous point by allowing the developer to focus on a specific application phase, in which some conflicts may exist between the scheduled RFUs, by removing from the conflict graph all of the irrelevant combination checks, i.e.: eliminating the combinations of RFUs that will never be configured together, because they never appear in the same static photo.

The bitstream files are parsed and validated for possible corruption, ensuring that only correct files will be configured on the device at configuration download time. This is the task of the validator module, which holds the parsers of the three different device families introduced in this work.
In this way the bitstream files are ensured to be correct and downloadable to the device. In particular, the log file of the parsing can be inspected, and this log provides the developer with the analysis of every command provided in the bitstream, therefore allowing a particular error situation, due for example to the wrong parameter passing to the *bitgen* program, to be spotted more easily, without having to cross reference the bitstream data with the Xilinx application notes that describe them.

From comparing the portion of the RPM grid defined by a partial bitstream against the overall RPM grid of the device, it is possible to determine the feasible positions for bitstream relocation, thus allowing the designer to experiment with different placements of the RFUs. This is possible because the RPM grid maintains the relative positioning of the device resources, thus providing the information of not only what kind of resources a module configures, but also where this module needs them, being it already translated into configuration information. This functionality is given by the design alteration module of the proposed framework. In this way, if a particular conflict has been isolated and the designer wants to try to solve it without resorting to modifications in the scheduled functionalities, or to the execution of the scheduling algorithm

with less stringent settings, a partial bitstream relocation can be attempted. The developer has thus a way to see where the partial bitstream can be relocated, and to monitor the conflict while experimenting with all the feasible locations, thus trying to solve a particular location conflict by altering the low level placement of the RFUs.

If the location of a bitstream is altered as explained in the previous paragraph, the reasoner module updates the information of the conflict graph between the modules and displays the new information. The designer can so check in real time the global situation of the occupation conflicts in the design, and the situation of the conflicts at a given time of the execution of the application, allowing in this way the possibilty to resolve occupation conflicts in a particular moment of the execution of the application.

The proposed approach tries to give an answer to the lack of support in the Xilinx toolchain for partial dynamic reconfiguration. The issues analyzed in the first part of the chapter, which often drive the developer's focus away from the application towards the reconfiguration details, have been addressed with the exposed functionalities of the Rebit framework.
These functionalities, coupled with a Graphical User Interface, allow the designer to easily spot errors in the resulting architecture and experiment with the relocation of the partial bitstreams in order to solve the issues of area conflicts.
All of the reasoning performed by the Rebit framework has been made at low level, because there was the need of having an alternative validation technique for the PDR architecture,

without the need of performing it directly on the FPGA.

In this sense the Rebit framework fills the gaps in the Xilinx software for developing Partial Dynamic Reconfigurable systems, by enabling a low level debugging facility that replaces many manual processes involved in the creation of such architectures.

CHAPTER 5

IMPLEMENTATION

This chapter will explain in detail how each of the features exposed in the previous chapter have been implemented into the program.

5.1 Data organization and class definition

The environment chosen for developing this particular project has been c++[1] and wxWidgets[2]. This combination of technologies gives a fair degree of portability to the system by allowing compilation and execution on different platforms.
The following subsections will introduce the relevant classes and data structures that have been implemented in the program. These classes roughly correspond to both the parser and the internal data modules illustrated in Figure 23, since they both perform the parsing of the associated files and store an internal representation of the extracted data.

5.1.1 The Project class

The first task of this class is to act as a container class for all of the files supplied by the user:

- the UCF file;

---

[1]The compiler used is GCC 4.0.1 on Mac OS X.

[2]The wxWidgets port used is wxMac version 2.8.7.

- the full configuration bitstream;
- the partial bitstreams.

The project is linked to a panel in the graphical user interface, shown in Figure 24, that allows the user to add the files and to save this list in an xml file for easy retrieval.

When the user adds a file, the object of the corresponding class is created and stored inside the

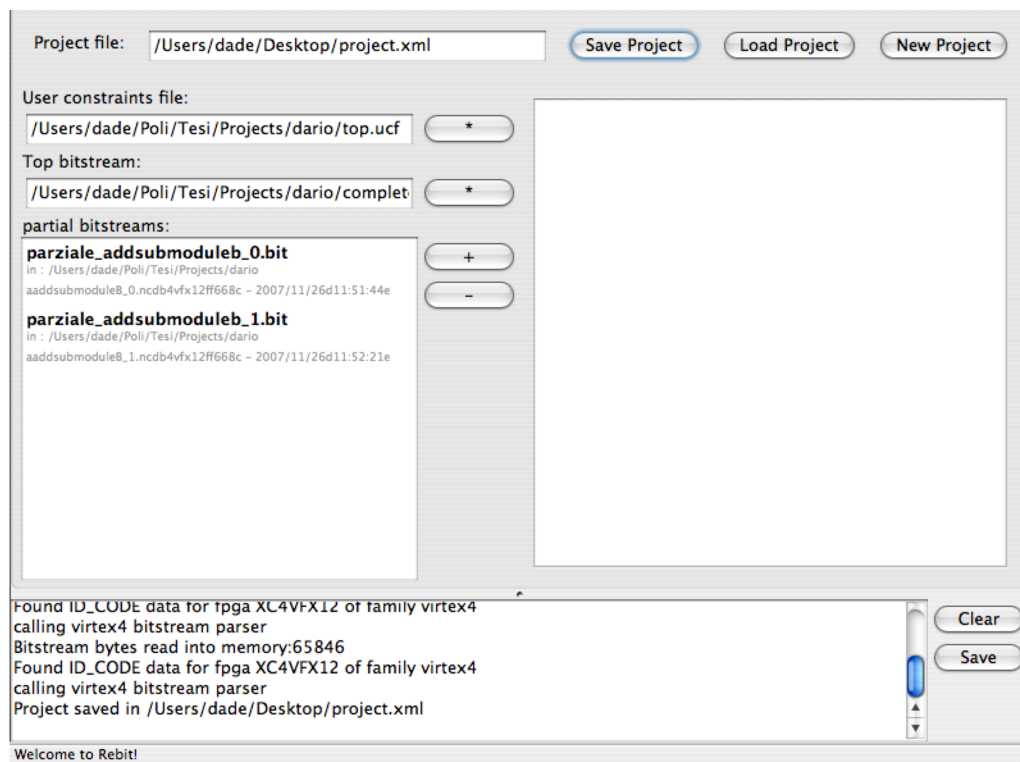

Figure 24. Rebit project panel

Project object. The partial configuration bitstream objects are placed in a vector, that gives a means of updating the number of these bitstream objects dynamically.

Besides the file objects, other global data are stored inside the Project class upon the selection of the first bitstream file: the correct FPGA object retrieved from the FPGA database class and the corresponding full RPM grid of the device. These objects are created by looking at the IDCODE field in the first bitstream added to the project. If a bitstream is added with a different kind of IDCODE, and thus for the configuration of a different FPGA, a warning is given in the console of the program. The information about the FPGA model in use is cleared upon the creation of a new project.

5.1.2 The Bitstream class

The bitstream class has the main duties of parsing the associated bitstream file correctly, prompting if this parsing has ended in an error state, and to store the bitstream text data along with its occupation in area on the device, expressed via a pair of slice corners on the device array. The information associated to the bitstream class is its partial RPM grid, whether it is full or not, the FAR address encountered during parsing, and the number of frames that it configures. The information of the RPM grid is replicated in another RPM object inside the bitstream class, which corresponds to the initial portion of the FPGA configured by the bitstream. This object will be used by the Design alteration modules to modify the placement of the bitstream without changing the original information.

The first operation of the Bitstream class is to copy the data in the file into a vector of *unsigned char*. On this vector is then performed a search for the SYNC word, which signals the beginning

of the actual configuration sequence. In this way, even if the file length is not exactly a multiple of 32 bits, that is it cannot be read word by word from the beginning, the parser can be aligned to a point where it can consume 32 bit words one by one by ending exactly at the last configuration word at the end of the file. Due to the differences in bitstream specifications, a separate parser function has been created for every family considered in this work, however, as seen in chapter 1 the structure of a bitstream file is similar from one family to the other, and the three parsers can thus have the common structure depicted in Figure 25 , which represents a simplified version of the state automaton created to implement them. The main state of the parser is the Header type 1 state: after consuming the sync word, that is the first word where the parser has been positioned by the bitstream constructor, the parser expects to read a type 1 header word. This word is then read and the address of the register it configures is decoded, driving the machine to the appropriate state, while storing the expected word count for the payload of the packet

The majority of the subsequent states expects only one configuration word to be inserted into the appropriate configuration register. These states are:

- **IDCODE**, which reads the code of the particular device for which the bitstream is made;
- **FLR**, which reads a word that indicates the length of the configuration frame in words;
- **COM**, which reads a word storing a command for the configuration logic;
- **COR**, which reads a word that holds the flags for the configuration logic;
- **FAR**, which reads a word that contain the FAR address of the first configuration frame;

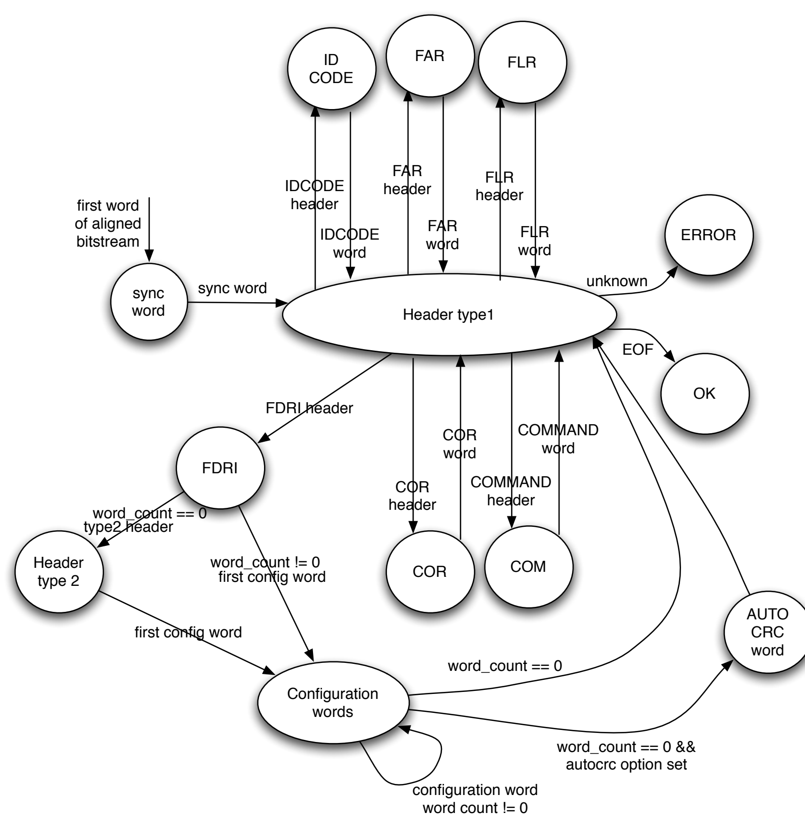

Figure 25. The automaton that parses a bitstream file

All of these states in their exit transition thus consume the packet type 1 payload, interpret it accordingly and return the machine to the Header type 1 state. In the parser realized in the framework there are also the states for all the other configuration registers. Figure 25 depicts only the ones which are relevant for gathering data about the bitstream, to be used throughout the framework. In particular, the COM state is used to interpret the commands given to the configuration logic and stores them for reference, to let the developer verify the correct startup sequences for the FPGA; The FAR address allows to give a placement on the FPGA of the resources configured by the subsequent writes to the FDRI; The FLR register allows to check if the frame length reported in the bitstream is the same for the chosen model of FPGA designed by the IDCODE register. Finally the COR register allows to check if an automatic checksum word is written at the end of the writes to the FDRI register.

When the FDRI header word is consumed, the parser goes in the FDRI state, and reads the word count of the corresponding type 1 frame header. If the word count is zero, a header of type 2 must be expected, where the actual word count is stored, otherwise the word count is read from the type 1 header and the state becomes the acceptance of configuration words. In this state, every time a word is consumed, the word count is decremented, until reaching zero words remaining. While counting the configuration words, a frame counter is incremented at every processed frame, thus keeping track of the number of frames that the bitstream configures and storing this information in the bitstream class.

If the corresponding COR option for the auto checksum word is set, the parser is driven in the AUTO CRC word state, where this word is read and the checksum is calculated to ensure

that the bitstream is not corrupted. After this optional state, the parser is driven again to the header type 1 state for the acceptance of the final configuration commands or to consume pad words, until reaching the end of file and finishing the bitstream parsing stage.

The transitions to the error state are not depicted in Figure 25 for clarity purposes. However some checks are made during the parsing of the bitstream according to the specifications given by Xilinx. For example, if the word count of any of the configuration registers but the FDRI is other than 1 the error state is reached by the parser, or if the IDCODE and the FLR register does not match according to the FPGA data-sheet, the error state is reached as well.

During the execution of the bitstream parsing, a text file is created reporting all the retrieved information word by word, serially, so that some debugging can be made if the error state is reached.

This is an excerpt of this file, showing the tail section of a full configuration bitstream.

```
index 131003 word 0x30008001

 header type1 - address 0x00000004 - wc=1 reg=cmd

index 131007 word 0x00000005

 command - 0x00000005  START command

index 131011 word 0x3000A001

 header type1 - address 0x00000005 - wc=1 reg=ctl

index 131015 word 0x00000000 skip..

index 131019 word 0x30000001

 header type1 - address 0x00000000 - wc=1 reg=crc
```

```
index 131023 word 0x00005F57 crc value

index 131027 word 0x30008001

header type1 - address 0x00000004 - wc=1 reg=cmd

index 131031 word 0x0000000D

command - 0x0000000D  DESYNCH command

index 131035 word 0x20000000 no-op

...

 Bitstream parsing finished

The parsed bitstream configures all the device
```

The most important duty of the bitstream class is the retrieval of the occupation data based on the information gathered by the bitstream parser and on the word count of the write sequences to the FDRI register. The next subsection is thus devoted to illustrate how this computation has been performed in the Rebit framework.

5.1.3 Bitstream FAR progression

As previously pointed out in section 1.3.3, the FAR address written in a standard bitstream is one for each chunk of contiguous frames written to the FDRI register. In order to understand how these chunks actually configure a device, the first step is to know the progression of the FAR address, that is the FAR address for every frame shifted into the FDRI register.

Fortunately, this information is available by invoking the *bitgen* program in debug mode[1] on

[1] bitgen -g DebugBitstream:Yes designfilename.ncd

a .ncd file relative to a full configuration design to discover that the produced bitstream is interleaved with a write to the LOUT register of the configuration logic each time a full frame is found in the bitstream.

From a slightly modified parser class, it is then possible to dump in a text file the values for the FAR address, and store this knowledge for further usage into the program. The needed alteration of the parser automaton is trivial and consists in the duplication of the FAR state into the LOUT state, which is entered upon the decoding of the LOUT register address and writes in a file the FAR address upon its exit transition back to header decoding.

The reason behind the writing of the LOUT register instead of the FAR register is to be found in the capability of Xilinx FPGAs to be configured in daisy chaining mode, that is the bitstream data streamed in the first device of the chain are forwarded to the next one in order to configure several devices at the same time. The repetition of the configuration memory address in this case is necessary in debug mode to avoid possible problems in the interconnection of the FPGAs, where it is easier to have corrupted or missing data streaming across the physical interconnection between the chips. In this case, the configuration logic of a FPGA in the middle of the chain can end up in an error state, preventing all the other FPGAs at deeper levels in the chain to be configured with faulty data.

An excerpt from a FAR progression file, generated starting from a Virtex II Pro full configuration debug bitstream is reported here:

```
: 0x00042800 - block 0 - major 2 - minor 20

: 0x00042A00 - block 0 - major 2 - minor 21
```

```
: 0x00060000 - block 0 - major 3 - minor 0

: 0x00060200 - block 0 - major 3 - minor 1
```

The first field of the file is the address written to the LOUT register, which corresponds to what would be written in a normal bitstream in the FAR address before shifting in the actual configuration words. The next three fields represent the decoding of such address according to the bit masks found in the user guide of the device. In this way the exact progression of the FAR register in the configuration logic internal to the device can be known starting from an initial value and from the corresponding number of configuration frames written to it. In the example reported here, it can be seen how upon the receipt of the 22th frame, corresponding to the minor address 21, the minor address is set back to 0 and the major address is automatically incremented, thus switching from the configuration of one CLB column to the next one.

Moreover, the analysis of the FAR progression for all of the device families object of this work has yielded some insight into the internal configuration memory architecture of Xilinx devices. In particular, what has been observed with Figures 6 and 7 is actually reflected in the FAR progression data for all of the devices taken into account, that is the progression of the FAR address can be strictly related to the architectural specification of the FPGA resource array. In particular, every time the minor address is reset to 0, the separation between the end of a column and the beginning of the next one can be inferred. Since a configuration column configures a particular area of the FPGA array, and the number of configuration frames inside a particular column varies according to the configured resource type, the architecture of the device can be mapped to the data in the FAR progression sequence, and in turn to any flat

sequence of configuration words and to a starting FAR address, thus allowing to know what portion and what resources of a device are configured by an arbitrary partial bitstream.

Some regularity assumptions on the configuration memory space have been made, basically starting from what the manufacturer's user guides hinted at. These assumptions are:

- the sequence of columns from the lowest to the highest major address configures the device array from left to right, at least for the columns configuring CLB sites[1].;
- inside a row of a Virtex 4 device, the progression of columns from left to right is preserved with the normal increments of the FAR address.
- the progression of row and bottom addresses in Virtex 4 devices configures first the top half rows, starting from the center and going towards the upper boundary of the device, then the bottom of the device, again starting from the row nearest to the center and going towards the very bottom edge of the device;
- the majority of the resources of the array is configured by columns having block type 0, and BRAM and BRAM interconnections have reserved block type addresses 1 and 2;
- resources paired with BRAMs, such as multipliers in Spartan and Virtex II Pro devices, and FIFOs in Virtex 4 devices are configured along with the BRAM interconnections;
- within block types 1 and 2 the order of configuration is still from left to right;

[1]Some exceptions have been found, for example in the central clock column of Spartan 3 devices, as reported in 5.1.4

- gaps generated by hard core Power PC processors do not affect the memory mapping scheme, and the configuration bits destined to the portion of the array occupied by the processor cores are simply ignored by the configuration logic.

These assumptions have been verified using the *FPGA editor* and *bitgen* programs in conjunction: first a blank bitstream was generated in the FPGA editor, that is a full bitstream having all of the configuration words set to 0x00000000[1].

A modification of this blank bitstream has then been created in the FPGA editor, by setting the LUT equations of a known slice site in the array in order to view the resulting modified portion of the bitstream by differentiating the resulting bitstream parser log files.

In this way it has been possible to verify the regularity of the configuration data and to completely identify the addressing scheme used in frame indexing. An example of this procedure is determining the meaning of the *top/bottom* bit in the FAR address of Virtex 4 devices: from the documentation it was known that this bit determines the half of the device being configured, but not which half corresponded to a value of 0 and which to a value of 1. By differentiating a bitstream having the equations for slice X0Y0 (situated in the lowest left corner of the device and thus in the second row of the bottom half) set to true against a blank bitstream, it has been possible to spot a difference in the end of the write to the FDRI in block type 0, so that

---

[1]in Virtex 4 devices the initialization of the DCMs and global clock lines is mandatory, so a dummy design with the minimum possible logic - a NOT gate - has been generated in ISE in order to have the DCMs instantiated automatically.

the meaning of the top/bottom bit could be inferred as being true for the bottom half of the device and false for the top half.

5.1.4 Far progression tables and occupation data retrieval

From the data obtained using the approach illustrated in subsection 5.1.3, the tables that map the far address to a particular column of the examined devices have been produced. These tables give an insight of what particular columns are configured by the data having a FAR address with determined top/bottom, row and major addresses, together with the range of the minor address for a particular combination of the other fields. The tables showing the progression of the FAR address for the devices taken into consideration are reported in figures 26, 27 and 28. Two numbers in a table cell indicate the presence of contiguous multiple columns and multiple frames in the configuration of a particular resource type. For example in the second table of Figure 27 there are 46 columns (major address 3 to 48) of CLB resources, each composed by 22 frames (minor address 0 to 21) to configure the whole CLB array of the device, in sequence.

The configuration memory maps for Virtex II Pro and Spartan 3 devices are similar, revealing an analogous structure of the configuration memory spaces for the two different architectural families: in particular the first resource to be configured is a central column of clock resources (gclk), then the IOBs (term) on the left and their interconnections (IOI), then a long sequence of frames configures the whole CLB array, then the IOBs on the right and their interconnections. Eventually the block type address changes to address the configuration of BRAMs and of their interconnections at the end of the bitstream.

XC3S200 configuration memory map / frame indexing			
column	type	major	minor
gclk	0	0	0->2
term_l	0	1	0->1
IOI	0	2	0->18
CLB	0	3->22	0->18
IOI	0	23	0->18
term_r	0	24	0->1
BRAM	1	0->1	0->75
BRAM interconnect	2	0->1	0->18

Figure 26. Spartan 3 XC3S200 far progression and configuration memory map table

XC2VP7 configuration memory map / frame indexing			
column	type	major	minor
gclk	0	0	0->3
IOB left	0	1	0->3
IOI	0	2	0->21
CLB	0	3->36	0->21
IOI	0	37	0->21
IOB right	0	38	0->3
BRAM	1	0->5	0->63
BRAM interconnect	2	0->5	0->21

XC2VP20 configuration memory map / frame indexing			
column	type	major	minor
gclk	0	0	0->3
IOB left	0	1	0->3
IOI	0	2	0->21
CLB	0	3->48	0->21
IOI	0	49	0->21
IOB right	0	50	0->3
BRAM	1	0->7	0->63
BRAM interconnect	2	0->7	0->21

Figure 27. Virtex II Pro XC2VP7 and XC2VP20 far progression and configuration memory map tables

XC4VFX12 configuration memory map / frame indexing					
column	top/ bottom	type	row	major	minor
IOB left	0	0	0	0	0->29
CLB	0	0	0	1->12	0->21
IOB center	0	0	0	13	0->29
CLOCK	0	0	0	14	0->2
CLB	0	0	0	15->18	0->21
DSP	0	0	0	19	0->20
CLB	0	0	0	20->27	0->21
IOB right	0	0	0	28	0->29
IOB left	0	0	1	0	0->29
CLB	0	0	1	1->12	0->21
IOB center	0	0	1	13	0->29
CLOCK	0	0	1	14	0->2
CLB	0	0	1	15->18	0->21
DSP	0	0	1	19	0->20
CLB	0	0	1	20->27	0->21
IOB right	0	0	1	28	0->29
IOB left	1	0	0	0	0->29
CLB	1	0	0	1->12	0->21
IOB center	1	0	0	13	0->29
CLOCK	1	0	0	14	0->2
CLB	1	0	0	15->18	0->21
DSP	1	0	0	19	0->20
CLB	1	0	0	20->27	0->21
IOB right	1	0	0	28	0->29
IOB left	1	0	1	0	0->29
CLB	1	0	1	1->12	0->21
IOB center	1	0	1	13	0->29
CLOCK	1	0	1	14	0->2
CLB	1	0	1	15->18	0->21
DSP	1	0	1	19	0->20
CLB	1	0	1	20->27	0->21
IOB right	1	0	1	28	0->29
BRAM Interconnect	0	1	0	0->2	0->19
	0	1	1	0->2	0->19
	1	1	0	0->2	0->19
	1	1	1	0->2	0->19
BRAM data	0	2	0	0->2	0->63
	0	2	1	0->2	0->63
	1	2	0	0->2	0->63
	1	2	1	0->2	0->63

Figure 28. Virtex 4 XC4VFX12 far progression and configuration memory map table

For Virtex 4 devices, the progression in each row configures the left IOBs, a first set of CLBs, the central column of IOBs and the clock column with DCMs and the global clock lines, a second portion of the CLB array, a column of DSPs, the third portion of the CLB array and the IOBs on the right.

The same structure is repeated for each row as the top/bottom bit and the row address change to cover the entire device. The separation of the BRAM configuration data from the rest of the array configuration has been preserved from the previous families, so that in the final part of the bitstream the block type address changes to indicate BRAM interconnect and BRAM frames, cycling for each type the row address and the top/bottom bit.

As a remark on the proposed tables, it is worth noticing how the evolution of the FPGA technology is pushing towards the realization of arrays more and more interleaved with dedicated hard-cores, and how this trend is reflected in the configuration of the newer device families: the configuration memory space of a Virtex 4 device, in fact is by far less homogeneous than the ones of the Spartan 3 and Virtex II Pro families.

The information used to generate the proposed configuration memory mapping has also been stored in text files in the program, and read into a vector of a structured type containing all of the fields of the FAR address. This progression vector is then colored during the parsing of the FDRI data of any bitstream fed into the program by setting a bit for each vector element which indicates whether a frame is configured or not.

In this way the very same progression used by the device configuration logic to increment the FAR address can be known by the framework and used to determine the configured resources.

Figure 29 illustrates how the coloring of the FAR progression vector is made using the data of the bitstream[1]. In each Bitstream class the progression vector is initialized with the sequence

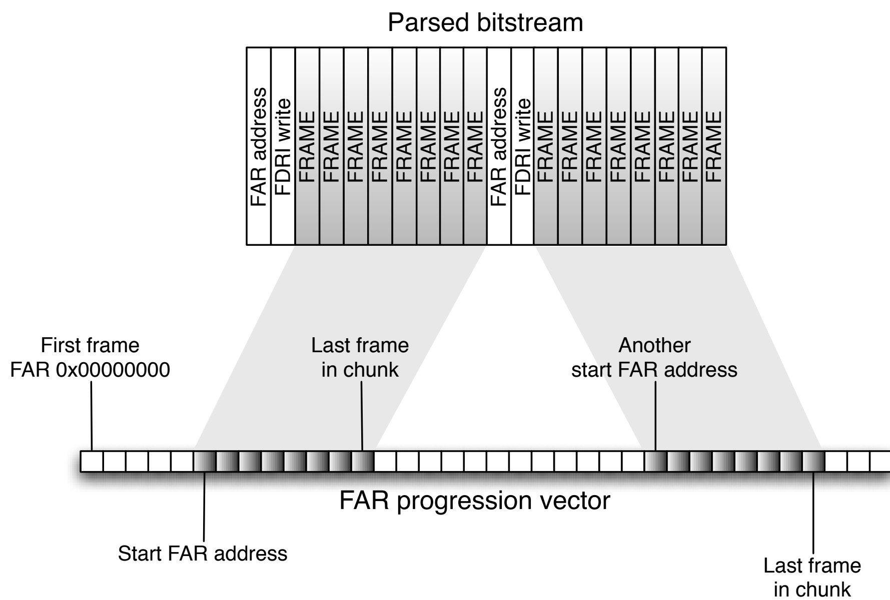

Figure 29. FAR progression vector coloring according to bitstream content

of all the possible FAR addresses of the device, then, in the parsing phase, the automaton of Figure 25 has been enriched in order to save the starting position upon encountering a write to

[1]The illustration of the bitstream composition of Figure 29 has been simplified with respect to the real bitstream structure to reflect only the relevant details used for area occupation retrieval.

the FAR register, and to color the current element of the array whenever a complete frame is recognized, incrementing the index of the array on every coloring operation.

Moreover, the presence of padding frames in Spartan 3 and Virtex II Pro devices has been taken into account, that is the last colored vector element is set back to a non-configured state when a chunk of writes to the FDRI is finished. This fact prevents the padding frames usually sent to the configuration logic to shift out the last configuration data from the FDRI register from being counted as actual configuration frames.

The last step in retrieving the occupation data for a generic bitstream consists in inspecting the colored FAR progression array to interpret the configured positions in order to extract slice occupation data. For Virtex II Pro and Spartan 3 devices the extraction algorithm is simple and consists basically in consuming all the non configured positions of the progression array, computing the left X slice bound for the first configured position, consuming all the configured positions and calculating the right X slice bound on the last configured position. The pseudo code for this algorithm in the case of a Spartan 3 device is given in the following listing:

```
//beginning of CLB columns
index = FindFarIndex(0,3,0);

//scan all the non-colored positions
while (farprogression[index].configured == 0)
        index++;

//index of the first configured frame
```

```
left_x=(farprogression[index].major - 3) * 2;

//scan all the colored positions
while (farprogression[index].configured == 1
          && farprogression[index].major <= 22){
         index++;
}

//go back to the last colored position
index--;

right_x=((farprogression[index].major - 3) * 2) + 1;
```

The equations for computing the left and right slice bounds have been generated on the basis of the data of the memory mapping tables, while the Y slice bounds are always equal to 0 and to the height of the slice array. The same basic algorithm has been ported to interpret the colored FAR progression vector of Virtex 4 devices, by doing the same scan for each row and computing the values of the Y slice coordinates according to which rows have configuration data. The extraction of the X bounds is not as straightforward as in the previous case, because the interruptions in the CLB array must be taken into consideration, again referring to the table in Figure 28. The X bounds of each row are then compared to those of all of the other

rows and tested for equality[1].

The operations done in the Bitstream class constructor are therefore:

- Populate a vector of *unsigned char* with the raw data of the bitstream.
- Sync the parsing finite state automaton on the beginning of the actual configuration words and extract the comment at the beginning of the bitstream.
- Find the IDCODE word and initialize the correct FPGA object from the database accordingly.
- Initialize the FAR progression array for the particular FPGA model.
- Call the appropriate parsing finite state automaton according to the device family.
- Call the appropriate slice occupation extraction function according to the FPGA model on the basis of the colored FAR progression vector.

The average running time for the constructor of Virtex 4 Bitstream classes has been measured to be in the order of magnitude of tenths of second on an Intel core 2 duo 2 GHz processor. In particular 600 milliseconds have elapsed during the construction of a Virtex 4 full bitstream and an average of 150 milliseconds for the parsing of partial bitstreams configuring one tenth of the same device.

5.1.5 The UCF class

The UCF class stores the lines in the UCF file that refer to an AREA GROUP constraint and stores for each one of these lines the name of the area to which the constraint belongs,

[1]It would be an error to have a non-rectangular area configured by a bitstream

the type of resource configured by a particular constraint and the lowest left and upper right corners in the resources definition file.

The c++ class that represents the UCF object has been paired with a C scanner function generated with GNU Flex[1] which recognizes the lines of interest via a regular expression and updates a vector of constraints in the UCF class itself.

If the corners are specified the other way around, i.e the upper left and the lower right the UCF class inverts them to have an uniform representation of the constraints.

5.1.6 DB class

This class holds objects for every family and every FPGA considered in the present work. These objects are made from a parent class that simply defines an object with properties, and these properties can in turn be one of the following types:

- **boolean flags**, to store information such as if the particular family supports 1D or 2D reconfiguration;
- **String objects**, such as the device or family name;
- **32-bit words**, used in the bitstream parsing section to identify the ID-Code for a particular device and the constants required in the parsing section;
- **integer numbers**, like the frame count of the device, the RPM grid maximum and minimum values, the length of the frame word etc.

---

[1]The version of GNU Flex used is 2.5.33

In these objects the various items are inserted in four different standard library *map* objects, one for each property type, that map a String identifying the property name with the property itself. Then the various getter and setter methods have been implemented.

For the expandability of the solution a simple XML format has been envisioned to store the different properties in a textual file, and populate the database upon the program execution. The global variable DB, initialized on program start, permits then the easy retrieval of the FPGA architectural information from any point of the program.

5.1.7 RPM class

The RPM class is basically a bidimensional vector of a structured type which holds:

- a string with the resource type, as named in the UCF file;
- an integer coordinate pair that represents the numbering associated with the resource in the conventional resource numbering system;
- another integer pair with the coordinates of the RPM grid.

A special type of resource, the EMPTY resource has also been devised to store the gaps in the grid.

Upon creation of the object, the grid is initialized to store an EMPTY type in all cells, then the coordinates are read in from coordinate files, one for each resource of a particular device. As mentioned before, these files have been created resorting to the log of the FPGA Editor program by Xilinx, selecting the resources individually.

The format of a file holding coordinate information has been kept simple in order to allow reads

with a common *sscanf* function, namely consisting in the coordinates of the resource in the normal numbering system and their corresponding value in the RPM system, as an example an entry for a slice would be "X39Y37 X60Y79".

The RPM grid for every FPGA of this work can thus be initialized starting from these files. A representation of a Spartan 3 FPGA is given in Figure 30 as the result of the interpretation of the RPM grid data. Another feature of the RPM class is a constructor that takes as a

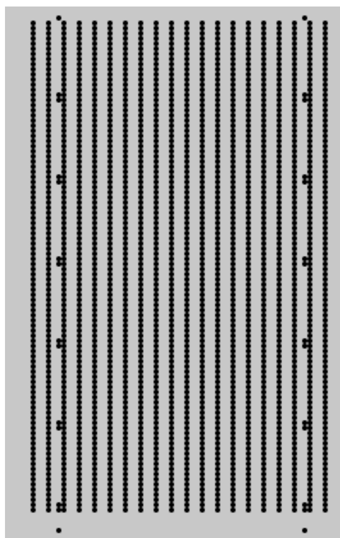

Figure 30. The representation of the resources of a Spartan 3 XC3S200 FPGA as the result of the interpretation of RPM grid data

parameter an existing base RPM grid and the coordinates of the corners of a slice portion of

the FPGA array in order to build a subset of RPM grid used to model the resources configured by partial bitstreams.

Finally, a method in this class allows the extraction of all of the resource bounds included in a given bound on the slice resource type, in order to allow the reasoner module to check if an UCF constraint includes all of the resources encompassed by a particular area defined in the slice numbering space, and to allow the UCF editor component to work on editing the slice constraints and to fill them with other resources on the commit of the editing operation.

The pseudo code for this method of the RPM class is given in the following:

```
vector<constraint> GetIncludedResources(x0,y0,x1,y1){
        vector<constraint> included;

        //convert slice to rpm coordinates
        (rx0, ry0) = find_rpm(x0,y0);
        (rx1, ry1) = find_rpm(x1,y1);

        for (scan rpm X from rx0 to rx1)
        for (scan rpm y from ry0 to ry1)
        {
         n = GetRPMNode (x,y);
         if (n.type != "SLICE"){
             if (n.type not found in included){
                 included.push (n.type, n.x, n.y, n.x, n.y);
             }else {
```

```
                included[n.type].x0 = min(included[n.type].x0, n.x);
                included[n.type].y0 = min(included[n.type].y0, n.y);
                included[n.type].x1 = max(included[n.type].x1, n.x);
                included[n.type].y1 = max(included[n.type].y1, n.y);
            }
        }
    }
    return included;
}
```

5.2 Canvas class

Another type of class, strictly related to the GUI of the framework, has been created. This Canvas class gives a way to draw the data of the program, for example the drawing in figure 30. This generic class serves as a base for implementing specific visualization classes, offering automatic zooming and centering services for the painted graphics.

The classes that inherit from this base Canvas class can then implement the *onpaint* event handler to draw on the required panel of the user interface.

5.3 Feature implementation

After having analyzed the model of data as it is retrieved and stored by the program, this section explains how the features introduced in the previous chapter have been implemented.

5.3.1 Validation of the flow constraints

The constraints of the PDR flows are validated and possible errors are reported to the user. The implementation of the checks based upon the data read from the UCF file supplied by the user. The checks performed are on the dimension of the reconfigurable areas defined in that file:

- the difference between the horizontal slice coordinates of a RR has to be multiple of four;
- the first horizontal coordinate of a RR must be a multiple of four itself;
- the height of a RR, in the case of Spartan and Virtex II Pro families must correspond to the height of the device, that is the vertical coordinates of the RR must be respectively zero and the height of the slice array.

Moreover a check on the UCF constraints is performed based on the data available for the RPM grid of the particular device of the project, that is all the resource types other than slices are associated with their RPM grid equivalent region, and this region is checked to be contained in the RPM region calculated with the slice constraint.

5.3.2 Validation of area constraints

The data retrieved from the partial bitstream files can be associated with slice occupation data: from the starting FAR address read from the bitstream and from the number of frames that have been found in the writes to the FDRI register it is possible to determine the area of the device where the partial bitstream is placed, by counting the columns of slices that are configured by the frames read.

This occupation data can then be easily compared with the slice constraint given in the UCF file for all the RR defined. The check made here for every partial bitstream is if its slice occupation either fits entirely into a RR or does not appear at all to be part of the RR under examination. In this way it is avoided that a particular partial bitstream configuration crosses the boundary of a particular RR, thus spotting possible inconsistencies generated by the PAR program.

5.3.3 Area conflict retrieval

In the Project class, a method is defined to check every partial bitstream occupation against each other, based on the slice portion that each particular bitstream configures.

In this way a conflict graph between each RFU can be produced, by checking if a particular pair of slice area corners creates an overlap.

The conflict graph will thus be composed by a node for every partial bitstream, and an arc between two conflicting bitstreams. The arcs on the graph are not directed, and there are no arcs that start and end on the same node, so it is possible to represent this data structure with the lowest left half of its incidence matrix with dimension the number of partial bitstreams. The diagonal of the matrix is not included since a particular bitstream cannot be in an area conflict with itself.

A sample conflict graph is shown in Figure 31. In this example it can be seen that RFU 1 cannot be configured together with RFU 3, RFU 3 cannot be configured with RFU 4 and RFU 2 and 5 cannot be configured together and so on. In other words wherever there is an arc, the two connected RFUs cannot be placed at the same time on the FPGA. The conflict property is not necessarily transitive, so that if there are two conflicts between RFU 1 and 3, and between

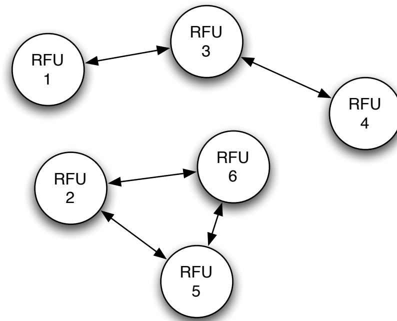

Figure 31. A sample conflict graph between 6 different RFUs

RFU 3 and 4, this does not mean that there is a conflict between 1 and 4, because that would have been indicated by a separate arc connecting the two corresponding nodes.

The conflicts are drawn with red squares in the program and offer a straightforward glance at which RFUs are compatible with each other in every moment of the application execution.

5.3.4 Conflict display in the evolution of the application

After the computation of area conflicts between the bitstreams, some subsets of the conflict graph can be shown, based on the information stored in the project class on which bitstreams will be configured at a given time frame. To obtain such information the user has two opportunities: the first one consists in pressing the 'n' key on the keyboard while viewing the conflict matrix. This option cycles through all the configuration combinations given in the project and, for each combination, adds a filter in the *onpaint* method of the canvas that displays the conflicts in order to view only the rows and columns of interest. In this case viewing the conflicts

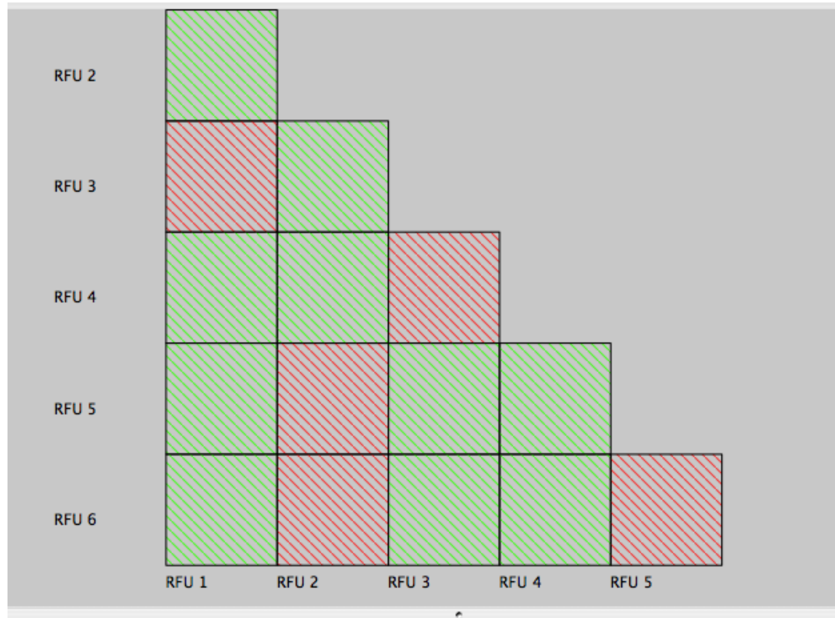

Figure 32. The conflict sub-matrix corresponding to the example in Figure 31

in a particular application situation is easier, because the matrix shows only the relevant data for the application in a particular moment, by letting the designer focus on the resolution of a given conflict. Figure 33 shows a particular instant of the application in which only RFUs 1,3,4,5 are configured at the same time. In addition to this view, a 3D view of the conflicts in the different moments of time has been implemented using the OPEN GL libraries. The user can scroll the view to browse the different moments of time in which the application evolves. Flattening the stack of the different time instants yields the overall conflict matrix as shown at the beginning of this section. A screenshot of this view is provided in Figure 34.

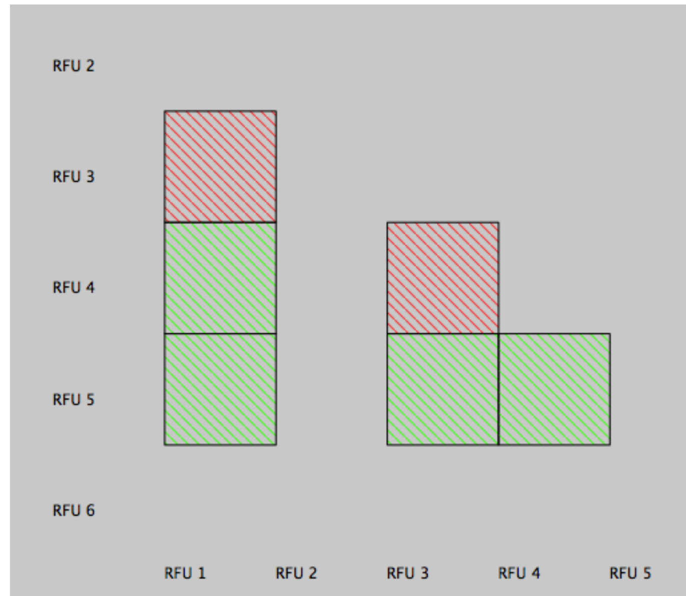

Figure 33. The conflict sub-matrix filtered to display only a subset of the RFUs.

5.3.5 Bitstream relocation

The relocation panel provides the necessary functionalities to simulate the relocation of partial bitstreams in the application. The key 'n' cycles through the representation of the partial bitstreams added to the project on a FPGA view.

The information used to draw the particular bitstream is its partial RPM grid, obtained as explained in the bitstream class description. This RPM grid is drawn upon the FPGA representation to show the area occupation information. If the user presses then the 'a' key, all the possible positions for the bitstream are displayed, and the relocation mode is entered. This mode calculates all the possible positions of the partial bitstream on the FPGA area.

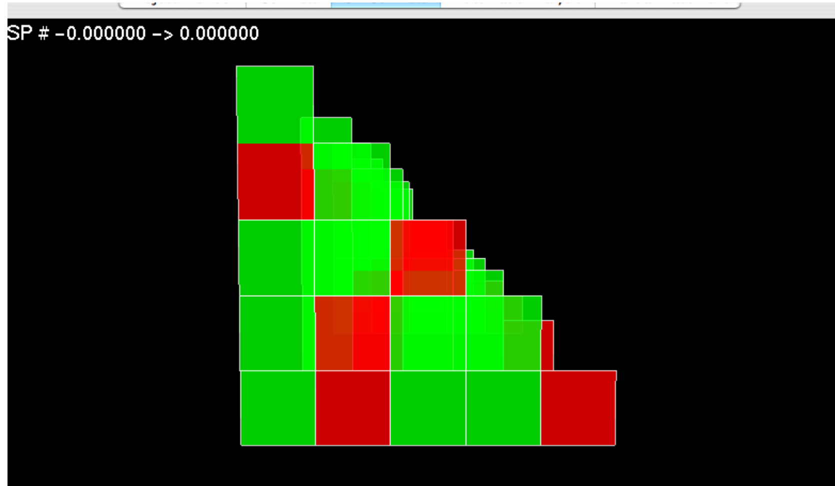

Figure 34. The 3D view realized to browse the different moments of the application evolution showing the appropriate RFUs.

The algorithm used to calculate these positions is to shift the partial RPM grid in every possible position upon the global RPM grid. If the resource types match in the two overlapped RPM grids, then a relocation is possible, because the resources offered by the FPGA in that particular point of its configuration array match the ones needed by the partial bitstream, both in number and in position.

When such a position is found, the RPM corners of that position are saved into a vector that keeps track of all the possible positions. These locations are then displayed in the user interface with red outlines, and the user can cycle through them with the letter 'q'.

The devised algorithm therefore employs a *sliding window* technique to make the partial bitstream RPM grid slide over the global device grid, keeping track of when the contents of the

window and of the partial bitstream grid generate a match, that is when each cell of the two RPM grids has the same type, no matter the coordinates stored in the cell. The pseudo code for the implemented algorithm is the following:

```
vector <constraint> ComputeRelocation (fpgaRPM, partialRPM){
vector <constraint> positions;

int max_offset_x = fpgaRPM.max_x - partialRPM.max_x + 1;
int max_offset_y = fpgaRPM.max_y - partialRPM.max_y + 1;

for (int j = 0; j < max_offset_y; j++){
for (int i = 0; i< max_offset_x; i++){

  if ( fpgaRPM.Match(partialRPM, i, j) ) {
        positions.push (current position slice bounds);
  }
}
}
return positions;
}

bool Rpm::Match (Rpm * portion, int offset_x, int offset_y){

for (int j = 0; j < portion.max_y; j++){
```

```
for (int i = 0; i < portion.max_x; i++){
          node n_sub; //a node from the sub grid
          node n_master; // a node from this grid
          n_sub = portion.Get(i,j);
          n_master = Get(i+offset_x, j+offset_y);

          if (n_sub.type != n_master.type) {
                    match = false;
                    return false;
          }
}
}
return true;
}
```

The first method in the code is responsible for shifting the portion of RPM grid (partialRPM) over all possible positions in the RPM grid of the FPGA (fpgaRPM), and for every position the Match method of the Rpm class is called in order to see if the two grids originate a match and thus seeing if a relaocation in that position is possible. If this is the case, the position is translated from RPM coordinates into slice coordinates resorting to the facilities of the Rpm class itself, and the resulting slice bound is then added in the vector positions.

The match method of the Rpm class in turn is responsible for computing if the passed RPM grid portion matches in a particular position of the RPM class. To do this every position in the

partial RPM grid is scanned, and a true value (a match) is returned if and only if all the node types of the shifted partial RPM grid and the ones from the global RPM grid match, otherwise a false result is returned.

The performance of the algorithm on the data for a Virtex 4 device has been measured to be in the order of the hundredths of seconds for computing the feasible relocation positions of a partial bitstream with a 10% occupation of the device. Figure 35 exemplifies the aforementioned algorithm. The type of the RPM grid cell is indicated by the letter S, for the slice type, and

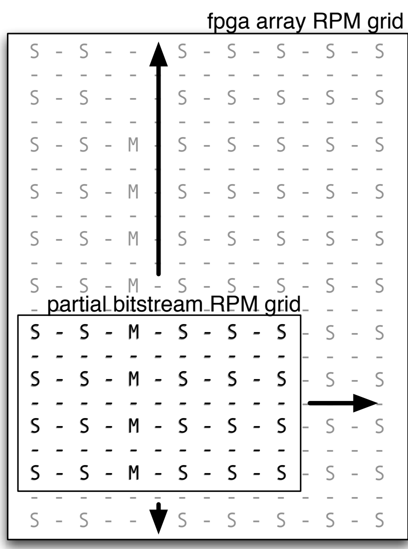

Figure 35. A sample of the sliding window algorithm used to compute the possible placement of partial bitstreams

with an hyphen where the content of the grid is empty. the letter M indicates the presence of a multiplier hard core in that position of the array. In the example the contents of the sliding window match the contents of the partial RPM grid, and thus a possible relocation position is recorded. On the opposite, if the sliding window would have been in a point of the algorithm where the multiplier resources were not available, the match of the two portions would have been negative, and thus no relocation position is recorded in this case.

While cycling through the different positions, the possible area conflicts with other RFUs are printed on the console of the program, so as to monitor which conflicts the relocation will cause. This is done by calling the area conflict reasoner, which this time works on the temporary data of the possible allocation

The 'c' key then allows to commit the position of the bitstream, so that the previous conflict checking and constraint validation phases can be rerun again by the program. The 'n' key causes the cycling on different bitstreams, in order to exit from a relocation operation on any of them.

Figure 36 shows the Placement window of the program with a partial bitstream configuration shown on the top left corner on the Virtex 4 VFX12 device. Four possible placements have been found by the program, and are shown in the figure marked with a red outline. No UCF file has been loaded, so that all the possible relocation positions have been found, without the constraints imposed by the Reconfigurable Region definition.

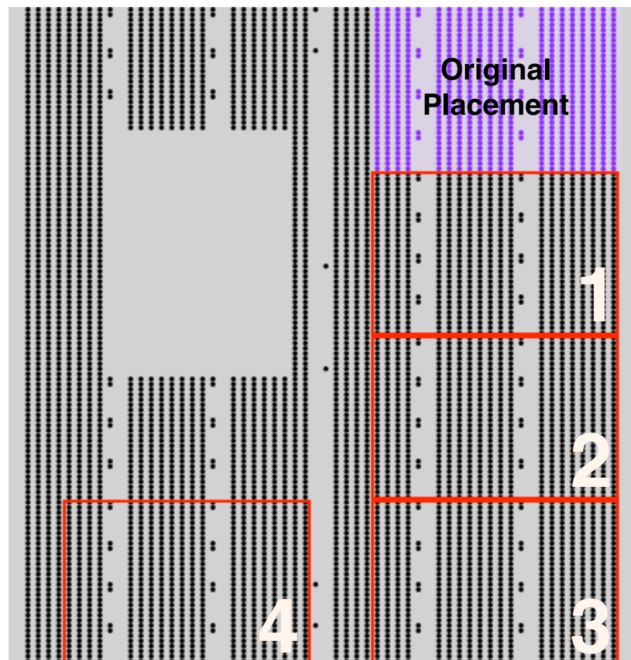

Figure 36. A screenshot of the program showing the positions computed for the partial bitstream relocation. From the original placement in the top left corner, four other different placements have been found.

5.3.6 UCF view

The last view offered by the program is the UCF view, in which the constraints provided by the user in the UCF file are viewable and editable. The editing of the UCF RR areas is aware of all of the constraints that are defined in the flows for dynamic reconfiguration, the same that are checked in the validation phase. On a Vitex 2 Pro, for example, vertically resizing a constraint is forbidden and the horizontal resizing and repositioning are always made on a 4 slice boundary. In Virtex 4 devices, instead, the vertical resizing of the ucf area is allowed on steps that include whole rows of frames.

Moreover, the editor forbids the crossing of two different RR, by checking if their slice boundaries do not cross each other. The resizing of a RR constraint always happens first on the slice boundaries. After that the constraints on other resource types are generated on the basis of the RPM grid information, by exploring the portion of grid define by the slice boundaries and retrieving the minimum and maximum coordinate values for every resource type encountered. The information of this editing is then written in a mirror data structure in the UCF file, as not to overwrite the one originally parsed by the program, and it is written on the console for the substitution in the original file.

CHAPTER 6

CASE STUDY

This chapter is devoted to the illustration of the employment of the proposed Rebit framework in the analysis of the configuration files of a specific use case, an architecture created for image processing algorithms implemented in hardware that make use of soft computing techniques for their optimization and that exploit the partial reconfiguration capabilities of a modern FPGA chip to efficiently optimize area usage of its resource array.

6.1 The chosen architecture

The platform chosen to validate and employ the framework proposed in the present work in a case study has been an architecture developed in our lab for research in the evolvable hardware field (Mattasoglio et al., 2006 2007). This topic of research employs soft-computing techniques in order to customize and optimize hardware soft-cores. In the case of the proposed work, the optimized functionality has been an hardware implementation of an edge detection algorithm ported in hardware. Besides the details involved in the particular research topic chosen in the development of this studied architecture, its implementation is well suited to be analyzed with the Rebit framework: the partial dynamic reconfiguration capabilities of a Virtex 4 FPGA are exploited to switch from two different IP-Cores during the execution of the edge detection algorithm, while a static part of the architecture takes care of the communication of the data to be streamed into the design and of reconfiguration of a RR destined to hold

two different hardware functionalities according to the particular needs in a moment in the execution of the application, thus time-sharing the space available on the FPGA array between different functionalities.

In the proposed architecture, the IP-Cores that share the resources offered by a reconfigurable region are two image processing functions: a grey-scale filter and an edge detection filter.

The representation of the design as given by the FPGA editor program is reported in Figure 37 The Power PC block in this case has not been used. The processor employed in the static part of the architecture is a Microblaze soft core processor, which communicates with the two RFUs via the OPB bus and the necessary bus macros. The reconfiguration scheme employed to switch from one RFU to the other has been external partial dynamic reconfiguration.

The input files given to the Rebit framework are: the full configuration bitstream of the device, the UCF file defining the boundaries of the reconfigurable region and the two partial bitstream files, as the screenshot of Figure 38 illustrates. The log of the program reports the successful parsing of the full and partial bitstreams, their slice occupation data, along with the FPGA model that all of these bitstream configure. In this way the correct loading of bitstreams for the same device can be identified, and the accidental substitution of a partial bitstream for a full one or vice-versa can be identified by looking at the slice occupation data and at possible warning messages in the log of the program itself.

At this point, all of the relevant data structures introduced in the previous chapter have been created and populated with the data retrieved either from the database of the program or from

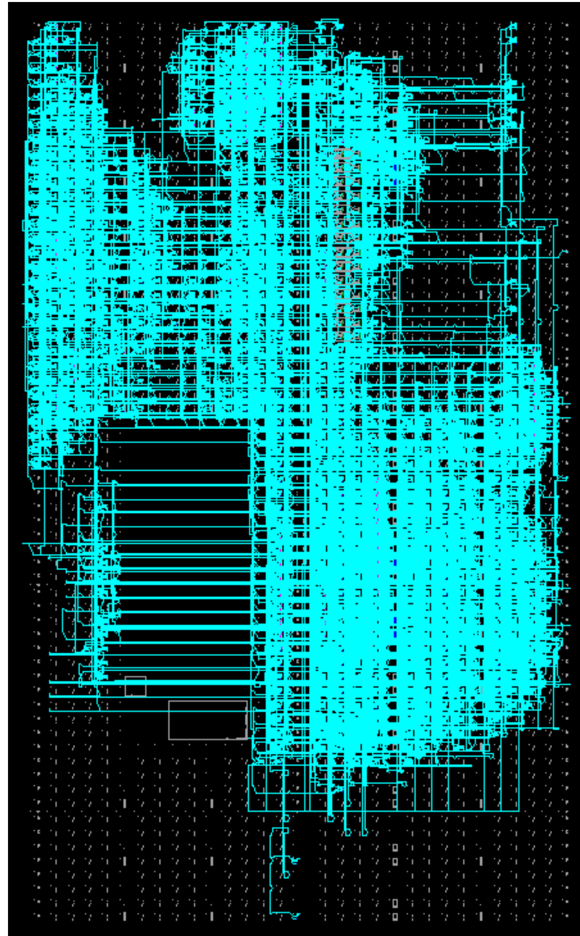

Figure 37. The representation given by the FPGA editor of the configuration of the XC4VFX12 FPGA with the static part of the evolvable hardware case and the edge detection module configured.

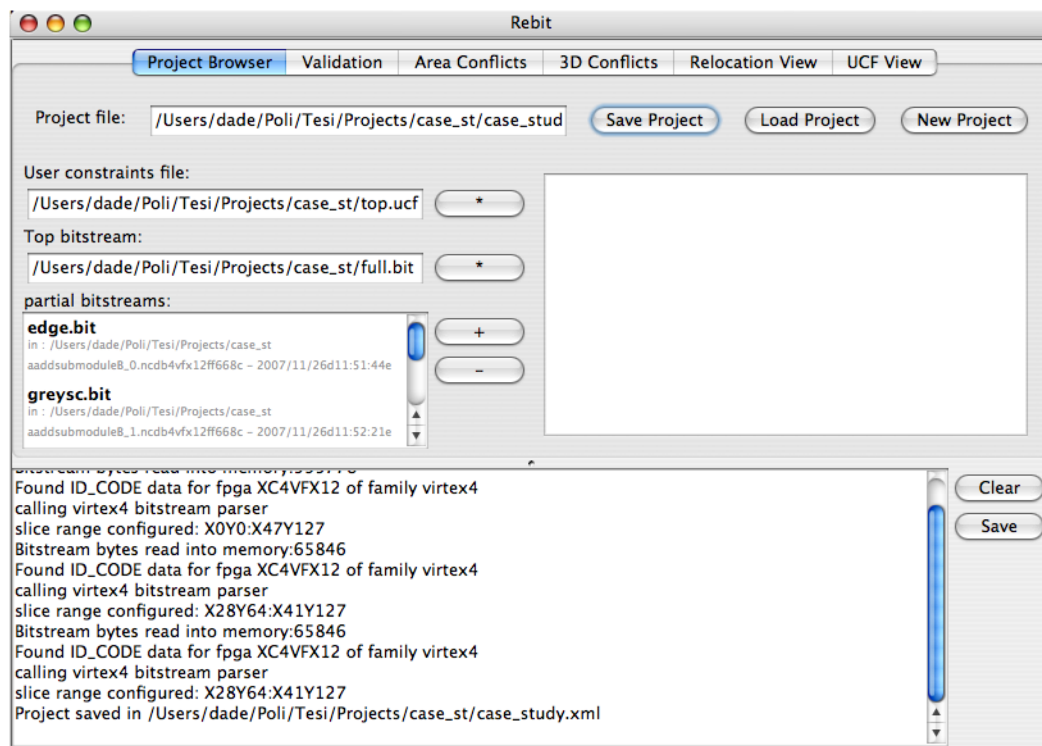

Figure 38. Screenshot of the Rebit project pane with the case study files loaded and the results of bitstream parsing.

the particular files loaded into it. The further sections are thus devoted to explain the analysis possibilities that the reasoner component of the framework can perform on the data.

6.2 Validation of the UCF file

The first step of analysis for the proposed architecture has been the analysis of the definition of the reconfigurable region in the UCF file supplied. In this case the UCF view panel of the program gives a view of the UCF slice constraints mapped on the FPGA representation obtained by the RPM grid data. Such a representation is illustrated in figure 39: only one reconfigurable region has been extracted from the UCF file parser, its area group constraints stored in the UCF class and this data has been overlayed by the rendering loop of the canvas class over the FPGA model. The corresponding data in the UCF file is the following:

```
INST "addsub_B" AREA_GROUP = "AG_PRMB";

AREA_GROUP "AG_PRMB" RANGE = SLICE_X28Y72:SLICE_X34Y127;

AREA_GROUP "AG_PRMB" RANGE = DSP48_X0Y26:DSP48_X0Y27;

AREA_GROUP "AG_PRMB" MODE  = RECONFIG;
```

Constraints are given for this reconfigurable region on the SLICE and DSP48 resource types. The DSPs are needed by the edge detection module, thus a region in the area of the device offering such hard-cores has been chosen by the system designer. The validation panel of the Rebit framework offers an interface with the reasoner module, in order to perform the checking of the soundness of the definition of such a reconfigurable region. The results of the validation of the reconfigurable region are given in Figure 40. The validation results report the analysis of each reconfigurable region at the beginning, followed by the inclusion check for each

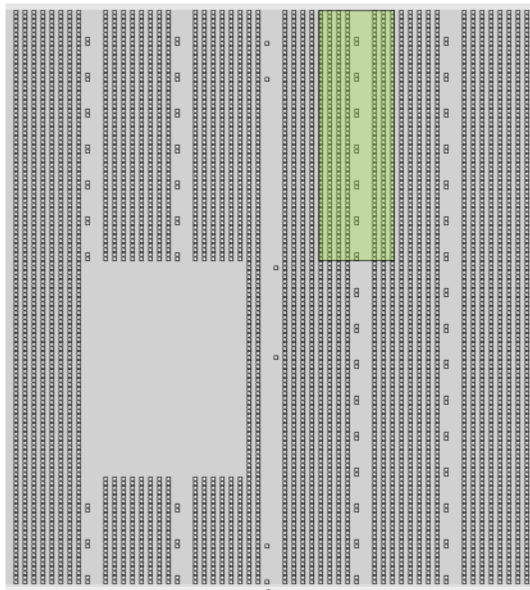

Figure 39. Screenshot of the Rebit UCF View with the initial slice bounds defined in the UCF file displayed.

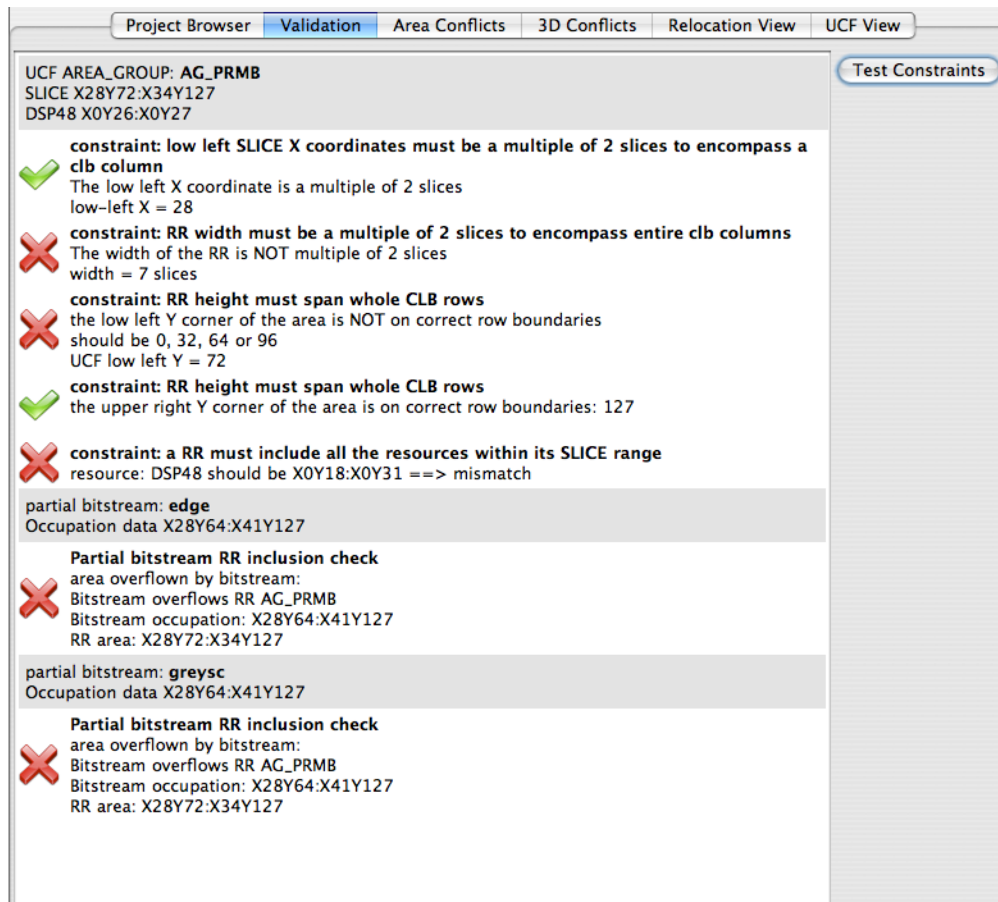

Figure 40. Screenshot of the Rebit Validation panel reporting the results of the validation of the UCF reconfigurable region and of bitstreams inclusion checks.

of the partial bitstreams specified in the project, to see if they are contained in one of the reconfigurable regions specified in the UCF file. The green mark indicates the conformance of the reconfigurable region to the particular constraint, while the red mark signals an error or a warning that could generate possible problems.

For the area group proposed by the system designer, the first constraint checked is if the RR defined is aligned with the CLB column granularity, that is, since a CLB is composed by a 2x2 set of slices, it is checked if the reconfigurable region does not have any edge dividing a CLB in half. The first two results report that such a constraint is satisfied for the left edge of the reconfigurable region, but not for the edge on the right, which gives origin to a RR of width 7 slices, while the immediately next possible values could only be 6 or 8 slices, to encompass whole CLB columns.

The next constraint is to be interpreted as a warning: it reports that the lower edge of the RR region rectangle is not aligned on row boundaries, so that a partial bitstream loaded in this area will certainly overflow it, because the data written into a bitstream must compulsory vertically span a set of contiguous rows on the FPGA device, that is the structure of the configuration bitstream, as explained in the previous chapter, only gives a row granularity on what resources are configured vertically, since its composition is given in frames, which span no less than an entire row in height.

However, given the glitchless reconfiguration capabilities of the device under examination, this constraint violation can be overridden, because the partial bitstream will configure both the RR area with a new functionality, and the area immediately under it with the same configuration

data of the static part, thus the vertical region from slices Y72 to slices Y127 will hold the RFU, while the area from slices Y64 to Y71 will be reconfigured with the same data as before, without alterations in the operation of the static part.

For the upper bound of the reconfigurable region, instead, these observations not apply because it is on a row boundary, and the reasoner reports no possible problems.

The final constraint validation on the proposed RR is on resource inclusion: the program has in fact retrieved all the resources included within the SLICE bound of the region, and has called the appropriate method on the FPGA RPM grid to retrieve all of the other hard-cores included in this area, comparing them with what has been declared in the UCF constraint. In this case only two DSPs are included in the UCF region definition, while the area has 14. The correct constraint is then reported as a hint to resolve the error: while in the UCF we have X0Y26 and X0Y27 DSPs, the program suggests to include DSPs from X0Y18 to X0Y31 in the area group definition.

The analysis is then performed on the partial bitstream occupation data, compared to all the reconfigurable regions defined in the project. In this case, the only RR defined in the UCF file is overflown by both the partial bitstreams supplied. The occupation of the bitstream is reported along with the UCF SLICE constraint that has been violated, to allow an easy comparison of the two.

For both partial bitstreams the area is overflown vertically and horizontally. The overflowing can be easily viewed by switching from the UCF View and the Relocation View panels of the program: the first shows the reconfigurable regions, and the next shows the occupation data

for one of the specified partial bitstreams. Figure 41 is a composition of the screenshots of the two panels, which shows how the bitstream occupation (in red) overflows the reconfigurable region (in green). The vertical overflowing, as said before, is only to be interpreted

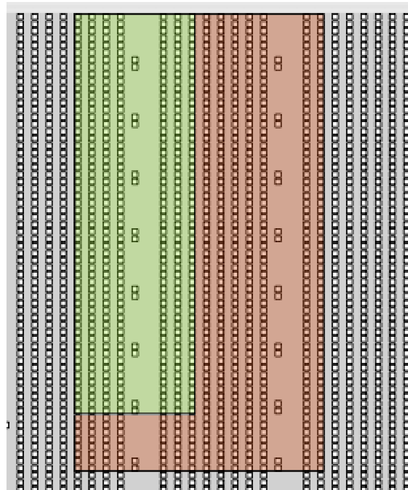

Figure 41. Overlaying of two screenshots from the Relocation View and from the UCF View panels showing how the partial bitstream (red) overflows a reconfigurable region (green).

as a warning, because it is caused by the bitstream configuration granularity. However, the horizontal overflowing has to be taken into account as a source of errors, because it clearly indicates that the resources defined in the UCF file to be part of the RR are too scarce to include the RFU defined by the bitstream.

To solve this problem, the UCF View has editing capabilities that allow the redefinition of the

UCF area while eaing the process of respecting the guidelines of the reconfiguration flows and the architectural details of the particular FPGA model, so that a correct set of constraints can be generated. In this case, this UCF editor has been used to generate a new constraint and to put it in the UCF file for another synthesis process of the architecture. Figure 42 illustrates the proposed SLICE constraint while in editing mode. Upon committing the changes to

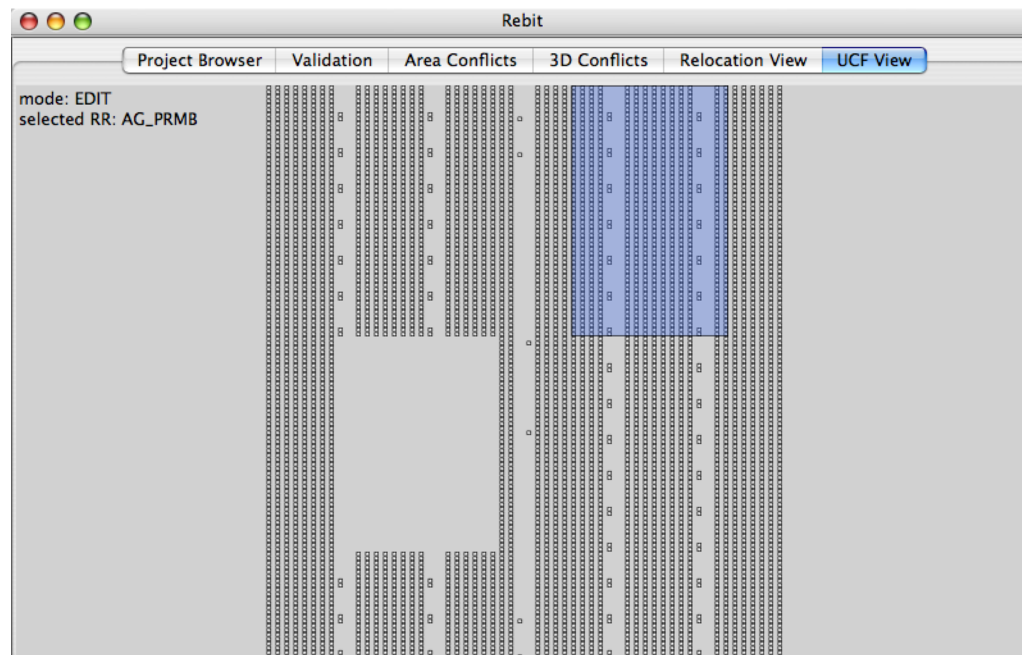

Figure 42. Screenshot of the editing of the reconfigurable region (in blue).

the area, the RPM grid function to list included resources is called, in order to generate all of

the necessary constraints to include all the resources encompassed by this newly defined SLICE bound into the UCF file. The resulting constraints now are:

```
INST "addsub_B" AREA_GROUP = "AG_PRMB";
AREA_GROUP "AG_PRMB" RANGE = SLICE_X28Y72:SLICE_X41Y127;
AREA_GROUP "AG_PRMB" RANGE = DSP48_X0Y18:DSP48_X0Y31;
AREA_GROUP "AG_PRMB" RANGE = RAMB16_X2Y9:RAMB16_X2Y15;
AREA_GROUP "AG_PRMB" RANGE = FIFO16_X2Y9:FIFO16_X2Y15;
AREA_GROUP "AG_PRMB" MODE  = RECONFIG;
```

And a new validation phase now reports the results in Figure 43: left and right edges are now on correct boundaries, the resource inclusion is correct, and the only warnings given by the program are generated by the bitstream configuration granularity issues explained at the beginning of this section. From the validation results, in fact, it is clear how the horizontal occupation of both bitstreams is included in the proposed reconfigurable region.

6.3 Area conflicts display

The conflict panel with only two partial bitstreams in the same RR displays portion of the incidence matrix of the conflict graph. This situation is illustrated in figure 44. The red square is the lowest left cell of the incidence matrix of the conflict graph, and the red color indicates a conflict between *edge* and *greysc* RFUs. The log of the program reports the calculated conflicts, thus verifying that the only possible solution in this case is to time share the resources defined in the reconfigurable region in order to use both functionalities in the application, so the configuration of the application will be first the static part with the

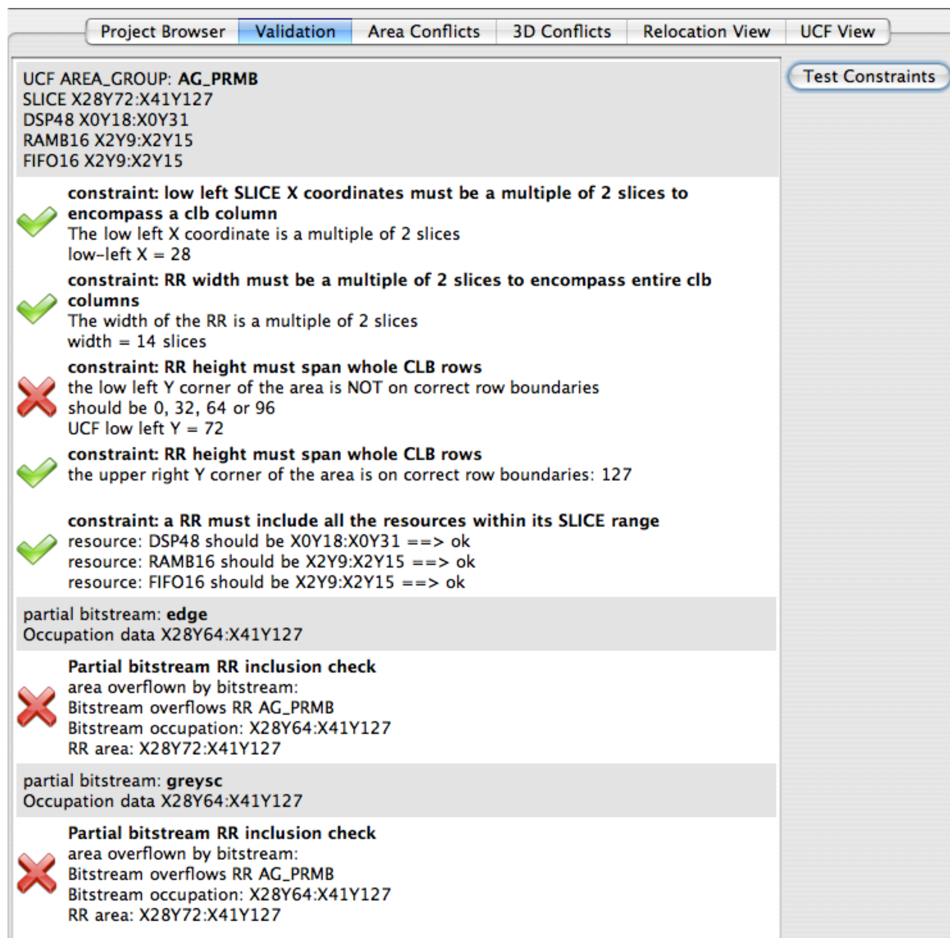

Figure 43. Screenshot of the validation results after the alteration of the UCF constraints for the RR of the case study.

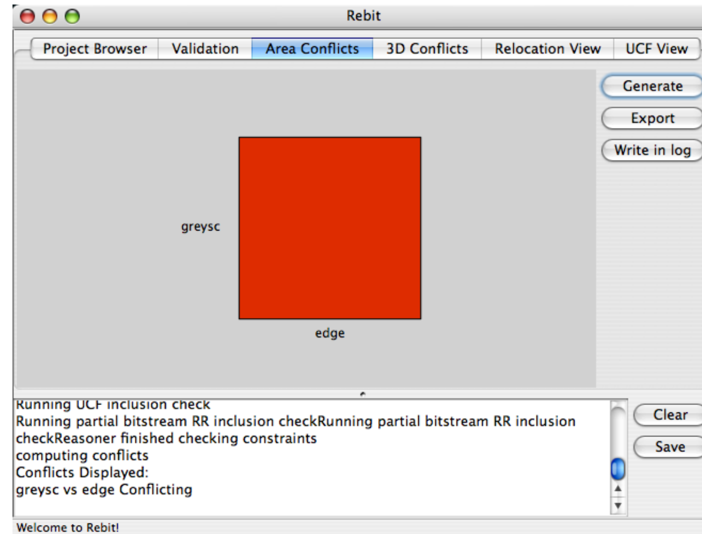

Figure 44. Screenshot of the Area Conflicts pane after the computation of the conflicts between the two partial bitstreams of the project.

grey scale conversion RFU loaded, then the same static part with the edge detection module configured in the RR previously defined. This version of the application is the one that has been proposed in (Mattasoglio et al., 2006 2007) and constitutes the final version submitted in that work, which successfully validated the proposed architecture on a development board mounting the XC4VFX12 FPGA.

6.4 Exploration of the relocation possibilities

A first step for exploring the relocation possibilities in this case study is to apply the feasible positions algorithm illustrated in the previous chapter to compute the possibilities of relocation for a partial bitstream in this case study. In this case the edge detection partial

bitstream has been selected in the Relocation View pane of the program, and the positions have been computed and displayed on the canvas. Figure 45 reports a screenshot of the computed positions given by the algorithm. Three positions have been found by the algorithm. These

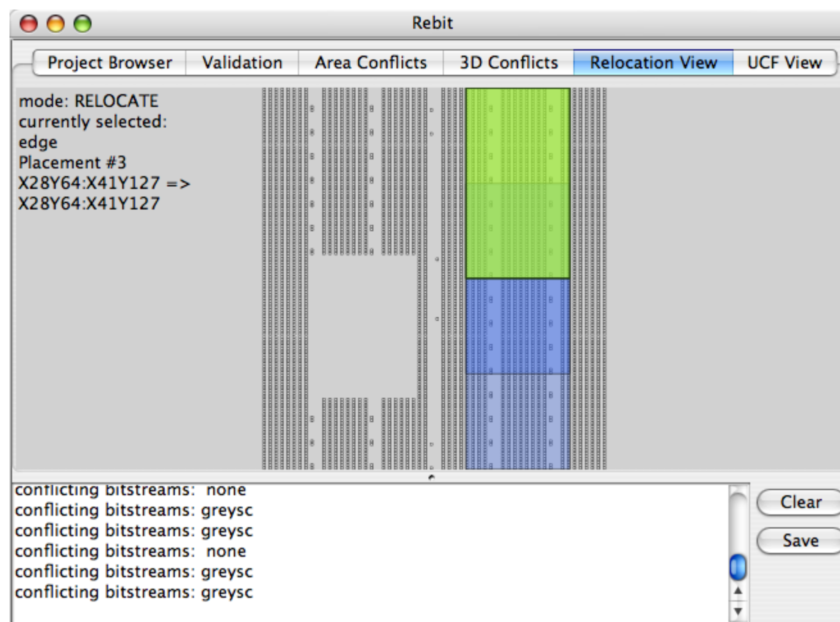

Figure 45. Screenshot of the Relocation View pane after the computation of the feasible positions for the edge detection partial bitstream.

positions are guaranteed to have all of the resources encompassed by the original placement of the partial bitstream on the FPGA array, and are thus feasible for a relocation. In the screenshot the original placement is highlighted in green, while the two other placements are

visible underneath. The user in this view can cycle through the proposed placement, and display the generated conflicts with other partial bitstreams in the log of the program.

To correctly support relocation, the area in the UCF file has been changed as illustrated before in order to span exactly two rows of the device and thus eliminating the warnings given by bitstream configuration row granularity. Moreover, seeing the feasible relocation positions, another reconfigurable region has been created in the UCF file to place a second RFU in the bottom half of the device, since the occupation data of the global system as seen in Figure 37 shows that the device utilization is not maximized. In this way two slots are available for the placement of RFUs in the application execution, allowing to explore new possibilities other than time-sharing the device resources between two different functionalities.

This new situation can allow different implementations of both the edge detection filter and of the gray scale conversion module to be implemented, and to eliminate the reconfiguration time by having them both configured at the same time at the beginning of the application execution. In this way for example a different customization of the grey scale image processing function can be configured while the edge detection filter is still transforming the image in the other RR, thus trading the time sharing capability offered by partial dynamic reconfiguration for an increased flexibility of the system. The adaptation of the application to use different algorithms in response to different data can be thus achieved following this model. Figure 46 illustrates the two SLICE areas defined in this new UCF file. The two areas are identical and allow the relocation of a partial bitstream from the top area to the bottom one. Two different partial bitstreams have been added to the project for both the grey scale conversion filter and

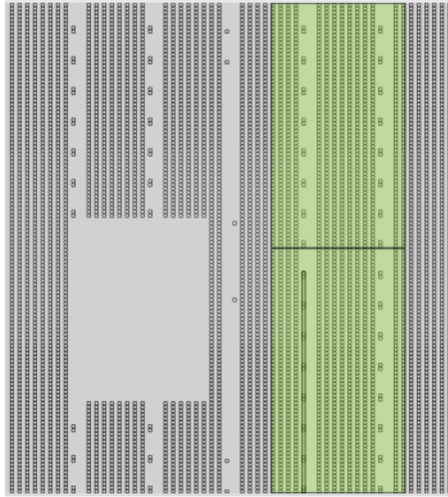

Figure 46. Screenshot of the UCF View pane with the representation of the two reconfigurable regions proposed.

for the edge detection filter, representing two different implementations for each one. In this case a new synthesis process has to take place to all of the RFUs, taking into consideration the advantage of being able to relocate a RFU from the top to the bottom half of the device and vice versa. The static photos of the system, representing the configured functionalities in a given time frame, have been loaded into the framework. Two application scenarios have been created, one with a first implementation of both filters, another with a second version of the same filters.

The original placement of the RFUs is in the top half of the device, thus the conflict graph is totally connected, because every partial bitstream is in an area conflict with each other. To solve these conflicts, the relocation computation can be exploited and the edge detection filters

can be placed in the bottom half of the device. Monitoring the conflicts in the Area Conflict pane now shows that all the combinations where an edge detection filter is configured at the same time with a grey scale filter are possible, thus allowing the global computation of a pass on an image not to be affected by the reconfiguration time needed to switch between two different RFUs.

Figure 47 illustrates the three steps for relocation: the first step is to view the conflicts between the RFUs in the Area Conflicts pane (a). In this case there are conflicts between functionalities that have to be configured at the same time on the chip, so the Relocation View offers the capability to relocate the partial bitstreams while monitoring in the log the situation of the area conflicts with the bitstream being currently relocated. Once every change has been committed in the Relocation View the conflict graph can be generated again on the basis of the new area occupation data, and viewed again in the Area Conflicts pane (c).

As a further support for the relocation exploration, the 3D Conflicts pane shown in figure 48 offers a browsable view of the application static photos and of the conflicts involved in each of them. In the proposed case, all of the conflicts have been resolved for the four static photos devised for the system:

```
0 edge_a.bit

0 greysc_a.bit

1 edge_b.bit

1 greysc_b.bit

2 edge_a.bit
```

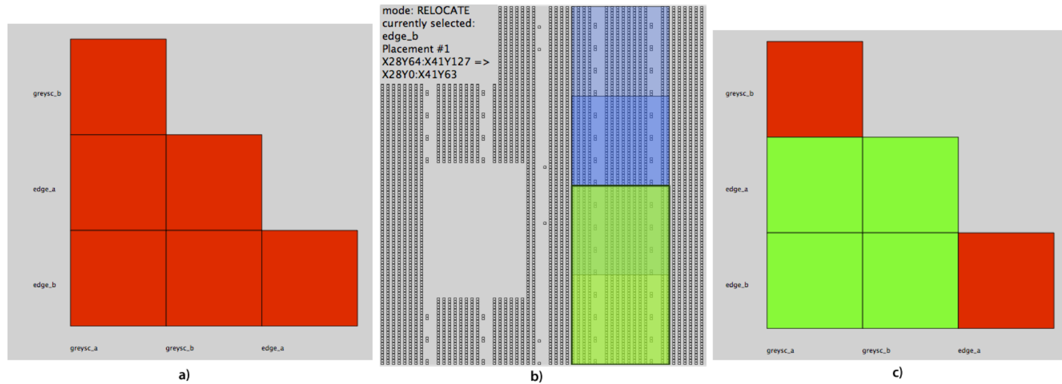

Figure 47. Three screenshots of the program showing three phases of the framework utilization in the relocation of partial bitstreams: the conflict graph prior to relocation (a), the relocation view showing how the placement is changed from top to bottom half (b) and the resulting conflict matrix (c).

```
2 greysc_b.bit

3 edge_b.bit

3 greysc_a.bit
```

The absence of red squares in the four incidence matrices displayed in Figure 48 indicates that no conflict is present in the evolution of the application and that all of the combinations of RFUs defined by the static photo description are feasible. The flattened 3D view corresponds to a subset of the final conflict matrix shown in Figure 47, in particular to the subset composed of only green squares.

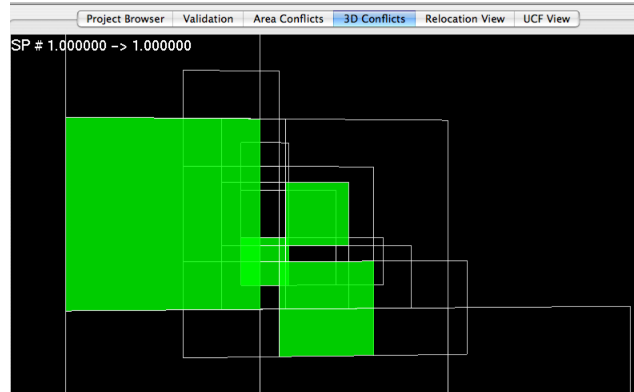

Figure 48. The 3D browser for the static photos implemented in the program showing the filtered conflicts on the basis of the four static photos supplied. Each layer in the 3D space has the incidence matrix of the conflict graph filtered by a particular static photo.

6.5 Conclusions

This chapter has shown how the Rebit framework can be used to validate, debug and explore new design solutions for an example of a real PDR architecture. The resolution of malformed RR constraints in the UCF file has been demonstrated with the utilization of the reasoner module of the program in conjunction with the UCF area group editor offered by the UCF view pane. The exploration of bitstream relocation and the proposal for a more complex architecture based on the case study that has been analyzed in this chapter has then been introduced, showing how the combination of the facilities for bitstream relocation computation and area conflict analysis have been used to devise a new system that trades in efficient area utilization for an enhanced flexibility of the application in the adaptation to different scenarios

and data. It is thus clear how the features often required by PDR architectures, as exposed in chapter 2 can be evaluated according to a particular application and the trade offs involved in both the selection of a particular device and in the utilization of PDR techniques can be viewed and taken into account more easily by the system designer, rising the level of development from having to manually inspect the design for constraint violation and having to manually produce constraints for the definition of reconfigurable regions to a visual, editable and architecture-aware representation of the relevant information of the design.

CHAPTER 7

CONCLUDING REMARKS AND FUTURE WORK

The aim of this work has been to explore the low level configuration details of the bitstream files resulting from the flows for Partial Dynamic reconfigurable architectures and to merge the information recovered from these files in order to aid the developers of such architectures to debug their designs, since often the only way to test the effectiveness of a partial dynamic reconfigurable architecture is to deploy the system on the FPGA itself.

The goal of this work is that the developed framework will be a real aid in the debugging of reconfigurable systems, by letting the developers concentrate more on the application itself than on the details of the flows, preventing possible errors and give an automated solution for the verification of the constraints and the guidelines that are involved in any flow for partial dynamic reconfiguration.

Moreover, the possibilities offered by bitstream relocation have been taken into account to give the developer a possibility to explore different design solution in the modification of the application schedule and the resolution of the possible conflicts between different reconfigurable functionalities even after the design has been synthesized. This has been possible by studying the internal model of the Xilinx FPGAs and finding a system to encode resource data in such a way to store both the available resources of the chip and to store the spatial relationships between each other.

From this model it is possible to have a convenient way to access the configuration structure

of a Xilinx device in a program, transferring the information available in the vendor's software suite into other works and thus allowing the creation of novel frameworks that can exploit this information. This gives way to two main future works that could arise from the present one.

A first opportunity is the addition of new devices and families to the framework: provided the underlying architectural description of new FPGA devices does not change radically, it will be possible to integrate into the framework the parsers for the bitstream configuration files for a new family, the feature description of new FPGA devices and their detailed configuration array structure by means of their RPM grid.

A second opportunity for future works is given by the exploitation of the data model created in this framework for the constraint checking of new partial dynamic reconfiguration flows, and in general to integrate new methodologies that must rely on the low level configuration structure of a Xilinx FPGA device. Finally, application dependent tools can be created from the basis of the proposed framework, in order to support the analysis of the bitstream files in relation to the higher level area definition in order to support application specific constraint checking.

BIBLIOGRAPHY

Bobda, C.: Introduction to Reconfigurable Computing Architectures, Algorithms, and Applications. Springer, 2007.

Castillo, J., Huerta, P., López, V., and Martínez, J. I.: A secure self-reconfiguring architecture based on open-source hardware. In International Conference on Reconfigurable Computing and FPGAs (ReConFig'05), 2005.

Chaubal, A. P.: Design and Implementation of an FPGA-based Partially Reconfigurable Network Controller. Master's thesis, Virginia Polytechnic Institute and State University, 2004.

Corbetta, S., Sciuto, D., and Santambrogio, M. D.: BAnMaT Light: creazione di un framework per il supporto alla rilocazione software dei bitstream. Master's thesis, Politecnico di Milano, 2005 - 2006.

Danne, K., Bobda, C., and Kalte, H.: Run-time exchange of mechatronic controllers using partial hardware reconfiguration.

Estrin, G.: Organization of computer systems–the fixed plus variable structure computer. Proc. Western Joint Computer Conf., Western Joint Computer Conference, New York, pages 33–40, April 1960.

Gokhale, Maya, Graham, and S., P.: Reconfigurable Computing Accelerating Computation with Field-Programmable Gate Arrays. Springer, 2005.

Guccione, S., Levi, D., and Sundararajan, P.: Jbits: Java based interface for reconfigurable computing. In SPIE Proceedings, volume 3526, 1998.

Karanam, R. K., Ravindran, A., Mukherjee, A., Gibas, C., and Wilkinson, A. B.: Using fpga-based hybrid computers for bioinformatics applications. Xcell journal, 2006.

Mattasoglio, D., Antola, A. M., and Santambrogio, M. D.: Sistema basato su Evolvable Hardware per il riconoscimento dei contorni in immagini digitali. Master's thesis, Politecnico di Milano, 2006-2007.

Note, J.-B. and Rannaud, É.: From the bitstream to the netlist, 2007.

Rana, V., Santambrogio, M. D., and Sciuto., D.: Dynamic reconfigurability in embedded system design. In IEEE International Symposium on Circuits and Systems, pages 2734–2737. IEEE, May 2007.

Tseng, C. W.: XAPP452 Spartan-3 Advanced Configuration Architecture. Xilinx Inc., 1.0 edition, December 2004.

eds. N. S. Voros and K. Masselos System Level Design of Reconfigurable Systems-on-Chip. Springer, 2005.

Wade, B.: Application Notes 416. Using an RPM Grid Macro to Control Block RAM-to-FF Timing. Xilinx Inc., August 2002.

Xilinx Inc.: Virtex-II Pro Data Sheet Virtex-II Pro™ Platform FPGA Data Sheet, 2003.

Xilinx Inc.: Application Notes 290. Two Flows for Partial Re-configuration: Module Based or Small Bit Manipulations. San Jose, California, 2004.

Xilinx Inc.: Development System Reference Guide. Xilinx Inc., 8.2i edition, 2005.

Xilinx Inc.: Early Access Partial Reconfiguration Guide. Xilinx Inc., 2006.

Xilinx Inc.: Spartan-3 FPGA Family: Complete Data Sheet. Xilinx Inc., 2007.

Xilinx Inc.: UG012 Virtex-II Pro FPGA User Guide. Xilinx Inc., 4.2 edition, November 2007.

Xilinx Inc.: UG071 Virtex-4 Configuration Guide. Xilinx Inc., 1.9 edition, October 2007.

Xilinx Inc.: Virtex-4 Family Overview. Xilinx Inc., 2007.