Design Flow to Support Dynamic Partial Reconfiguration on Maxeler
Architectures

BY
MATTEO MASTINU
Laurea, Politecnico di Milano, Milano, Italy, 2010

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, 2013

Chicago, Illinois

Defense Committee:
Piotr J. Gmytrasiewicz, Chair and Advisor
Zhichun Zhu, University of Illinois at Chicago
Marco D. Santambrogio, Politecnico di Milano
To my family
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 High Performance Computing</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Reconfigurable Computing</td>
<td>7</td>
</tr>
<tr>
<td>2 CONTEXT DEFINITION</td>
<td>13</td>
</tr>
<tr>
<td>2.1 FPGAs: An overview</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1 FPGAs Structure</td>
<td>14</td>
</tr>
<tr>
<td>2.1.2 Basic Principles</td>
<td>16</td>
</tr>
<tr>
<td>2.1.3 Partial Reconfiguration</td>
<td>20</td>
</tr>
<tr>
<td>2.1.4 Lower level details</td>
<td>23</td>
</tr>
<tr>
<td>2.1.4.1 Advantages</td>
<td>26</td>
</tr>
<tr>
<td>2.1.4.2 Limitations</td>
<td>28</td>
</tr>
<tr>
<td>2.2 Maxeler Architecture</td>
<td>30</td>
</tr>
<tr>
<td>2.2.1 Hardware point of view</td>
<td>31</td>
</tr>
<tr>
<td>2.2.2 MaxWorkstation</td>
<td>33</td>
</tr>
<tr>
<td>2.2.2.1 MAX3 Board</td>
<td>33</td>
</tr>
<tr>
<td>2.2.3 Potential advantages of the Partial Reconfiguration support</td>
<td>38</td>
</tr>
<tr>
<td>2.3 Dataflow Programming</td>
<td>39</td>
</tr>
<tr>
<td>2.4 Summary</td>
<td>43</td>
</tr>
<tr>
<td>3 RELATED WORKS</td>
<td>45</td>
</tr>
<tr>
<td>3.1 FPGA related works</td>
<td>45</td>
</tr>
<tr>
<td>3.2 Maxeler related works</td>
<td>50</td>
</tr>
<tr>
<td>4 STARTING SOLUTIONS</td>
<td>56</td>
</tr>
<tr>
<td>4.1 Maxeler Software Environment</td>
<td>56</td>
</tr>
<tr>
<td>4.1.1 Preliminary analysis</td>
<td>58</td>
</tr>
<tr>
<td>4.1.2 Compiling</td>
<td>60</td>
</tr>
<tr>
<td>4.1.3 Manager Compiler</td>
<td>62</td>
</tr>
<tr>
<td>4.1.4 Kernel examples</td>
<td>66</td>
</tr>
<tr>
<td>4.2 Xilinx Flow</td>
<td>68</td>
</tr>
<tr>
<td>4.2.1 Partial Reconfiguration case</td>
<td>70</td>
</tr>
<tr>
<td>4.3 Summary</td>
<td>75</td>
</tr>
<tr>
<td>5 PROPOSED APPROACH</td>
<td>76</td>
</tr>
<tr>
<td>5.1 Current system analysis</td>
<td>76</td>
</tr>
<tr>
<td>5.1.1 MaxCompiler proposed modification</td>
<td>79</td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Conclusions ..............................</td>
</tr>
<tr>
<td>6</td>
<td>PROPOSED IMPLEMENTATION ........................</td>
</tr>
<tr>
<td>6.1</td>
<td>User interface ..................................</td>
</tr>
<tr>
<td>6.1.1</td>
<td>DFE code .....................................</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Host code .....................................</td>
</tr>
<tr>
<td>6.2</td>
<td>Design flow implementation ....................</td>
</tr>
<tr>
<td>6.2.1</td>
<td>First Stage ...................................</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Second Stage ..................................</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Final stage ....................................</td>
</tr>
<tr>
<td>6.2.4</td>
<td>State Machine ..................................</td>
</tr>
<tr>
<td>6.2.5</td>
<td>File system ....................................</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary ........................................</td>
</tr>
<tr>
<td>7</td>
<td>VALIDATION .....................................</td>
</tr>
<tr>
<td>7.1</td>
<td>The application: Canny edge detection .............</td>
</tr>
<tr>
<td>7.2</td>
<td>Implementation ..................................</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Structure overview .....................</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Host Code .....................................</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Hardware Code ..................................</td>
</tr>
<tr>
<td>7.2.3.1</td>
<td>GrayScale Kernel ..................................</td>
</tr>
<tr>
<td>7.2.3.2</td>
<td>Gaussian Blur Kernel ..........................</td>
</tr>
<tr>
<td>7.2.3.3</td>
<td>EdgeLaplace Kernel ............................</td>
</tr>
<tr>
<td>7.2.3.4</td>
<td>Threshold Kernel ..............................</td>
</tr>
<tr>
<td>7.2.3.5</td>
<td>Manager ........................................</td>
</tr>
<tr>
<td>7.2.3.6</td>
<td>Builder ........................................</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Considerations ..................................</td>
</tr>
<tr>
<td>8</td>
<td>CONCLUSIONS ....................................</td>
</tr>
<tr>
<td></td>
<td>CITED LITERATURE ................................</td>
</tr>
<tr>
<td></td>
<td>VITA ...........................................</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>4</td>
</tr>
<tr>
<td>II</td>
<td>32</td>
</tr>
<tr>
<td>III</td>
<td>34</td>
</tr>
<tr>
<td>IV</td>
<td>50</td>
</tr>
<tr>
<td>V</td>
<td>51</td>
</tr>
<tr>
<td>VI</td>
<td>54</td>
</tr>
<tr>
<td>VII</td>
<td>54</td>
</tr>
</tbody>
</table>

I COMPARISON OF THE IANUS ARCHITECTURE WITH A STANDARD PC

II PROGRAMMING LANGUAGES

III FPGA\textsc{s} COMPARISON

IV MAX2 VS TESLA

V INTEL NEHALEM E5520 QUAD-CORE CPU

VI MAX\textsc{node}-1821 VS 8 CORE XEON SERVER SPEEDUP

VII POWER USAGE FOR 1U COMPUTE NODES
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Evolution of Intel processors. CPU frequency has stalled, and the gain in performance is due to parallelism[^1].</td>
</tr>
<tr>
<td>2</td>
<td>IANUS architecture. A 4 × 4 grid of processing elements (Xilinx FPGAs) communicates with the host computer.</td>
</tr>
<tr>
<td>3</td>
<td>The reconfigurable structure of an FPGA.</td>
</tr>
<tr>
<td>4</td>
<td>Arrangement of Slices within the CLB. The two slices do not have direct connections to each other, and each Slice is organized as a column. Each Slice in a column has an independent carry chain.</td>
</tr>
<tr>
<td>5</td>
<td>Comparison between 1D and 2D placement constraints and 1D and 2D reconfiguration.</td>
</tr>
<tr>
<td>6</td>
<td>Communication problem between two different configurations.</td>
</tr>
<tr>
<td>7</td>
<td>Partial Reconfiguration scenario.</td>
</tr>
<tr>
<td>8</td>
<td>In a Reconfigurable block inside the FPGA it is possible to place multiple Reconfigurable Modules, defined by their relative partial bitstreams.</td>
</tr>
<tr>
<td>9</td>
<td>Comparison of the FPGA state when loading a full BIT file (a) and a partial BIT file (b).</td>
</tr>
<tr>
<td>10</td>
<td>Maxeler dataflow system architecture. A General Purpose Central Processing Unit (GPCPU) is connected to a DataFlow Engine (DFE). Both of them have access, independently, to Random Access Memory (RAM).</td>
</tr>
<tr>
<td>11</td>
<td>MaxWorkstation architecture. The &quot;Reconf&quot; area represents the VHDL state machine responsible for configuring the CFPGA.</td>
</tr>
<tr>
<td>12</td>
<td>max_load(MaxFile,device_name) call.</td>
</tr>
<tr>
<td>13</td>
<td>Comparison between (a) traditional control-flow and a dataflow engine (b). The code defining the dataflow engine is translated in a graph of nodes representing very simple operations (dataflow cores). Data flow through the dataflow cores, which perform the operation they represent as soon as their inputs are available. Data are written back to memory only when the entire DFE has been crossed.</td>
</tr>
<tr>
<td>14</td>
<td>Graph representing a Kernel which outputs 255 if the input stream value is above 10, 0 in the other case.</td>
</tr>
<tr>
<td>15</td>
<td>MaxNode-1821 architecture diagram.</td>
</tr>
<tr>
<td>16</td>
<td>Interaction among the software component in the Maxeler architecture.</td>
</tr>
<tr>
<td>17</td>
<td>Compilation flow in the MaxCompiler. The back-end compilation represents the Xilinx tool chain.</td>
</tr>
<tr>
<td>18</td>
<td>Application's Streams. The data flow sequentially through the four Kernels, and finally go back to the host application.</td>
</tr>
<tr>
<td>FIGURE</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>19</td>
<td>Moving average Kernel code and graph.</td>
</tr>
<tr>
<td>20</td>
<td>Xilinx flow, from the VHDL or Verilog code to the bitstream.</td>
</tr>
<tr>
<td>21</td>
<td>Design Hierarchy: a &quot;top&quot;, static, partition contains two static partitions and two reconfigurable partitions.</td>
</tr>
<tr>
<td>22</td>
<td>Partial Reconfiguration software flow. Starting from the modules' definition, all the bitstreams are obtained through the process of synthesis.</td>
</tr>
<tr>
<td>23</td>
<td>Bitstreams generation: for each Reconfigurable Block there is a partial bitstream, and a full bitstream describes the static part.</td>
</tr>
<tr>
<td>24</td>
<td>In this example there is one static (top-level) partitions and three reconfigurable ones, arranged in three Configurations. In the first phase, they are synthesized by calling XST and NGCBuild. In the second phase NGDBuild, MAP, PAR and Bitgen are called, to produce the full bitstreams and all the partial ones.</td>
</tr>
<tr>
<td>25</td>
<td>Monolithic Design VS Modular Design. A large Kernel can be broken up in multiple Kernels of smaller dimension, which, put together, perform the same functionality.</td>
</tr>
<tr>
<td>26</td>
<td>Clock Region in a Xilinx XC6VSX475T, the board employed as CF-PGA in the MAX3 DFE.</td>
</tr>
<tr>
<td>27</td>
<td>DFE flow from the user perspective.</td>
</tr>
<tr>
<td>28</td>
<td>File system example, with relative building phase on the right.</td>
</tr>
<tr>
<td>29</td>
<td>Comparison between the original image (a) and the same image after the Canny edge detector filter has been applied (b). In the image (b) all the edges of the original image have been strongly emphasized.</td>
</tr>
<tr>
<td>30</td>
<td>Demo application flow.</td>
</tr>
<tr>
<td>31</td>
<td>Conversion to scale of gray of the Lena image.</td>
</tr>
<tr>
<td>32</td>
<td>Image (a) shows a generic function, in this case the arctangent function. Image (b) corresponds to the first derivative, with shows a maximum in the gradient when the x coordinate is equal to 0. The image (c) represents the second derivative (that it is possible to obtain by applying the Laplace operator), and it crosses the x axis in the same point of the maximum of the first derivative.</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface.</td>
</tr>
<tr>
<td>APU</td>
<td>Accelerated Processing Unit.</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange.</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit.</td>
</tr>
<tr>
<td>AVI</td>
<td>Audio Video Interleave.</td>
</tr>
<tr>
<td>CFPGA</td>
<td>Compute FPGA.</td>
</tr>
<tr>
<td>CLB</td>
<td>Combinatorial Logic Blocks.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit.</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Checking</td>
</tr>
<tr>
<td>CTR</td>
<td>Compile Time Reconfigurable.</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access.</td>
</tr>
<tr>
<td>DRC</td>
<td>Design Rule Check.</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array.</td>
</tr>
<tr>
<td>FWI</td>
<td>Full Waveform Inversion</td>
</tr>
<tr>
<td>GPCPU</td>
<td>General Purpose Central Processing Unit.</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics Processing Units.</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS (Continued)

**GPU** Graphics Processing Units. xi, 51

**GUI** Graphical User Interface. 56

**HDL** Hardware Description Language. xiii, 68, 69

**HPC** High Performance Computing. xii, xiii, 1, 6, 13, 56

**I/O** Input/Output. 16, 28

**IC** Integrated Circuit. 13, 15

**ICAP** Internal Configuration Access Port. 17, 20, 22

**IFPGA** Interface FPGA. 33, 43, 80, 81, 116

**IP** Intellectual Property. 65

**ISE** Integrated Synthesis Environment. 75

**JTAG** Joint Test Action Group. 20

**LC** Logic Cell. 16

**LUT** Look-Up Table. 15, 49

**MM** Mapped Memories. 115

**MR** Mapped Registers. 115

**NCD** Native Circuit Description. 69

**PC** Personal Computer. xi, 4, 16

**PCF** Physical Constraints File. 69

**PCI** Peripheral Component Interconnect. 35, 37, 43, 62
LIST OF ABBREVIATIONS (Continued)

PR  Partial Reconfiguration. xii–xiv, 1, 11, 12, 27, 29, 38, 43, 45, 49, 55, 76, 77, 79–83, 94, 98, 112–116

PROM  Programmable Read-Only Memory. 13

RAM  Random Access Memory. vi, 31

RC  Reconfigurable Computing. xiii, 1, 9, 13

RPU  Reconfigurable Processor Units. xi

RTM  Real Time Migration. 52

RTR  Run-Time Reconfiguration. 46

SDRAM  Synchronous Dynamic Random-Access Memory. 33, 48

SLiC  Simple Live CPU. 57, 62, 83, 103

SO-DIMM  Small Outline Dual In-line Memory Module. 33

SoC  System-on-Chip. xi

TMR  Triple Modular Redundancy. 49

UCF  User Constraint File. 68, 71, 114

XML  Extensible Markup Language. 114

XST  Xilinx Synthesis Technology. 68
SUMMARY

Nowadays computing systems are becoming more and more complex, with huge differentiations among the segments which constitute the large spectrum of computing systems, from grid systems made of thousands of computing platforms spread all over the world (e.g. Amazon® Elastic Compute Cloud (EC2)\(^2\)) to the smallest portable devices, like tablets (Apple® iPad) and smartphones (Samsung® Galaxy S III\(^3\)). Personal Computer (PC) are characterized by a constant growth in complexity, so that they are now equipped with multi-core Central Processing Unit (CPU) with up to 6 (Intel® Core i7\(^4\)) and 8 (AMD® FX-8\(^5\)) cores or solutions where CPU and Graphics Processing Units (GPU) coexist in the same chip (e.g. AMD Accelerated Processing Unit (APU)\(^6\)). Even our portable devices are equipped with very powerful System-on-Chip (SoC) (e.g. NVIDIA® Tegra 3\(^7\), Apple® A6).

In such a scenario, where the quest for performance must be balanced with the need for low power consumption, Reconfigurable Hardware plays a key role. Systems based on reconfigurable hardware are able to modify their behavior after their deployment, by removing, adding, or changing the modules of which they are composed. In this way, both hardware and software can adapt to the environment and to the user needs. Moreover, by tailoring the hardware to the application, applications running on Reconfigurable Hardware architectures require often lower power consumption with respect to standard software solutions\(^8\).

Field-Programmable Gate Array (FPGA)\(^9\)\(^10\) are probably the most popular devices in the field of Reconfigurable Hardware: with their always growing improvements in terms of programmable components’ density and capacity (as for instance the Xilinx®\(^11\) Virtex-6 family and the Altera®\(^12\) Stratix V family) they are becoming more and more relevant in different types of applications, spacing from building SoC to employing them as co-processors in heterogeneous systems. Cray®\(^13\) proposed the XT5h super computer, based on a combination of
AMD Opterons CPU with FPGA-based Reconfigurable Processor Units (RPU), while Intel in 2010 proposed a system where a CPU and a FPGA are placed on the same die$^{[14]}$.

Another example of a company proposing a heterogeneous system based on the interaction between a CPU and a FPGA co-processor is Maxeler. Maxeler® Technologies Ltd.$^{[15]}$ is a company operating in the High Performance Computing (HPC) field, which develops both the hardware platforms and the software environment necessary to use their hardware platforms. Their systems are based on the interaction between a standard Intel CPU and a so called DFE: from a hardware point of view the DFE is the hardware board used to exploit hardware acceleration. The name DFE derives from the programming paradigm employed to write the code which runs on the DFE: Dataflow Programming paradigm. Dataflow Programming is substantially different from the Control Flow Programming paradigm used in most of the programming languages and computer architectures, and it is the most suited choice to accelerate a large class of algorithms$^{[16]}$.

Maxeler’s DFEs are composed of one or more Xilinx’s FPGAs, and they represent the portion of the system devoted to the actual HPC: for this reason, the Maxeler system can be viewed, at a lower level, as the result of the interaction between a CPU and one or more FPGAs.

This work addresses the Maxeler® Technologies Ltd. platforms, and the principal goal of this work is to design a new methodology to support Partial Reconfiguration in the Maxeler design flow. The concept of Partial Reconfiguration (PR)$^{[17]}$ enhances the flexibility and the possibilities offered by FPGAs: with PR it is possible to modify the behavior of a determined portion of the device while the rest of the board keeps running.

Maxeler design flow allows to configure the FPGAs contained in the DFE starting from the Java code describing the part of the application that must run on the DFE. To do that, Maxeler created MaxCompiler, a complex tool designed for translating the Java code in an optimized VHDL (which exploits as much as possible the parallelism inherent to a hardware implementation of the code) and to execute all the Xilinx tools that produce the FPGA con-
figuration file. But the Maxeler flow does not allow to specify a partially reconfigurable design: after the DFE has been configured its behavior remains fixed for all the application’s run-time. Supporting PR in the Maxeler flow would allow the design of more flexible project, where the DFE behavior can change at run time. But flexibility is not the only improvement brought by PR: its employment would allow to overcome the limitations due to the FPGAs limited capacity, by time-multiplexing\cite{18} the device.

There are only few frameworks and tools to exploit PR\cite{19}\cite{20}, and proposing a new methodology to exploit PR in Maxeler platforms, that have already demonstrated to guarantee great performance\cite{21}, would be an important step to boost the diffusion of PR in both the academic and the commercial world. This work goes beyond merely proposing a possible methodology: the entire work has been implemented in the MaxCompiler, proving the effectiveness of the entire design and solution. Furthermore, the work will be released by Maxeler in the new MaxCompiler releases, making it available to the public. To validate the work, a video filtering analysis application will be proposed, consisting of an edge-detector implemented in the Maxeler architecture, exploiting the newly supported Partial Reconfiguration.

The remainder of this dissertation is structured as follows. Chapter 1 introduces the premises behind the field in which the work operates, namely the field of Reconfigurable Computing (RC) used to exploit HPC. Chapter 2 defines the context of the work, deepening the description of FPGAs, presenting the Maxeler architecture and the Dataflow Programming paradigm. Chapter 3 is dedicated to showing the state of the art in the field of heterogeneous systems, and academic works where Maxeler platforms are employed to accelerate applications. Chapter 4 presents the starting point from which the proposed solution has been developed: the Xilinx tool flow, which allows to synthesize a Hardware Description Language (HDL) design and to produce the FPGA configuration file, and MaxCompiler. Chapter 5 presents the guidelines of the proposed solution, starting from the analysis of the current system and ending with the rationale behind the proposed solution. Chapter 6 shows how the solution proposed
in the previous has been implemented in the Maxeler system. Chapter 7 proposes a video filtering application designed to show the potentialities of PR in the Maxeler architecture, and that the proposed solution effectively works. Finally, Chapter 8 explains the limitations of the proposed approach and recommends some future works to improve the design flow.
CHAPTER 1

INTRODUCTION

In this work, two different but related fields meet each other: that of HPC and of RC. In fact, the platform this work targets is devoted to HPC, but its core part, the DFE, is composed of FPGA, that are among the most known devices belonging to the Reconfigurable Computing world. In the Maxeler architecture two different entities coexist, shaping a heterogeneous system: a CPU and a DFE. The DFE is the part of the system where the actual acceleration is exploited, and it consists of one, or more, Xilinx FPGAs. The nature itself of the Maxeler architecture is intrinsically bounded to the Reconfigurable Computing field, and this means that Maxeler systems can be seen as reconfigurable systems whose remarkably high potential is exploited employing them as high performance computing platforms.

The concept of HPC and that of RC are not always strictly tied together as in the Maxeler case: there are countless examples of HPC systems which are not based on reconfigurable hardware (e.g. the one based on General Purpose Graphics Processing Units (GPGPU), like the Cray XK7\[^{[22]}\]) and, similarly, there is a plethora of application fields of RC different from HPC.

The first section of this Chapter will define the concepts behind HPC by means of some practical examples, the second one will introduce RC, presenting the FPGA technology and giving a first definition of PR: anyway, it is essential to notice that the work fits in the field of HPC and in that of RC at the same time.

1.1 High Performance Computing

Modern Science strongly relies on the application of numerical techniques. When these techniques are applied to enormously large sets of data (as it often happens in many applica-
tions of physics, biology, chemistry and engineering) the progresses in Science are indissolubly
bounded to progresses in computer technology: the available computational power and com-
puter memory resources dictate both the scope of the scientific problems that can be targeted
and the achievable accuracy level given the same theoretical modeling. For this reason it is clear
how, without the development of new computational technology, with much higher performance
with respect to the actual ones, modern science would not be able to address more challenging
problems on a larger scale emulating more accurate theoretical models.\cite{16} In the last years it
has been acknowledged that it is possible no more to obtain significant gains in performance by
simply scaling processor frequency. Industry has moved from producing processors consisting
of single cores, characterized by an yearly growth in frequency, to the production of CPUs
consisting in more than one core, with no significant improvement in clock frequency. Figure 1
shows the trend in Intel processors performance with respect to their frequency in the recent
years.

This is not only due to a mere fact of performance, but also to that of power consump-
tion. Processors with high frequency tend to consume more energy than processors with lower
frequency, therefore is often convenient (in order to preserve energy consumption) to produce
processors made of more than ore core with modest frequency than just a single core with very
high frequency. The need for processors with low energy consumption is not inherent only to
portable devices (such as smartphones, tablets, laptops) but also to the servers world: in 2005,
the world’s estimated 27 million servers consumed 0.5\% of all the electricity produced on the
planet, percentage which grows to 1\% if we take into consideration also the energy for associ-
ated cooling and auxiliary equipments (\textit{e.g.} backup power, power conditioning, air handling,
lighting and chillers)\cite{23}.

Computer architectures used in modern scientific computing can be divided in into:

- General-purpose;
- Problem-specific.
Figure 1: Evolution of Intel processors. CPU frequency has stalled, and the gain in performance is due to parallelism\[1\].

The father of general-purpose computing can be considered J. Von Neumann, and his work dates back to 1945. Even nowadays general-purpose computing is the primary model of computation, despite the fact it is a 60 years old idea. Anyway, during the last decade, the scientific community has started to explore and to appreciate the still young field of problem-specific architectures to solve the most challenging computational problems. When using problem-specific architectures instead of adapting the application to the computer, the computer is adapted to the application: the special-purpose computers are optimized for the application’s particular requirements, giving significant performance speed-up. This means that in problem-specific computing the first step for the computer engineer is to decide which problem to target, then to find a possible solution for solving it, and finally to implement the solution in a customized computer architecture. The resulting computer architecture should possibly be very efficient.
at solving that specific problem, while it could be much less efficient (or totally unable) at solving different problems. On the other hand, a general-purpose computer presents a fixed architecture: the computer engineer has to implement the solution for the problem she is trying to solve by making use of the set of computational elements offered by that architecture. Resulting solution is often not as effective as in the case of a problem-specific architecture, but the advantage is that can be easily adapted to solve a different type of problems.

There is a good number of highly successful applications which run on dedicated special-purpose architecture in order to improve their performance with respect to the same application run in a general-purpose architecture, and two of them will be briefly presented.

In the work presented in [24] dedicated hardware is used to compute a Spin system, a discrete system whose variable, called spin, sit on a lattice\(^1\) of very large dimensions: for a lattice of \(48 \times 48 \times 48\) sites a traditional PC would take a simulation time of \(\simeq 10^4\) years. The authors propose a system, called IANUS composed of a \(4 \times 4\) grid of Xilinx Virtex4-LX160 and Virtex4-LX200 FPGAs, called processing elements (SP). The result is a many cores processor where each core performs the same algorithm on a subset of the spins. Figure 2 represents the IANUS architecture.

The resultant architecture showed incredible improvements in the performance with respect to the implementation in a PC (an Intel Core2Duo - 64 bit - 1.6 GHz processor), with an update spin for the entire IANUS board of 1 ps/spin.

<table>
<thead>
<tr>
<th>Update Rate</th>
<th>LX160</th>
<th>LX200</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 ps/spin</td>
<td>1 ps/spin</td>
<td>700 ps/spin</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\)A lattice is a partially ordered set in which any two elements have a supremum (also called a least upper bound or join) and an infimum (also called a greatest lower bound or meet).
Figure 2: IANUS architecture. A $4 \times 4$ grid of processing elements (Xilinx FPGAs) communicates with the host computer.

The architecture has therefore a processing power equivalent to $\geq 10000$ PCs making possible to carry out the simulation in about one year. Furthermore, the power dissipation is of $\approx 4KW$, a huge improvement with respect to a large PC farm.
In\textsuperscript{[25]} a special-purpose hardware is employed to quantum chromodynamics (QCD), a theoretical framework used to describe the properties and interactions of quarks and gluons, which are the building blocks of protons, neutrons, and other particles. The designed architecture is made of an IBM’s PowerXCell 8i multicore processor tightly coupled with a custom-designed network processor in which we connect each node to its nearest neighbors in a 3D toroidal mesh: each one of these nodes consists in a Xilinx Virtex 5 LX110T FPGA. PowerXCell 8i processor contains one PowerPC processor element and eight synergistic processor elements. Each SPE runs a single thread and has its own 256 Kbytes on-chip local store (LS) memory, which is accessible by direct memory access or local load/store operations to and from 128 general-purpose 128-bit registers. An SPE can execute two instructions per cycle, performing up to eight single-precision floating-point operations. Thus, the total single-precision peak performance of all eight SPEs on a Cell/B.E. is 204.8 Gflops at 3.2 GHz. The peak double-precision performance is 102.4 Gflops, and the memory bandwidth is 25.6 Gbytes/s. The entire QPACE installation comprises 4608 node cards, where each card is composed of:

- a PowerXCell 8i processor, which provides the computing power;
- a network processor (NWP), a Xilinx Virtex-5 LX110T FPGA, implements a dedicated interface to connect the processor to a 3D high-speed torus network used for communications between the nodes and to an Ethernet network for I/O.

The HPC platform used in this work is produced by Maxeler Technologies, and exploits the opportunities offered by \textit{Dataflow Computing}, a class of dedicated-purpose computers which are programmable and can be tailored to different applications and to different needs at different times. In some types of applications, dataflow computing offers many advantages\textsuperscript{[16]} with respect to “traditional” computing, such as:

- Orders of magnitude lower power consumption;
- Lower datacenter space requirements;
• Order of magnitude higher performance, in terms of speed.

A more detailed description of both Dataflow Computing and Maxeler platforms can be found in Chapter 2.

1.2 Reconfigurable Computing

The concept of reconfigurable computing has been around since the 1960 when Gerald Estrin, a computer scientist at the University of California Los Angeles, proposed a new computer architecture which combined some of the flexibility offered by the software with the high performance offered by hardware.

The result was a computer consisting of a standard processor augmented by an array of reconfigurable hardware\(^2\), where the goal of the processor was to manage the behavior of the reconfigurable hardware. The reconfigurable hardware is tailored to a specific task, and as soon as the assigned task is completed the reconfigurable hardware can be changed to perform some other task.

Before considering the specific area of the reconfigurable computing it is important to try to define what reconfiguration means. On\(^2\) it is possible to find a first interesting definition\(^3\):

**Definition 1.1. Reconfiguration:** The process of physically altering the location or functionality of network or system elements. Automatic configuration describes the way sophisticated networks can readjust themselves in the event of a link or device failing, enabling the network to continue operation.

Traditionally there are two main methods in computing for the execution of an algorithm.

The first one is to use an Application Specific Integrated Circuit (ASIC), or a board-level solution, to perform the computation in hardware. ASICs are tailored for the particular type of computation, and thus they are very fast and efficient. However, once they are produced, their functionalities cannot be modified: if the user needs to modify any of the parts of the
circuit, the entire chip must be redesigned. This is a very expensive process, especially in the case where there are many deployed devices to substitute. Moreover, producing an ASIC is a very expensive process due to the mask’s cost.

The second method consists in using software-programmable microprocessors: they can execute a given sequence of instructions to perform an operation, and by changing the software instructions it is possible to alter the resulting computations. The downside of this great flexibility is that performance often decreases sharply with respect to an ASIC implementation counterpart, at least at a work rate (the number of tasks the architecture is able to accomplish per unit of time) level. The processor must:

1. Read each instruction from memory;
2. Decode the instruction meaning;
3. Execute the instruction.

It is clear that this implies a large execution overhead. Furthermore, the basic set of available instructions is determined at fabrication time of the processor, and any other operation that is to be implemented must be built out of this base set.

Reconfigurable computing is intended to be midway between hardware and software, achieving much higher performance than software while being at the same time more flexible than hardware\cite{18}. In fact, many emerging products in communication, computing and consumer electronics demand that their functionality remains flexible also after the system has been manufactured and that is why the reconfiguration has to be considered into the design flow as a new relevant degree of freedom, in which the designer can have the system modify its functionalities according to the application changing needs. Differently from ordinary microprocessors it is possible to make substantial changes to the datapath itself in addition to the control flow, and differently from ASICs it is possible to adapt the hardware during runtime by ”loading” a new circuit on the reconfigurable fabric: the main difference between reconfigurable hardware
and microprocessors is the ability of the first to be substantially modified at datapath level in addition to the control flow. On the other hand, the main difference between reconfigurable hardware and custom hardware is the possibility for the first to be adapted at run time. This improvement in flexibility of reconfigurable hardware with respect to ASICs brings, as a consequence, a strong reduction in the costs of a reconfigurable hardware circuit with respect to an ASIC: very often an application would be best served by custom circuitry targeted specifically for it; and, in fact, ASICs are often made in response to special needs. But no one can afford to turn out a custom chip for every application she wants to run. As technology has improved, a market has grown up for versatile off-the-shelf parts that can be programmed to emulate arbitrary digital circuits in place of ASICs. FPGAs are one class of such devices, distinguished by their ability to be reprogrammed (reconfigured) any number of times.

For its nature, RC is a major discipline that covers various subjects of learning, including both computing science and electronic engineering. Reconfigurable computing involves the use of reconfigurable devices, such as Field Programmable Gate Arrays (FPGAs), for computing purposes. They contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements are connected by using a set of routing components, which are programmable as well. In this way it is possible to build custom digital devices, whose functionalities are not fixed but can change over time, by simply reconfiguring the device.

Reconfigurable computing systems often have impressive performance, exceeding that of their software implementation counterparts. Sheer speed, while important, is not the only strength of reconfigurable computing. Another compelling advantage is reduced energy and power consumption. In a reconfigurable system the circuitry is optimized for the application, so that performance per Watt will tend to be much lower than that for a general-purpose processor.
Other advantages of reconfigurable computing include a reduction in size and component count (and hence cost), improved time-to-market, and improved flexibility and upgradability. These advantages are especially important for embedded applications. Obviously, Reconfigurable Computing does not come with benefits only. The versatility and reprogrammability of FPGAs comes at a price. Only a few years ago, the algorithms that could be implemented in a single FPGA chip were fairly small. In 1995, for example, the largest FPGAs could be programmed for circuits of about 15,000 logic gates at most. Since a fast 32-bit adder requires a couple hundreds of gates, the capabilities of such devices were somewhat limited. More recently, though, FPGAs have reached a size where it is possible to implement reasonable sub-pieces of an application in a single FPGA part. The inclusion of reconfigurable array logic into a microprocessor provides an alternative growth path which allows application specialization while benefiting from the full effects of commoditization. Like modern reconfigurable logic arrays, a single microprocessor design can be employed in a wide variety of applications. Application acceleration and system adaptation can be achieved by specializing the reconfigurable logic in the target system or application\textsuperscript{[29]}. This has led to a new concept for computing: if a processor were to include one or more FPGA-like devices, it could in theory support a specialized application specific circuit for each program, or even for each stage of a program execution. The unlimited reconfigurability of an FPGA permits a continuous sequence of custom circuits to be employed, each optimized for the task of the moment. Because FPGAs scale better than superscalar techniques, such designs have the potential to make better use of continuing advances in device electronics in the long term\textsuperscript{[30]}.

Another disadvantage of using FPGAs is the time taken for reconfiguring the device: in time critical applications, if the FPGA needs to be reconfigured frequently the reconfiguration time can represent a big portion of the total application time. In fact, designs implemented on FPGAs need on average 40 times as much area, draw 12 times as much dynamic power, and
are three times slower than the corresponding ASIC implementations\textsuperscript{[31]}. A deeper coverage about reconfigurable computing and reconfigurable systems can be found in\textsuperscript{[18]} and\textsuperscript{[8]}.

One of the methods to address the problem of the limited FPGAs’ capacity is the \textit{Partial Dynamic Reconfiguration}. In \textit{partial} reconfiguration, only \textit{portions} of the reconfigurable device are involved by the configuration change. \textit{Dynamic} reconfiguration allows the device portions that are not directly involved in the reconfiguration to run without interruption through the reconfiguration process. A commonly adopted approach is the definition of predetermined area portions on the device, reconfigurable slots, in which components implementing different tasks, or modules, can be configured. PR provides the flexibility and run-time reconfigurability that neither pure hardware or software solutions can offer: it allows to \textit{multiplex} the hardware resource of the device, so that in different time it is possible to change the device’s functionality, without switching it off. The FPGAs ability to modify their configuration memory easily at any time is enhanced to a higher level by allowing to reconfigure only a particular portion of the chip configuration memory, and allowing the user to load or unload these functional hardware modules without interrupting or resetting the rest of the device. If an FPGA, after its deployment, becomes static, it is just a low performance and expensive ASIC. The possibility to remote updating it is a great advantage, because it can be seen as an equivalent to in-application programming for microprocessors and it used to dynamically tailor the hardware to the application’s needs in real time.

From a general point of view, as described in\textsuperscript{[32]}, PR can be performed following two different approaches: \textbf{module-based} or \textbf{difference-based}. The first one is the module-based approach, that is characterized by the division of reprogrammable devices in a certain number of portions, each one of which is called reconfigurable slot. In this scenario it is possible to reconfigure one or more reconfigurable slots with a hardware component that is able to perform a specific functionality, called modules. Obviously, the modules contained in slots that are not involved in the reconfiguration task do not have to stop during the reconfiguration process. The difference-
based approach does not require slots and modules definition, but it is suitable only when the differences between two configurations are very small since the process on which it is based is suitable only when little changes in the design are required. The most general design approach for dynamically reconfigurable embedded systems, as described in [33], is the modular-based design. This approach is the one taken into consideration in the present work.

As stated in [34], Partial Reconfiguration, by time-multiplexing the device, gives the designer the possibility to make use of an FPGA that virtually contains more resources than are actually present, providing multiprocessing across both time and space. Furthermore, this method not only reduces the total reconfiguration time (because it allows to reconfigure only a selected portion of the device, instead of the entire area) but also the total bitstreams size. PR allows to use more hardware than that the physically present in the FPGA, allowing to reduce its size. Furthermore, one of the effects of using a smaller portion of the FPGA is the reduction in power consumption. A more precise coverage about FPGAs and PR will be presented in Chapter 2.
CHAPTER 2

CONTEXT DEFINITION

The previous Chapter provided an introduction of the concepts of HPC and RC. Since this work targets a particular RC platform, the one produced by Maxeler Technologies, this Chapter will present the Maxeler architecture and the RC devices on which it is based. This Chapter is divided in four sections: the first one is dedicated to FPGAs and to the concept of Partial Reconfiguration; in the second one it will be presented an overview of the Maxeler architecture, focusing particularly on the system targeted by this work, the MaxWorkstation; the third section will present the concept of Dataflow Programming, necessary to understand how the DFEs work; finally, the fourth section will summarize this Chapter’s content.

2.1 FPGAs: An overview

A Field-Programmable Gate Array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing. This section, and the rest of the work, is based on the description of Xilinx® FPGAs, because Xilinx FPGAs are the ones used in the Maxeler boards. FPGAs from other vendors, such Altera® have similar characteristics. Among the Xilinx FPGAs this work will focus on the Virtex-6 family[^35], because is the FPGA family employed in the Maxeler platform taken into consideration.

FPGAs are the most common reconfigurable devices today. FPGAs are similar in principle to, but have a vastly wider potential application than, Programmable Read-Only Memory (PROM) chips. FPGAs are used by engineers in the design of specialized Integrated Circuit (IC)s that can later be produced hard-wired in large quantities for distribution to computer manufacturers and end users. Ultimately, FPGAs might allow computer users to tailor micro-
processors to meet their own individual needs.

2.1.1 FPGAs Structure

FPGAs are independently packaged parts marketed both as prototyping platforms and as reconfigurable alternatives to ASICs. The structure of an FPGA looks like Figure 3.

![Figure 3: The reconfigurable structure of an FPGA.](image)

The canonical logic block is often considered to be a look-up table that takes some fixed number of bits of input (six in the Virtex 6 case) and generates some fixed number of bits of output (two in the Virtex-6 case). By filling the table with the right bits, any logic function with that number of input bits can be realized: of course, the higher the number of input bits, the more powerful the blocks are. In the Virtex-6, a Combinatorial Logic Blocks (CLB) is
composed of two slices, and each slice is made of four Look-Up Table (LUT)s and their eight flip-flops as well as multiplexers and arithmetic carry logic$^{36}$.

Figure 4: Arrangement of Slices within the CLB. The two slices do not have direct connections to each other, and each Slice is organized as a column. Each Slice in a column has an independent carry chain.

Even a complex Xilinx Virtex-6 logic block is quite small compared to the usual functional units of a computer. But in large numbers, small logic blocks can add up to considerable computing power. The die sizes of the largest parts are generally at the boundary of what can be manufactured, but this of course is not true of the smaller parts, and the density is expected to grow strongly in the future$^{30}$. FPGAs are ICs that contains many, in many case more than
10,000, identical Logic Cell (LC)s that can be viewed as standard components. Each logic cell can independently take on any one of a limited set of personalities. The individual cells are interconnected by a matrix of wires and programmable switches. An user design is implemented by specifying the simple logic function for each cell and selectively closing the switches in the interconnect matrix. Complex designs are created by combining these basic blocks to create the desired circuit. Field Programmable means that the FPGA function is defined by an user program rather than by the manufacturer of the device. FPGA has three major configurable elements: CLBs, input/output blocks, and interconnects. The CLBs provide the functional elements for constructing user’s logic. The IOBs provide the interface between the package pins and internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks\[37\].

2.1.2 Basic Principles

FPGAs configuration capabilities allow a great flexibility in hardware design and, as a consequence, they make it possible to create a vast number of different reconfigurable systems. These can vary from systems composed of custom boards with FPGAs, often connected to a standard PC or workstation, to standalone systems including reconfigurable logic (rl) and general purpose processors (GPCPU), to System-on-Chip’s, completely implemented within a single FPGA mounted on a board, with only few physical components for INPUT/OUTPUT (I/O) interfacing. There are different models of reconfiguration, that can be classified according to the following scheme (an extension of the one proposed in\[38\]):

- *who* controls the reconfiguration;
- *where* the reconfiguration controller\(^1\) is located;
- *when* the configurations are generated;

\(^1\)the element that is responsible for the physical implementation of a reconfiguration process
• *which* is the granularity of the reconfiguration;

• *in what* dimension the reconfiguration operates.

The first subdivision (*who* and *where*) is between *external* and *internal reconfiguration*. In the first scenario, the reconfiguration is managed by an external entity, usually a PC or a dedicated processor. Internal reconfiguration, instead is performed completely within the FPGA boundaries; for this to be possible, the device must have a physical component dedicated to reconfiguration. This component will be referred as the *Internal Configuration Access Port (ICAP)* component in Xilinx FPGAs. It is important to stress that a scenario can be truly termed internal only when both the controller *and* the reconfiguration controller are placed inside the FPGA.

The generation of the configurations (*when*) can be done in a completely *static way* (at design time) by determining all the possible configurations of the system. Each module must be synthesized and all possible connections between modules and the rest of the system must be considered. Other possibilities are *run-time placement of pre-synthesized modules*, which requires dynamic routing of interconnection signal, or completely *dynamic modules generation*. This last option is currently impracticable, since it would require run-time synthesis of modules from VHDL (or other *hw* description language) code – a process requiring prohibitive times in an on-line environment.

Reconfiguration can take place at very different granularity levels (*which*), depending on the size of the reconfigured area. Two typical approaches are *smallbits* and *module based*: the first one consists in modifying a single portion of the design, such as single Configurable Logic Blocks (CLB) or I/O blocks parameters (as described in[^39]), while the second one involves the modification of a larger FPGA area by creating *hw* components (modules) that can be added and removed from the system: each time a reconfiguration is applied, one or more modules are linked or unlinked from the system.
The last property is the *dimension*. One can distinguish between two different possibilities: mono-dimensional (1D) and bi-dimensional (2D) reconfiguration. In a truly 2D reconfiguration it is possible to reconfigure an arbitrary portion of the FPGA without affecting the execution of the rest of the implementation. Most FPGAs, instead, require that in order to reconfigure a portion of a column of reconfigurable cells the whole column must stop its operations – the reconfiguration can act on a specific 2D portion of the column, but it still affects the execution of the *whole* column.

![Comparison between 1D and 2D placement constraints and 1D and 2D reconfiguration.](image)

The easiest way in which an FPGA can be reconfigured is called *complete*. In this case the configuration bitstream, containing the FPGA configuration data, provides information regarding the complete chip and it configures the entire FPGA, that is why this technique is
called complete. With this approach there are no particular constraints that have to be taken into account during the reconfiguration action. Obviously that does not mean that the designer is allowed whatever she wants just because she is using a configuration technique based on a complete-reconfiguration, in fact, if two different bitstreams implement two functionalities that have to work one after the other, see Figure 6 for an example of such a scenario, the designer has to take into account where to store the data between these two configurations.

Figure 6: Communication problem between two different configurations.

The main disadvantage of an approach based on the complete reconfiguration technique is the overhead introduced into the computation by the reconfiguration. In order to cope with this situation a partial reconfiguration approach has been proposed.
To configure an FPGA one (or more) bitstreams are needed. A bitstream is a binary file in which are stored informations concerning the configuration that the device ought to assume. A configuration memory stores system configuration, and it is possible to read and write to such memory trough a set of configuration registers, such as ICAP, SelectMAP, Joint Test Action Group (JTAG). The memory is implemented using volatile memory like SRAM cells, therefore, every time the device is powered off, the configuration is lost. If we want to modify the behavior of the FPGA we have to switch off the device, to connect it to a computer and to load a new bitstream. In order to allow more flexibility, the concept of Partial Reconfiguration has been introduced.

2.1.3 Partial Reconfiguration

Partial Reconfiguration is the process of changing portion of reconfigurable hardware circuitry while the other part is still running/operating, by loading a partial configuration file.

Partial reconfiguration is useful for applications that require the load of different designs into the same area of the device or the flexibility to change portions of a design without having to either reset or completely reconfigure the entire device\cite{10}. For current FPGA devices, data is loaded on a column-basis, with the smallest load unit being a configuration bitstream frame, which varies in size based on the target device. Active partial reconfiguration of Virtex devices, or simply partial reconfiguration, is accomplished in either slave SelectMAP mode or Boundary Scan, JTAG mode. Instead of resetting the device and performing a complete reconfiguration, new data is loaded to reconfigure a specific area of the device, while the rest of the device is still in operation\cite{41}.

The scenario shown in Figure 6 turns into the scenario prosed in Figure 7. Using an approach based on partial reconfiguration, as the one proposed in Figure 7, the basic idea is to partition the system in a set of functionalities \( f_1, f_2, \ldots, f_n \) able to produce a set of bitstreams \( b_1, b_2, \ldots, b_n \) that are not used to reconfigure the entire system but just a known portion of
it. The first bitstream is obviously a complete bitstream but the other functionalities are downloaded to reconfigure just portions of the architecture.

In order to be able to hide the reconfiguration time it is not only necessary to partition the FPGA to obtain the ability to compute partial reconfiguration bitstream, but it is also necessary to guarantee that a reconfiguration is not going to imply a standby of the logic of the FPGA which is not involved in the reconfiguration itself. Such a scenario brings us the definition of Dynamic Partial Reconfiguration.

**Definition 2.1.** *Dynamic Partial Reconfiguration* Dynamic partial reconfiguration is performed when the device is active. Except during some inter-design communication, certain areas of the device can be reconfigured while other areas remain operational and unaffected by the reprogramming.
Up to now reconfiguration has been defined from the area and the time prospectives, but there is still an important factor that can be used to classify a reconfigurable approach: the location of the controller of the reconfiguration. *External reconfiguration* implies that an active array may be partially reconfigured by an external device such as a Personal Computer, while ensuring the correct operation of those active circuits that are not being changed. *Self or Embedded Reconfiguration* extends the concept of dynamic partial reconfigurability. It assumes that specific circuits on the array are used to control the reconfiguration of other parts of the FPGA. Clearly the integrity of the control circuits must be guaranteed during reconfiguration, so by definition self-reconfiguration is a specialized form of dynamic reconfiguration\[^42\]. An important feature in FPGA architectures is the ability to reconfigure not only all the device but also a portion of it while the remainder of the design is still operational. Once initially configured, self-reconfiguration requires an internal reconfiguration interface that can be driven by the logic configured on the array. Starting with Xilinx Virtex II parts, this interface is called the internal configuration access port, *ICAP*\[^39\]. These devices can be configured by loading application specific data into the configuration memory which is segmented into frames, the smallest unit of reconfiguration. The number of frames and the bits per frame are different for the different devices of the Virtex-6 family. The number of frames is proportional to the CLB width of the device. The number of bits per frame is proportional to the CLB height of the device.

According to Xilinx, using Partial Reconfiguration should guarantee that:

- Modifying a region of the component does not affect the configuration memory of other, unmodified, regions;
- When the content of the configuration memory is overwritten by the same content, its corresponding logic can operate normally without being affected;
2.1.4 Lower level details

While the previous parts of the section focused on Partial Reconfiguration from a high level view, this sub-section will explore more technical details. The informations provided hereafter are useful to better grasp the solution that will be presented in this work.

Partial Reconfigurability is not supported on all FPGAs, and it requires to design the project in a hierarchical, modular way, using the design techniques called Partitioning. While a flat flow has the advantage of being able to optimize the entire design at once, a hierarchical design enables the designer to break up the design into smaller logical blocks, allowing each major function to be worked on independently.

In a hierarchical design flow, the design is broken up into blocks, referred as partitions. Partitions are used to define hierarchical boundaries or insulation around the hierarchical module instances, isolating them from other parts of the design. A partition that has been implemented and exported can be re-inserted into the design, preserving the placement and the routing results of the module instance. Partitions definitions and controls are specified in the xpartition.pxml file, read when the tools are run.

The file xpartition.pxml is basically an XML file defining the status of each partition, where the state correspond to the preservation level.
Listing 1: Example of a PXML file. There is a static module, imported from a previous configuration, and two reconfigurable ones.

In fact, the main purpose of the partition is to preserve the results of a run by importing previous results. In the PXML it is also possible to define the level of the preservation(routing/placement/synthesis/inherit).

Partitioning\[43\] is essential to Partial Reconfiguration because it allows to define different modules and to declare which one will be reconfigurable, and to make them independent from the rest of the design. Furthermore, it ensures that the logic and routing common to each of the
multiple designs is absolutely identical: once one design configuration meets all requirements, the designer can reuse the results from that implementation to create the other configurations. After all configurations are implemented, verification routines validate consistency among all the versions. All these checks are necessary to guarantee a safe environment when loading a partial bitstream into an operating FPGA.

The entire design can be seen as divided in two parts:

• A reconfigurable part, composed of all the Reconfigurable Areas;
• A static part, basically all the portion of the design that can not be reconfigured.

In this work, the reconfigurable portions of the FPGA will be called Reconfigurable Area, while Reconfigurable Module will refer to the reconfigurable logic that can be placed inside a Reconfigurable Area.

It is possible to look at the static part and to the reconfigurable part also from a hierarchical point of view:

• The static part is the top level part of the design. It contains a number of black box, where each black box correspond to a reconfigurable area;
• The reconfigurable modules are the bottom level part. They implement some functionalities that can be implemented inside the black boxes.

FPGA technology provides the flexibility of on-site programming and re-programming without going through re-fabrication with a modified design. Partial Reconfiguration (PR) takes this flexibility one step further, allowing the modification of an operating FPGA design by loading a partial configuration file, usually a partial BIT file: after a full BIT file configures the FPGA, partial BIT files can be downloaded to modify reconfigurable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured.
Figure 8: In a Reconfigurable block inside the FPGA it is possible to place multiple Reconfigurable Modules, defined by their relative partial bitstreams.

Figure 8 illustrates the premises behind Partial Reconfiguration.

In the example in Figure 8, the function implemented in Reconfig Block A (that in the top level design is seen as a black box) can be modified by downloading one or several partial BIT files, A1.bit, A2.bit or A3.bit. The grey area in the FPGA in the figure represents the static part, while the orange block represents the reconfigurable logic. When loading a partial BIT file, the static logic keeps running and is completely isolated and unaffected by it. The reconfigurable logic, on the other hand, is replaced by the contents of the partial BIT file.

2.1.4.1 Advantages

There are many reasons that make Partial Reconfiguration advantageous\[44][45]:

- The size of the FPGA device required to implement a given function is reduced, with consequent reductions in cost and power consumption;
- An application can use different algorithms or protocols, providing more flexibility;
• It allows to share the hardware among multiple applications, because multiple applications can run on the same FPGA;

• It reduces the reconfiguration time. The length of the reconfiguration time is directly proportional to the size of the bitstream, and since PR allows to reconfigure just a portion of the FPGA instead of the entire device, the reconfiguration time gets shorter;

• It can improve the FPGA fault tolerance;

• It accelerates configurable computing, because it is possible to reconfigure just a portion of the device while the rest is still active, with no down-times.

Furthermore, as mentioned in[46], there are great advantages related to its use in professional electronics:

• **Task Speed**: It allows to overcome the limitation due to the FPGA size, by multiplexing the hardware. This means that a larger number of functions can be implemented in hardware, with relative speed-up with respect to software applications;

• **Power Reduction**: Since it is possible to use just a portion of the FPGA, there is a reduction in power consumption, both in terms of static and operating power consumption[47]: the reduction in static power consumption is due to the fact that the designer can use a smaller device, because not every part of the design is needed 100% of the time and she can load in the device just the reconfigurable modules she needs; on the other hand, it is possible to reduce the operating power consumption using PR to swap out a high-performance design with a low power version of the same design, instead of designing exclusively for maximum performance (which may be required a small percentage of the time);

• **Survivability**: Even if part of the device is damaged, the parts in different Reconfigurable Areas can still run. This is very helpful if the applications is running in a harsh environment (or, for instance, in the space);
- **Mission Change**: If the application is configured for a long period, it is possible to reconfigure the device safely and without interrupting services. In this way, it is not necessary to provide all the functionalities in one design: it is sufficient to switch, at runtime, from a design to another;

- **Environment Change**: The application can be developed for different environment condition, and the user can change the modality at real time;

- **Adaptive Algorithm Change**: An algorithm can be adapted dynamically depending on the external conditions, at lower level of granularity respect to environment change;

- **Online System Test**: It is possible to check online if the circuit is correct how it is supposed to;

- **Hardware Virtualization**: The available hardware is more than the physical FPGA’s area. This makes possible to manage the set of partial hardware design as a component library.

### 2.1.4.2 Limitations

Using Partial Reconfiguration brings some limitations in the design\[^{[45]}\]:

- Clocks and Clock Modifying Logic must reside in the static region;

- I/O and I/O related components, serial transceivers (MGTs) and related components, individual architecture feature components (such as BSCAN, STARTUP, etc.) must remain in the static region of the design;

- Reconfigurable Modules must be locally reset to ensure a predictable starting condition after reconfiguration. In addition, clock and other inputs should be decoupled to prevent spurious writes to memories during reconfiguration;

- No bidirectional interfaces are permitted between static and reconfigurable regions, except in the case where there is a dedicated route;
Moreover, Reconfigurable Modules inside the same Reconfigurable Area must have the same number and types of inputs/outputs.

Additionally to this limitations, PR is not yet diffused also because of the few and not always fully satisfying tools supporting it. Creating a design presenting reconfigurable regions is more complex than creating a standard design. Moreover, it is hard to validate and simulate the design, given the absence of tools.

To prevent a corrupted design to start running, full BIT files contain the informations to verify if they are corrupt and in this case, when loaded into the device, the \texttt{DONE} signal is never asserted and the FPGA does not enter in user mode. Partial BIT files, on the hand, do not contain this type of informations.

When the Partial BIT file is loaded, the FPGA is already in user mode, and no \texttt{DONE} signal is raised. This means that the corrupted design could become active, potentially damaging the FPGA.

Since the partial BIT file is essentially made of just addresses and data, there are two types of errors: \textit{data errors} and \textit{address errors}.

- If the error is in the data error, recovering is simple and consists in loading a new partial BIT file (even a “blank” one) to solve the corruption;
- If the error occurs in the address portion, recovery is more invasive. The corruption could have modified the static part of the design too, and to eliminate the corruption one should load a new full BIT file, resetting the entire FPGA.

The designer should implement Cyclic Redundancy Checking (CRC) to avoid this potentially dangerous situations: a possible solution is to implement a circuit in the static part of the FPGA which analyzes the partial BIT file, and stops the partial reconfiguration process is an error is detected.
2.2 Maxeler Architecture

Maxeler produces super computing systems based on the interaction between a CPU and a DFE, connected to each other via a high speed PCI Express cable.
Maxeler produces both the hardware platforms and software environment which allows to make use of it. Thus, in order to understand how the system works, it is necessary to present it both from an hardware, physical, point of view, and from a software point of view.

### 2.2.1 Hardware point of view

As Figure 10 shows, the two main component of the architecture are the CPU and the DFE, connected via PCI Express cable with a $2GB/s$ capacity. Moreover, both the CPU and the DFE are connected independently to RAM.

The DFE is a board, produced by Maxeler, made of one (or more) Xilinx FPGAs. The DFE is where the actual hardware acceleration is exploited: the code written for it is implemented in hardware, granting huge speed-up improvements with respect to a software implementation. There can be more than one DFEs, connected together via high-bandwidth *MaxRing* intercon-
nect, which allows applications to scale linearly with multiple DFEs in the system, supporting at the same time full overlap of communication and computation.

This hardware separation brings to a difference from a software point of view as well. Programming an application for a Maxeler platform requires to write two different types of code:

- The **host code**, which runs on the CPU;
- The **dataflow code**, which runs on the DFE.

In this way, the user can choose what to run on the CPU, and what to accelerate in the DFE, and they can easily interact by using the PCI Express cable.

The host code can be written in either C/C++ or FORTRAN, while the dataflow code must be written in Java.

Table II summarizes the programming languages that can be used with MaxCompiler, and how they map in the DFE and CPU:

<table>
<thead>
<tr>
<th>Language</th>
<th>Mapping</th>
<th>MaxCompiler Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernels</td>
<td>Java</td>
<td>DFE</td>
</tr>
<tr>
<td>Manager</td>
<td>Java</td>
<td>DFE</td>
</tr>
<tr>
<td>CPU Application</td>
<td>C/C++, FORTRAN</td>
<td>CPU</td>
</tr>
</tbody>
</table>

Table II outlines one of the greatest advantages of using Maxeler platforms: one of the reasons why hardware acceleration on FPGA is not as popular as hardware acceleration based on GPGPU, is because programming an FPGA is usually a complex task, due to the difficulty intrinsic to writing code in some Hardware Description Languages. The usage of a well known
high level programming language, such as Java, makes the task a lot easier for many programmers.

2.2.2 MaxWorkstation

Maxeler provides a large variety of super-computing systems. This thesis targets the MaxWorkstation\textsuperscript{[49]}.

MaxWorkstation is basically a desktop computer running the Linux distribution Centos 5 and it is composed of two main part:

- An Intel i7 processor (with relative main memory and hard disk), responsible for running the host code part of the application;
- A MAX3 board, the so called ”Data Flow Engine” (DFE), where all the parts of the application which must be run in hardware are implemented.

The i7 processor has four cores with a single frequency of 2.8 GHz, and three level of cache memory. It is connected to a Synchronous Dynamic Random-Access Memory (SDRAM) with 16 GB size and a frequency of 1333 MHZ and to 500 GB hard disk. The system composed of the i7 processor, the SDRAM and the hard disk will be referred as the host. The host and the MAX3 board are connected by a PCI express link. The maximum speed of this connection is 2GB/s in each direction.

2.2.2.1 MAX3 Board

Maxeler MAX3 board is composed of two FPGAs, both of them members of the Xilinx Virtex-6 family:

- A Virtex-6 XC6VLX75T, called Interface FPGA (IFPGA);
- A Virtex-6 XC6VSX475T, called Compute FPGA (CFPGA).
Moreover, the MAX3 board has a 24 GB DDR3 SDRAM memory mounted on it: the SDRAM memory is connected via six Small Outline Dual In-line Memory Module (SO-DIMM) sockets to the FPGA, and the maximum supported memory size is 48GB. Maxeler provides a high performance memory controller that accesses all six 64-bit memory modules as one 384-bit wide channel. At the maximum DRAM frequency of 400MHz at double data rate, it gives the theoretical maximum read/write bandwidth of \(384/8 \times 2 \times 400\text{MHz} = 38.4\text{GB/s}\). Table III\[35\] presents a comparison between the capacity of the IFPGA and the CFPGA: the CFPGA results about four times bigger than the IFPGA.

<table>
<thead>
<tr>
<th>Device</th>
<th>Logic Cells</th>
<th>Slices</th>
<th>Max Distributed RAM (Kb)</th>
<th>DSP48E1 Slices</th>
<th>18 Kb RAM blocks</th>
<th>36 Kb RAM Blocks</th>
<th>Max (Kb)</th>
<th>MMCMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC6VLX130</td>
<td>128,000</td>
<td>20,000</td>
<td>1,740</td>
<td>480</td>
<td>528</td>
<td>264</td>
<td>9,504</td>
<td>10</td>
</tr>
<tr>
<td>XC6VSX475</td>
<td>476,160</td>
<td>74,400</td>
<td>7,640</td>
<td>2,016</td>
<td>2,218</td>
<td>1,064</td>
<td>38,304</td>
<td>18</td>
</tr>
</tbody>
</table>

The two FPGAs are connected by means of an Inter-FPGA link (simply a serial link). The CFPGA presents a much bigger area than the IFPGA: it is one of the largest devices available today, and it has 476,160 logic cells and 38,304 kb of Block RAM (BRAM). In fact, the CFPGA is the board where the actual hardware part of the application is implemented and computed. For this reason, it can be seen as a big reconfigurable region.

There is a very simple reason that explains why the IFPGA is needed:

When using the MaxWorkstation, the CFPGA can be configured with full bitstreams more than one time. This can basically happen in two similar fashions:
Some applications may need to load different bitstreams at different times in the same run;

When testing an application the CFPGA is configured many times until the application works as expected. The user loads different full bitstreams to see what the results are.

Without the IFPGA, every time we reconfigure the CFPGA we would lose the Peripheral Component Interconnect (PCI) Express connection, in the sense that the entire board would not be visible by the host. This means that every time the user loads a different bitstream, she should reboot the entire device in order to make the CFPGA visible to the host again, an annoying process which could bring to large time overheads. Therefore, the role of the IFPGA is to ”keep alive” the PCI Express connection, even when the CFPGA is configured multiple times.

When the MaxWorkstation is switched on, the IFPGA is configured with the bitstream contained in a flash memory inside the MAX3 board, while the CFPGA is initially empty. The bitstream used for configuring the CFPGA is not stored in the same flash memory where the IFPGA’s bitstream is stored, but it is stored externally to the MAX3 board, in the host hard disk.

In order to understand how the CFPGA configuration is managed, it is necessary to introduce the different types of streams that are part of the Maxeler architecture.

In the MAX3 board there are three different types of streams:

- Input streams, no more than eight;
- Output streams, no more than eight;
- Control streams, just one.

The input streams go from the host to the CFPGA. They pass initially through the PCI Express, reaching the IFPGA, and they then arrive to the CFPGA by means of the Inter-FPGA link.
Output streams follow the same path of the input streams but on the opposite direction.

While input streams represent the input of the hardware part of the application, output streams represent its output. Therefore, they allow the communication between host code and hardware code. Their number is set to maximum eight just to balance the need for many different inputs and outputs with the need for high bandwidth streams.

Control stream, on the other hand, has a totally different roles, and, among the others, the one of allowing the configuration of the CFPGA.
The bitstream for configuring the CFPGA is part of the so called \textit{MaxFile}. When the application is compiled, the resultant binary contains a buffer (similar to a C array) which stores the bitstream.

The call of the SLiC function \textit{max\_load(MaxFile,device\_name)} in the host code starts the configuration process. By exploiting MaxelerOS routines, the daemon which runs on the Max-Workstation takes sequentially with direct memory access 64 bits chunks of the bitstream, and they arrive through the PCI Express link to the IFPGA.

In the IFPGA is implemented a state machine, written in VHDL, which manages the configuration process. When it receives the command to configure the CFPGA (this happens with the call of \textit{max\_open\_device()}), the bitstream flows from the PCI express port to the state machine through the control stream, and finally to the SelectMAP on the CFPGA, configuring it. Thus, the only role of the VHDL state machine is that of managing the configuration, allowing the passage of the bitstream and signaling possible errors in the process. When the configuration has finished (the \texttt{DONE} signal in the CFPGA is set to high) the state machine asserts the CFPGA reset for a few cycles and then signals to the daemon that the configurations has successfully
ended. At that point, `max_open_device()` returns and the application continues with the rest of the code.

### 2.2.3 Potential advantages of the Partial Reconfiguration support

MaxCompiler generates a monolithic, flat design. There is no use of modularity, and the result of the flow is a single full bitstream. There are three major advantages when using a Maxeler platform:

- MaxCompiler itself, which allows to write in a high level language such as Java the code that will run in hardware. User does not need to know VHDL, neither she must have a deep knowledge of FPGAs;
- The use of the dataflow programming paradigm, and the increase in performances it brings due to features such as the high level of parallelization;
- The automatic interaction between a CPU and a DFE, thanks to SLiC. The user can decide what must run in hardware and what must run in software, passing data from the CPU to the DFE and vice versa just using SLiC and the high data rate PCIe connection.

Of course, since MaxCompiler automatically generates the VHDL code, the user can not directly control how the design is mapped on the FPGA, losing some degrees of flexibility, but it seems like a reasonable drawback given the how easy and fast is to create an application, and the speed-up obtained.

Adding the PR to MaxCompiler can potentially bring to very exciting implications, both in the academic and in the commercial world:

- The already very high performances of Maxeler applications can be enhanced by the use of PR. For instance, an application that needs to repeatedly switch between two or more different modalities could benefit from PR, partially reconfiguring the DFE every time
without needing to stop the application, loading the entire full bitstream, and starting again;

- Researcher can exploit PR in an easier way, making use of the well tested and reliable MaxCompiler;
- Partial Reconfiguration, despite its large spread in the academic field, has not yet gained a significant presence in commercial applications. Its usage in Maxeler platforms could help to enlarge its diffusion in the commercial world and to make it known to a larger number of people, with obvious advantages for the research.

2.3 Dataflow Programming

In a "standard" software application for a CPU, the program is typically written in a high-level programming language (like C, or Python) that is then compiled into a set of instructions. The CPU is an architecture which does not change with time, with a number of functional units that are reused over time, depending on the types of instruction that need to be executed, and with the data being read from and written to memory.

The instructions impose to the functional units what to do with the input data, and when to do it: the program thus consists in a series of statements saying "do this in this moment, do that in this other moment".

The Maxeler paradigm, for the code which runs in the DFE, is different, and is called **dataflow model** of computation: instead of specifying a sequence of processor instructions (computing *in time*), computations are described structurally (computing *in space*): every dataflow core computes simultaneously on neighboring data items in a stream, instead of processing different operations at different times in the same functional units. In other words, the processor is not told when to do something and in which order, but just what to do.

As Section 4.1.2 will show, a high-level language, Java, is used to generate a *graph* of operations, where each node executes a specific function on incoming data and outputs the
result, which becomes the input of another node in the graph. Following this graph, the data flows from a node to the next, without requiring to write back to memory until the chain of processing is complete.

Figure 13: Comparison between (a) traditional control-flow and a dataflow engine (b). The code defining the dataflow engine is translated in a graph of nodes representing very simple operations (dataflow cores). Data flow through the dataflow cores, which perform the operation they represent as soon as their inputs are available. Data are written back to memory only when the entire DFE has been crossed.

Figure 13 describes the comparison between the flow in traditional programming and the one in dataflow cores: the heart of a control-flow processor is a latency-critical loop:

1. Data is read from memory into the processor core;
2. In the processor, operations are executed;
3. The results of the operation are written in memory.
In the dataflow programming paradigm, on the other hand:

1. Data is streamed from memory onto the chip (where all the operations are performed);
2. Data is forwarded directly from one dataflow core (responsible for executing operations) to another, as soon as the operations are performed and the results available;
3. When the chain of processing is complete, data are written to the off-chip memory.

Each dataflow core implements a very simple operation, like for instance an elementary arithmetic operation.

The key structure is therefore a directed graph of operations, where each operation is seen as a black box with well defined inputs and outputs. They run as soon as all of their inputs become valid, as opposed to when the program encounters them. Figure 14 presents such a type of graph, and it is automatically generated by the Maxeler tool MaxRenderGraph.

In a dataflow language, a stream of data is passed from instruction to instruction to be processed. Conditional execution, jumps and procedure calls route the data to different instructions. This could be seen as data flowing through otherwise static instructions like how electrical signals flow through circuits, so the name dataflow.

On the other hand, in a traditional program there is a stream of instructions that operates on data, that are static unless the instruction stream does move them. Parallelism is thus inherent to dataflow programming: operations have not some hidden states to check to see if they can perform some actions. As soon as their inputs are ”ready” the operation can execute, and if there are more than one operations that are ready for the execution, they all can be executed at the same time.

While in a traditional programming paradigm (e.g. control flow programming) the representation inside the computer is a series of instructions running one after the other, a dataflow program can be represented as a hash table where the uniquely identified inputs are the keys, and they are used to look up pointers to instructions.
When any operation completes, the program looks at the list of operations to find the first operation where all of the inputs are currently valid, and runs it. When that operation finishes it will typically put data into one or more outputs, thereby making some other operation become valid.

In this way for parallel operation only the list needs to be shared, and the list itself is the state of the entire program. Instead of being the programmer the one who is in charge of keeping track of the program’s state, it is the program runtime itself that does so.

Therefore, it is possible to divide the application code in two main parts:

Data flow trough streams from the host to the dataflow engine and vice versa. In order to create efficient dataflow application, one must use very regular streams, in order to create a deeply pipelined implementation, which exploits the intrinsic parallelism in the DFE.

Figure 14: Graph representing a Kernel which outputs 255 if the input stream value is above 10, 0 in the other case.
A dataflow engine, in a Maxeler system, is composed of two main elements: Kernels and a Manager.

- Kernels are hardware data-paths implementing the arithmetic and logic computation needed within the algorithm. They perform all the actual computation inside the DFE;
- A Manager comprises all the logic that manages the data flow between Kernels and off-chip I/O in the form of streams. Its goal is to manage the connections among the Kernels, and among the Kernels and the host application.

### 2.4 Summary

This Chapter provided the description of the main concepts relative to the presented work. The first section of the Chapter covered FPGAs, spacing from a general description of their physical structure to the different kinds of FPGAs configuration, especially focusing on the concept of Partial Reconfiguration. Showing how FPGAs are structured and the different types of reconfiguration is necessary to introduce the Maxeler architecture, based on the interaction of a CPU and an FPGA, while Partial Reconfiguration is one of the very main concepts of this work, which aims to design a project flow based on the Maxeler architectures able to support PR.

In Section 2.2, a description of the Maxeler architecture has been provided, with a particular emphasis on the Maxeler system targeted by this work, the MaxWorkstation. MaxWorkstation is the entry-level solution to Maxeler system, but conceptually it does not present any difference from more performing Maxeler systems. Moreover, in Section 2.2.2.1 the DFE of the MaxWorkstation, the MAX3 board, has been presented. MAX3 is made of two FPGAs of the Xilinx Virtex-6 family, the IFPGA, with the role of managing the PCI Express connection and reconfiguring the other Xilinx device, and the CFPGA, where all the hardware section of the
application is implemented. Anyway, MaxWorkstation is not the only Maxeler platform based on the MAX3 board, since it is employed as DFE in other Maxeler products.

Finally, Section 2.3 presents an overview of the **Dataflow Programming** paradigm. This programming paradigm is largely different from the “traditional” Control-flow programming paradigm: as the name suggests, while the former focus on the *data*, which are streamed through the nodes (*i.e.* the atomic operations to be performed) of a graph (*i.e.* the entire application) as soon as they are available, the latter focus on explicitly defining the sequence of all the various steps that must be performed in order to carry out successfully the application.

Despite the concepts in this Chapter have been presented separately, they are all strictly related when talking about Maxeler systems: Dataflow Programming is the paradigm which allows to write code that has to run in Maxeler’s DFEs, and Maxeler’s DFEs are made of FPGAs. The only missing piece to this mosaic is the Partial Reconfiguration, whose integration in the system is the main goal of this dissertation. Maxeler platforms are designed for HPC. Adding to this kind of platforms the flexibility offered by PR would enhance consistently their potentialities, creating a *high performance platform* able to *adapt* itself to the environment and to the user needs in real time.
CHAPTER 3

RELATED WORKS

Configurable devices such as FPGA, provided their extreme flexibility, can be used in very different ways, from using them to create a complete System on Chip to their usage as co-processors to exploit the increase in performance due to hardware acceleration. In this case, FPGAs are often associated to a general-purpose microprocessor: the operations that cannot be executed efficiently in hardware (such as data-dependent control) run in the microprocessor, while the computational cores are mapped into the FPGA.

This Chapter is structured as follows: the first section will present works related generically to FPGAs, focusing in the fields of PR and of FPGAs used as co-processor in video analysis applications, while the second section will present works in which Maxeler platforms are employed to accelerate various kind of applications.

3.1 FPGA related works

The need for FPGAs implementing some applications was born with the necessity to make hardware applications more flexible, changing their functionality after they are manufactured. Standard ASICs do not allow it.

Usually, the applications running on system based on FPGAs are implemented using a single configuration, meaning that the application cannot change the behavior of its circuit while it is running. This kind of applications can be referred as being Compile Time Reconfigurable (CTR), because the entire device’s configuration is determined at the compile-time and does not change throughout system operation. Another, newer, implementation strategy is to implement an application with multiple configurations per single FPGA.
As shown in Section 1.2, in this case the application is divided into time-exclusive operations that do not need (or cannot) operate at the same time, managing to *multiplexing* the FPGA space: each operation corresponds to a distinct configuration that can be loaded into the FPGA when necessary, at run-time and while other operations are possibly running. This approach can be referred to as *Run-Time Reconfiguration (RTR)*, or *Dynamic Reconfiguration*.

There are two ways to implement Dynamic Reconfiguration:

- Dynamic External Reconfiguration;
- Embedded Reconfiguration.

The first one implies that an active array (a column of FPGAs components) may be partially reconfigured by an external device such as a Personal Computer, while the activity of the rest of the circuit remains unchanged. The latter enhances the concept of Dynamic Reconfigurability, assuming that the partial reconfiguration process is led by a part of the FPGA itself.

In[50] is presented a dynamic architecture that, thanks to the processor embedded into the FPGA, is able to dynamically change the design implementation to meet and satisfy all the requirements of the system implementation. The proposed methodology provides a solution for the partial dynamic reconfiguration of an embedded system (Embedded Reconfiguration), using only a common FPGA and development board, without any specific or dedicated device. The Caronte approach creates a new design flow using Xilinx development tools, combining different design flows into a new methodology.

There are plenty of examples in literature of FPGA systems used as co-processors, and among the most successful ones there are those in which the FPGA is used as co-processor for implementing video analysis.

The traditional hardware implementation of image processing uses Digital Signal Processors (DSPs) or Application Specific Integrated Circuits (ASICs). The growing need for faster, and cost-effective systems triggered a shift to Field Programmable Gate Arrays (FPGAs), where the inherent parallelism results in better performance.
As stated in [51], there are five main reasons why implementing image processing in a FPGA is more efficient than implementing it in a standard processor:

- Many image-processing applications are inherently parallel, and the amount of exploitable parallelism is limited only by the size of the image and by the available hardware. In fact, most of the image operations can be performed in parallel, and each output pixel is only dependent upon a small number of input pixels and, in most cases, all output pixels can be computed simultaneously;

- Many image-processing applications consist of very simple operations that are repeatedly applied in a fixed sequence to a set of input images. This sequence of operations can be implemented as a deep pipeline where each stage of the pipeline implements some image-processing operation and all stages operate concurrently;

- Usually image data are low resolution, because current sensor technology limits the width of each pixel to 8-12 bits, where 8 bits tends to be typical;

- Data sets are quite large. A single image can contain several megabytes of information, consisting of a large matrix of pixels. Often, the data is streaming in real-time from a sensor and so represents an infinitely long sequence;

- Finally, many image-processing applications can be represented as simple dataflow graphs with simple control and no loop-carried dependencies.

All this five points make clear that image-processing applications fit naturally into a DFE platform as the Maxeler one. When an application requires real-time processing, like video or television signal processing or real-time track computation of a robotic manipulator, then specifications are very strict and are better met when implemented in hardware. Computationally demanding functions like convolution filters, motion estimators, two dimensional Discrete Cosine Transforms (2D DCTs) and Fast Fourier Transforms (FFTs) are better optimized when targeted on FPGAs. Features like embedded hardware multipliers, increased number of memory
blocks and system-on-a-chip integration enable video applications in FPGAs that outperform conventional DSPs. On the other hand, solutions to a number of imaging problems are more flexible when implemented in software rather than in hardware, especially when they are not computationally demanding or when they need to be executed sporadically in the overall process. Moreover, some hardware components are hard to be redesigned and transferred on a FPGA board from scratch, when they are already a functional part of a computer-based system. Such components are frame grabbers and multiple-camera systems already installed as part of an imaging application.

The work presented in concluded that for some automation systems, given an already installed computer-based imaging application, it is often needed to integrate its components with an FPGA-based accelerators that can exploit the low-level parallelism inherent in hardware structures. For this reason, a critical need arises for an embedded hardware/software interface that can allow for high-bandwidth communication between hardware accelerators and a host application. In fact, in order to obtain high performance either a FPGA system or a GPCPU alone are not enough, and they need to be integrated with a standard software application, giving birth to a heterogeneous system.

In is presented a video filtering processor implemented in a FPGA, where the image RAW data are streamed to the FPGA, where the filters are applied, and the result (a pixels stream) is streamed back to a Video Graphics Array generator that displays it on a monitor. In fact, computationally demanding functions like filters to control brightness, contrast, edge detection, smoothening, gray scale, and scaling are better optimized when targeted on FPGAs.

A similar work, consisting in applying a Sobel edge detector to a video, reducing the memory bandwidth of 75%, is presented in.

Partial Reconfigurability can be used not only to improve systems’ performance, but also to improve fault detection and fault recovery.
In fact, one of the biggest drawbacks to Field Programmable Gate Arrays very high flexibility is that both the SDRAM memory elements storing the configuration bitstreams and the other memory element of the device, are susceptible to radiation-induced temporary faults, called soft errors. This requires the use of some techniques able to cope with fault detection and tolerance.

In [58] partial dynamic reconfiguration is employed to mitigate soft errors. Soft errors are transient fault causing a bit-flip of a value stored in a memory cell, error that can be recovered by re-writing the correct value. In the FPGAs case soft errors may not only corrupt the content of an application register (which brings to an error in the data or in the control) but a configuration register as well, modifying the functionality of a LUT or a routing cell between CLB. The second situation cannot be recovered by a re-computation or a simple application reset, but requires a reconfiguration of the FPGA. The proposed solution uses Triple Modular Redundancy (TMR) to identify and mitigate the occurrence of a bit-flip. In the general case, TMR consists in using three different replicas of the whole system and adding a voter that identifies the correct result among them on the basis of a majority vote. Since the voter itself can be affected by a fault, it is Totally Self-Checking, i.e. it is able to detect its own faults. Moreover the voter signals not only which is the correct result among the three replicas, but also which one of the replicas (or the voter itself) computes the erroneous value. Additionally, it determines whether the fault occurred in a temporary register or in the configuration memory, triggering the reconfiguration of the FPGA’s area affected by the fault.

Partial Reconfiguration has also been used for testing faults in the design, using the fault injection technique. In [59] partial dynamic reconfiguration is used to inject both permanent and transient stuck-at faults in the combinatorial part of the circuit, and asynchronous transient faults in both the combinatorial part and the flip-flops. In this approach, FPGAs are employed for prototyping the circuit, since they allow to perform ”in-system” emulation before any manufacturing, with great economical advantages. The usage of PR led to noticeable time-savings.
compared to other fault injection approaches, where the speed of the reconfiguration is one of the main factors on which the performance of such technique depends.

### 3.2 Maxeler related works

The works previously presented run on "standard" FPGAs systems, like the Xilinx or Altera ones. Maxeler platforms are based on the interaction between one (or more) FPGAs and a standard software application, which runs externally on a CPU. The FPGAs are thus used to exploit hardware acceleration: software programs, which runs in a CPU are often outperformed by hardware programs which runs in a FPGA. Despite the clock frequency in a high-end CPU is often much higher than the clock frequency of an FPGA, tailoring the FPGA configuration on the particular application, together with the usage of deeply pipelined solutions, brings to applications that outperforms their software counterparts both in terms of raw speed and of energy consumption. Maxeler platforms are relatively recent products, but there are some publications which shows their potential in various fields, even if always related to hardware acceleration and dataflow computing.

In [60] is presented a comparison among modern commercial heterogeneous architectures dedicated to hardware acceleration, and the following tables present a comparison of their performance.

The Maxeler board taken into consideration is the MAX2 board, different from the MAX3 board object of this thesis work, and with lower performance.

<table>
<thead>
<tr>
<th></th>
<th>Clock Frequency</th>
<th>Peak Performance</th>
<th>RAM size</th>
<th>Bandwidth to Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max2 Board</td>
<td>150 MHz</td>
<td>116 GFLOPs</td>
<td>1814,4 MB</td>
<td>28 GB/s</td>
</tr>
<tr>
<td>Tesla CI060</td>
<td>1.3 GHz</td>
<td>933 GFLOPs</td>
<td>480 KB</td>
<td>102 GB/s</td>
</tr>
</tbody>
</table>
Table IV presents a comparison between the MAX2 board and the Tesla C1060 GPU, while the Table V shows the Intel Nehalem E5520 performance. Despite to the lower peak performance of the MAX2 board, with respect to Tesla GPU, the authors notes that this parameter is not a good guide of the overall achieved performance, because the high flexibility of the system allow to reach very good solution, superior to alternative more rigid solutions which make use of higher peak performance.

Moreover, authors state the Maxeler platform is able to deal with two of the main problems of the FPGA-based architecture:

- **Ease of Use**, intended as the employed methodology for generating the bitstream for the FPGA, the presence of tools well suited for debugging an application which runs on both the CPU and the FPGA, and the interface between the application and the hardware system (*Application Programming Interface (API)*). Maxeler platforms, thanks to MaxCompiler, the debugging tool *MaxDebug* and SLiC, allow to overcome all this possible issues;

- **Performance**, intended as the optimization of data streaming between CPU and FPGAs, and the scalability of the system. Maxeler platforms, thanks to the fast PCI express link between the host and the DFE and the possibility to connect multiple DFEs with MaxRing interconnect, address both these problems.
Finally, authors agree on the fact that for applications relying on very large datasets and complex numerical computations, the Maxeler dataflow machine offers great performance, superior to other platforms.[60]

In[61] Maxeler platforms are used to compute 3D finite difference modeling. Finite difference is a numerical method applied to different disciplines, such as geosciences, medical imaging and physics simulations. In this publication, finite difference is used to model the wave propagation through earth.

User program is mapped directly into hardware by using a programming environment called MaxGenFD. MaxGenFD is a domain specific compiler for 3D finite difference applications designed to enable geoscientists and programmers to effectively harness the compute power of dataflow hardware without learning circuit design. It is specifically targeted at implementing seismic wave forward modeling, Real Time Migration (RTM) and Full Waveform Inversion (FWI).

MaxGenFD handles the complexities facing any finite difference implementation such as managing very large data sets, boundary conditions and domain decomposition across multiple compute elements with halo exchange. In addition, the compiler removes the need for the programmer to perform hardware-specific optimizations such as customizing data-types and generating optimized convolution stencils. MaxGenFD is implemented as a layer on the top of MaxCompiler. Results showed that performance of a single DFE equates 100-200 CPU cores, with a 30x lower energy consumption.

Besides geosciences, another field where the Maxeler platforms’ potential has been explored is the one of finance, with application focusing on accelerating derivatives’ prices computation.

One of the key enablers of the growth and innovation in the global derivatives markets is the development and intensive use of complex mathematical models. Implementing such mathematical models in the process of valuing and managing the risk of complex portfolios requires thousands of CPU cores for the daily computation of value and risk. The usage of a
such amount of CPU cores requires an incredibly elevate energy consumption, both for powering the CPUs and for the cooling system.

In a Monte Carlo model for simulating interest rates has been implemented using a Maxeler platform, specifically a MaxNode-1821. This platform is composed of eight Intel Xeon cores and two Xilinx Virtex-6 FPGAs connected through MaxRing interconnect, as shown in Figure 15.

In order to find a design that balanced arithmetic optimizations, accuracy, power consumption, and reliability, two design points were selected for implementation:

- A full precision design, for fair value calculations;
• A reduced precision design for scenario analysis.

The full precision has been designed for an average relative accuracy of $10^{-8}$ and the reduced precision for an average relative accuracy of $10^{-4}$. Both designs share identical implementations, with only the compile-time parameters of precision, parallelism, and clock frequency varying.

The design was tested on a fixed population of 29,250 CDO tranches. Table VI reports the speedup achieved by a MaxNode-1821 over an 8 cores (Intel Xeon E5430 2.66 GHz) server, using in both architectures multiple threads to price up to eight tranches in parallel.

<table>
<thead>
<tr>
<th>Precision</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Precision</td>
<td>31x</td>
</tr>
<tr>
<td>Reduced Precision</td>
<td>37x</td>
</tr>
</tbody>
</table>

Table VII shows that the power usage per node, and highlights the fact that it is important to keep the FPGAs busy for as much time as possible.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Idle (W)</th>
<th>Processing (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual Xeon L5430 2.66 GHz Quad Core 48GB DDR DRAM</td>
<td>168</td>
<td>246</td>
</tr>
<tr>
<td>(as above) with MAX2-4412C Dual Xilinx SX240T, 24GB DDR DRAM</td>
<td>191</td>
<td>238</td>
</tr>
</tbody>
</table>
In[63] MaxWorkstation is used in an application related to image analysis: in this paper an algorithm for *Localization Microscopy* is implemented in the MaxWorkstation.

Localization Microscopy improves the resolution of visible light microscopy (namely, of about half the wavelength of light) of approximately an order of magnitude, allowing the non-destructive imaging of living cells with visible light. The algorithm is implemented with two Kernels, where the first one finds the region of interest in the image, and the second applies the actual algorithm only in the regions of interest found in the first Kernel.

The authors state that implementing the algorithm in the MaxWorkstation brought an improvement of the computing time as 225 times the implementation of the same algorithm in Matlab 7.10.0 running in a Intel i5 450.

The first work which tried, in some way, to combine PR with Maxeler platforms is the one presented in[64].

The aim of the work is to exploit partial dynamic reconfiguration in a system based on the interaction between a FPGA and a CPU to accelerate databases management system, with the work of an FPGA based query processor with a regular software database.

The system used in the work is a MaxWorkstation, of the same type of the one used in this work. Since Maxeler did not support PR, the author wrote the code of the partial modules in VHDL, and placed it inside the Kernels, creating custom HDL nodes. Unfortunately the author was not able to create the partial bitstreams to reconfigure the FPGA, and the design remained static.

Despite the not completely successful outcome, the work in[64] shows some interest for Maxeler platforms and their possible usage for accelerating data filtering queries; for this type of work the possibility to exploit PR in the MaxCompiler flow would have been greatly beneficial, because it would have allowed the author to easily create the partially reconfigurable modules and to focus just on accelerating the databases management system.
CHAPTER 4

STARTING SOLUTIONS

The present work, consisting in targeting a HPC platform for implementing a design flow able to support dynamic partial reconfiguration, did not start from scratch.

Maxeler Technologies provides a software environment which allows, starting from a Graphical User Interface (GUI) called MaxIDE, to ignite the entire building process, that will finally produce as output a configuration file, with .max termination. Since the DFE in Maxeler platforms is composed of Xilinx FPGAs, there is a step in the Maxeler design flow where all the Xilinx tools are called in order to produce the bitstream. Aim of this Chapter is to provide a description of the Maxeler tool flow and of the Xilinx tool flow, which are the foundations from which the proposed solution has been built.

The first section of this Chapter will present the Maxeler software environment and its relative design flow, while the second one will delve into the Xilinx tool flow. The last section will present a summary of what has been previously described.

4.1 Maxeler Software Environment

In order to create a Maxeler-based application there are four main software components which helps in the design:

- MaxIDE, an Eclipse-based integrated development environment to aid the development process. The entire project can be written with MaxIDE, which helps in the process in the same way Eclipse eases the process of writing a Java project. After the project has been completed, the user can starts the build process (presented in Section 4.1.2) directly from the GUI, and the Makefile to run the application in hardware is automatically generated;
• **MaxCompiler** is the name for the set of software tools which allows, starting from the code describing the DFE and the code designed for the GPCPU, to generate an executable file suited for the Maxeler platforms. It provides the compilers and libraries necessary for the user to create a dataflow design and using it from the CPU. The various steps performed in the compilation process are described in Section 4.1.2;

• **Simple Live CPU (SLiC)** interface is automatically generated by the dataflow program, and it allows to call dataflow engines from CPU attached to them. In other words, SLiC provides the set of functions that the user can call from the host code to interact with the DFE, e.g. to load the bitstream into the DFE and to stream data to and from the DFE and to and from the memory;

• **MaxelerOS** is used internally by MaxCompiler to provide the link between the CPU and dataflow engines of an accelerated application. MaxelerOS runs on every node with a Maxeler dataflow system, providing drivers for the dataflow engines and IP cores facilitating the communications between the CPU and dataflow engines. It comprises the Linux device driver and a daemon process that controls and monitor the DFE, and it hides the memory management from the host application.

SLiC interface can be accessed at three different levels, depending on the application needs:

• The **Basic Static** interface, where a single function call can run a single DFE using only static memory allocation and static actions;

• The **Advanced Static** interface, which allows to make use of multiple DFEs, of setting multiple complex actions and to set the connections between CPU and DFE;

• The **Advanced Dynamic** interface, which allows to exploit a large set of DFE optimization and a fine-grain control of allocation and de-allocation of all the DFE resources.

The name **DFE** comes from *Dataflow Programming*, the programming paradigm used to define the code which runs on hardware, which has been presented in Section 2.3 and which is
4.1.1 Preliminary analysis

The main goal of a Maxeler system is to improve the performance of a "standard" application, both in terms of speed and of energy consumption. To do it, it is necessary to analyze substantially different from the Control flow Programming paradigm.
the application source code in order to decide which part of the code has to run in the dataflow engine and which part can execute on the CPU. Selecting the optimal components can be a not trivial task, and Maxeler in [48] identifies the following key factors:

- To measure how long it takes to run the application on CPUs given a set of representative (large enough) datasets. CPU memory systems do not scale linearly with problem size and dataflow technology is targeting large datasets, therefore focusing in toy examples is just a waste of time;

- To perform a more detailed analysis able to provide the distribution of runtime of various parts of the application including an analysis of time spent in computation and time spent in communication. Most of the analysis can be achieved with time counters and profiling tools like for instance gprof[65], oprofile[66], valgrind[67] etc;

- The previous two steps aims at maximizing run-time coverage: The selected components should represent the majority of the application run-time. Speedup is limited by the portion of the run-time that remains on the CPU[68];

- To maximize regularity of computation: it is better to run repeatedly on the dataflow engine the same operation on many different data items, for instance when computing an inner loop. It is suggested to consider all the possible loop transformations to estimate their relative performances, in order to chose the one that suits better the implementation on the DFE;

- To minimize communication between CPU and dataflow engines: Sending/receiving many data between the CPU and dataflow engine is expensive, since communication is slower than computation. A good solution is to find a way to keep both CPU and DFE busy while data are streamed between them.
Once the components have been identified and both the host code and the dataflow code have been written, it is possible to compile the entire application directly from MaxIDE.

4.1.2 Compiling

The compilation process is made of several stages, and it can be started directly from MaxIDE:

1. **Java Compilation**: In this stage, the MaxCompiler Java compiler is used to compile the user code, with standard Java syntax checking etc. taking place;

2. The following steps of the compilation are executed at **Java run time**, when the compiled Java code is executed. This stage encapsulates the following steps:

   (a) **Graph construction**: In this stage a graph of computation is constructed in memory, based on the user calls to the Kernel Compiler API;

   (b) **Kernel Compiler compilation**: The Kernel compiler takes the generated graph, optimizes it and converts it into either a low-level format suitable for generating a dataflow engine or a simulation model;

   (c) **Back-end compilation**: For a hardware build (therefore, not in case of a simulation) Xilinx tools are called automatically by MaxCompiler to further compile the design into a chip configuration file.

Figure 17 represents the compilation flow.

In the compilation process it is necessary to call the Xilinx tools because the dataflow engine is basically composed of one or more Xilinx FPGAs (depending on the Maxeler board used).

There is, then, a phase of the compilation process where the Kernels and the Manager are "translated" in VHDL. This highlights one of the greatest strengths of the Maxeler architecture: all the component of the application, from the host code to Kernels and Managers, are written
in high level languages, such as C++ and Java. Despite the fact that Kernels and Manager will be implemented in an FPGA, the programmer does not need to write a single line of VHDL code. The Java code which describes Kernels and Managers is "translated" automatically in VHDL by the MaxCompiler, more precisely by the MaxDC tool.

For these reasons, designing an application using Maxeler flow is much easier and faster with respect to the "standard" hardware design flow.

The Xilinx tools employed are the same that are usually employed in a Xilinx hardware design, and they will be shown in Section 4.2.
The result of the Xilinx flow is the BIT file, that will be then encapsulated in the dataflow engine configuration file, with a .max extension. This file contains both data used to configure the dataflow engine and meta-data used by software to communicate with this specific dataflow engine configuration.

To integrate with a CPU application, the designer replaces the accelerated sections of code in the CPU application with API calls to talk to the Kernels and Manager in the dataflow engine. SLiC provides an interface to MaxelerOS for streaming data to and from the Kernels in the dataflow engine, abstracting away the details of the Direct Memory Access (DMA) transfers over the PCI Express bus. Using a standard software compiler such as gcc\textsuperscript{[60]}, SLiC interface and the CPU software application are linked against the .max file generated by MaxCompiler. MaxelerOS is then able to automate the configuration of dataflow engines and provide the communications between the CPU software and the dataflow engine implementation.

4.1.3 Manager Compiler

For very simple design usually it is enough to use a Standard Manager. The Standard Manager can manage a single Kernel, limiting in a strong way its possible usage. For more complex designs, it is recommended to use the Manager Compiler\textsuperscript{[70]}, which allows to create more elaborate designs with, for instance, multiple Kernels, interconnections among multiple boards, multiple memory controller groups and memory command generators, memory control streams generated by Kernels or the CPU and blocks to split and join streams. The Manager Compiler does not consist in a new phase in the design flow, but it is part of the tool which "translates" the Java code into the VHDL code: all the steps following the Java to VHDL translation stay the same. It is essential to show how complex Managers are structured and implemented because the proposed solution is largely based on a new Manager design.

In a Manager Compiler design there are two types of object:
• **Blocks**, which performs some function or connect to some resource and have a number of input streams and a number of output streams. Blocks include Kernels themselves, blocks for splitting and joining streams and implicit blocks for interfaces to resources external to the Dataflow Engine;

• **Streams**, which are intended as unidirectional and asynchronous from each other, and the latter characteristic distinguishes them from the streams inside a Kernel, which are synchronous. The synchronous nature of streams inside a Kernel can make applications where the processing rates of streams are different tricky to implement, requiring complex conditional stream control that renders the Kernel very convoluted. Such processing can be split into two or more Kernels, where the streams within each Kernel are synchronous but the Kernels run asynchronously from each other.

Moreover, streams in the Manager Compiler can only be connected to one input stream and the output of a stream can only be connected to one other stream. Managers created using the Manager Compiler extend the class *Custom Manager*, permitting to tailor the Manager functionalities to the application’s needs. The following code shows an example of Custom Manager.
public class FilterManager extends CustomManager {
    FilterManager(FilterEngineParameters engineParameters) {
        super(engineParameters);
        KernelBlock k1 = addKernel(new GrayScaleKernel(
            makeKernelParameters("GrayScaleKernel")));
        KernelBlock k2 = addKernel(new GaussianBlurKernel(
            makeKernelParameters("GaussianBlurKernel")));
        KernelBlock k3 = addKernel(new EdgeLaplaceKernel(
            makeKernelParameters("EdgeLaplaceKernel")));
        KernelBlock k4 = addKernel(new ThresholdKernel(
            makeKernelParameters("ThresholdKernel")));
        Stream x = addStreamFromHost("input");
        //GrayScale kernel takes the input from host
        k1.getInput("input") == x;
        //GaussianBlur takes as input the GrayScale kernel output
        k2.getInput("input") == k1.getOutput("output");
        //EdgeLaplace kernel takes as input the GaussianBlur kernel output
        k3.getInput("input") == k2.getOutput("output");
        //Threshold kernel takes as input the EdgeLaplace kernel output
        k4.getInput("input") == k3.getOutput("output");
        Stream z = addStreamToHost("output");
        //Finally, Threshold kernel output goes to the host application.
        z == k4.getOutput("output");
    }
}

Listing 2: Custom Manager example
In this example the main components are the **KernelBlocks** and the **Streams**. Inside a KernelBlock it is possible to place one (or more) Kernels. KernelBlocks are connected to each others and to the host application by the Streams. The **StreamFromHost** represents the input of the DFE (coming from the host code), while the **StreamsToHost** represents the output of the DFE (going back to the host code). Figure 19 shows the connections among KernelBlocks and the host code.

![Diagram](image)

**Figure 18:** Application’s Streams. The data flow sequentially through the four Kernels, and finally go back to the host application.

If the user needs to integrate into the Maxeler flow a third party Intellectual Property (IP) block written in either Verilog or VHDL, she can instantiate it directly inside a Custom Manager. The result is a Manager block which can communicate with other blocks and with the
CPU via streams.

4.1.4 Kernel examples

In order to show how a Kernel works, and the resulting graph, an example taken from the Maxeler tutorial\cite{48} can be very useful. The example presents a moving average Kernel: The application computes a 3-point moving average over a sequence of $N$ data values.

The moving average can be expressed as:

$$y_i = \begin{cases} (x_i + x_{i+1})/2 & \text{if } i = 0 \\ (x_{i-1} + x_i)/2 & \text{if } i = N - 1 \\ (x_{i-1} + x_i + x_{i+1})/3 & \text{otherwise} \end{cases}$$

Differently from a software implementation, where an array would be used to hold the data to be then scanned with a loop, in the Kernel is only necessary to manipulate the input stream, exploiting dataflow programming model of computation.

The code in Figure 19 does not take into consideration boundary conditions.

Until now, the only proposed way to allow the communication between Kernels and the Manager have been the streams. But, besides the streams, Kernels and the CPU application can communicate also by using the so called scalar inputs and scalar outputs. As the name suggest, they allow to exchange between Kernels and the host application a single, atomic, scalar value, instead of an entire stream. They (especially the scalar inputs) have been introduced in the Maxeler design to permit some kind of dynamic change in the application: consider, for instance, Listing 3, an example taken from\cite{48}.\[48]
public class MovingAverage extends kernel {
    public MovingAverage(KernelParameters params, int N) {
        super(parameters);

        // input
        HWVar x = io.input("x", hwFloat(8, 24));

        // Data
        HWVar prev = stream.offset(x, -1);
        HWVar prev = stream.offset(x, 1);
        HWVar sum = prev + x + next;
        HWVar result = sum / 3;

        // Output
        io.output("y", result, hwFloat(8, 24));
    }
}

Figure 19: Moving average Kernel code and graph.

class AddScalarKernel extends Kernel {
    AddScalarKernel(KernelParameters parameters) {
        super(parameters);

        HWFloat singleType = hwFloat(8, 24);
        HWVar a = io.input("a", singleType);
        HWVar b = io.scalarInput("b", singleType);
        HWVar result = a + b; // StreamOutput
        io.output("c", result, singleType);
    }
}

Listing 3: Example code of a Kernel which makes use of a scalar input.
In this very simple example it is possible to modify the behavior of the Kernel even after the code has been compiled and the .max file produced: from the host code, by calling the function `AddScalar(size, scalarIn, dataIn, dataOut)`, the value of \( b \) can be changed at run-time. Transferring scalar inputs and outputs back and forth from and to the DFE is not fast, therefore it is recommended to set all the scalar inputs together, so that they will be streamed to the DFE in a single transaction, and then to run the DFE for a long time.

### 4.2 Xilinx Flow

The Xilinx design flow starts with the writing of the code which describes the digital logic of the hardware circuit, using a HDL such as VHDL or Verilog. As seen in Section 4.1.2 MaxCompiler takes care of translating the Java code describing the DFE in VHDL. After this step, the following Xilinx tools are employed:

1. The first step consist in calling **Xilinx Synthesis Technology (XST)**. This tool allows to synthesize the HDL files, obtaining netlist files. Netlists are the files describing the connectivity of an electronic design;

2. In the second step **NGCBuild** is called. This tool allows to compile multiple source netlists into a single NGC file that can be delivered as an atomic entity to the following steps of the flow;

3. **NGDBuild**: NGDBuild reads in a netlist file and creates a Xilinx Native Generic Database (NGD) file that contains a logical description of the design in terms of logic elements, such as AND gates, LUTs, flip-flops and RAMs. In addition to the netlists file, NGDBuild takes in input a User Constraint File (UCF). The UCF file contains timing and layout constraints that affect how the logical design is implemented in the target device;

4. **MAP**: The MAP tool maps a logical design to an FPGA. The input is an NGD file, generated at the previous step. MAP, first, performs a logical Design Rule Check (DRC)
on the design in the NGD file, then maps the design logic to components in the target FPGA. The output is an Native Circuit Description (NCD) file, a physical representation of the design mapped to the components in the targeted Xilinx FPGA;

5. **PAR**: PAR accepts a mapped NCD file as input, place and routes the design, and outputs a routed NCD file. In addition to the mapped NCD file PAR takes as input a Physical Constraints FIle (PCF), that contains constraints based on timing, physical placements and other attributes placed in a UCF file;

6. **BitGen**: BitGen generates a a bitstream for Xilinx devices configuration. BitGen takes a fully routed NCF file as input and produces a configuration bitstream (BIT) as output. A BIT file is a binary file with a .bit extension.

More informations can be found in [33]. Figure 20 will summarize the Xilinx flow.

---

Figure 20: Xilinx flow, from the VHDL or Verilog code to the bitstream.

Thus, by following the Xilinx flow, it is possible to obtain the BIT file starting from the HDL.
4.2.1 Partial Reconfiguration case

In the case the project contains partially reconfigurable areas, the design flow is slightly different.

The project synthesis is done following a bottom-up fashion: each Reconfigurable Module is synthesized independently from the other, while the static modules can be synthesized together, to generate one single netlist, or individually, generating multiple static netlists.

The final result of the steps following the synthesis is a full design containing one Reconfigurable Module for each Reconfigurable Areas and the static logic. To implement all Reconfigurable Modules it is important to choose a subset of all possible Reconfigurable Module combinations and implement them as unique designs. Each unique implementation is called a Configuration.
It is sufficient to implement only the Configurations that contain each module once, since the partial BIT file for a module is independent of the other Reconfigurable Modules. However, after the first configuration has been implemented, it is necessary to import its static part in all the following configurations, otherwise the different Reconfigurable Modules implementations will not be compatible.

The result of the flow is a full BIT file, describing the static part and every Reconfigurable Modules, and one partial BIT file for each Reconfigurable Modules. The entire flow is represented in Figure 22.

Once a partial bitstream is created, it can be loaded in the FPGA device in any combination of full or partial bitstreams created within that PR project.

In addition to the PXML file (used to define partition), in order to make Partial Reconfiguration work properly, the user must specify an Area Group Constraints for each partition in the UCF file.

UCF files are American Standard Code for Information Interchange (ASCII) files specifying constraints on the logical design. In a UCF file an AREA_GROUP constraint is a grouping constraints that associates logical design elements with a particular label or group. AREA_GROUP constraints and Partition definitions are necessary to delineate the static (non-reconfigurable) logic from the reconfigurable logic, preventing logic in the static design from merging with logic in the RMs, and vice versa. The AREA_GROUP constraints must be defined for each Reconfigurable Partition.

The Xilinx tools employed to generate a partially reconfigurable design are the same used in the static case, with the only difference that the different tools are called multiple times. The flow can be seen as divided in two phases:

- In the first phase, XST and NGCBuild are called independently for each partition, both the static and the reconfigurable ones;
In the second phase NGDBuild, MAP, PAR and Bitgen are called as many times as the number of Configurations in the design, in a sequential fashion. It is possible to import...
Figure 23: Bitstreams generation: for each Reconfigurable Block there is a partial bitstream, and a full bitstream describes the static part.

the results of one Configuration to a following one, as specified in the PXML file. Moreover, the user can call the `pr_verify` tool to make sure Configurations results are compatible.

Figure 24 represent the design flow.

A specific tool that is usually employed to create a partially reconfigurable design with a Xilinx FPGA is PlanAhead[^71]. PlanAhead is a tool which can be used to analyze and floorplan Xilinx designs. It can be used for integrating partial reconfiguration in the Xilinx design tools, and to constrain the routing for the interface with the partial region and it makes use of the proxy logic.

Proxy Logic is a fixed, known point which acts as interface between static and reconfigurable logic[^45]. It consists in a LUT automatically inserted for each Partition Pin, where a Partition Pin is the logical and physical connection between static and reconfigurable logic. There is one Partition Pin for each Reconfigurable Area input and output, an this is one of the reason why
Figure 24: In this example there is one static (top-level) partitions and three reconfigurable ones, arranged in three Configurations. In the first phase, they are synthesized by calling XST and NGCBuild. In the second phase NGDBuild, MAP, PAR and Bitgen are called, to produce the full bitstreams and all the partial ones.
the inputs and outputs of Reconfigurable Modules belonging to the same Reconfigurable Area must match. The Proxy Logic is placed in the static part of the design (even if within the Area Group of its Reconfigurable Area), so that it stays the same for each module loaded inside a Reconfigurable Area.

4.3 Summary

In this Chapter it has been described the starting point of the proposed work. Aiming to propose a design flow which supports Partial Reconfiguration in the Maxeler architecture requires a deep analysis of the current tool chain. The current MaxCompiler version is the starting point of the work: without a good knowledge of its structure, reaching this work’s main goal would have not been possible. Similarly, as the DFE components are Xilinx FPGAs, it has been necessary to study a way integrate them as smoothly as possible in the design flow.

Since the entire proposed design flow has been thought to be as automated as possible (i.e., requiring the user to only start the building process, without having to take continuously low-level decisions) all the Xilinx tools have to be called by MaxCompiler itself from the command line\cite{33}: this means that graphical tools such as PlanAhead, which takes care of many details (e.g. the generation of the PXML file) could have not been employed, making the implementation work harder.

As a matter of fact, the approach that will be proposed in the next Chapter aims to substitute entirely the ”standard” Xilinx tools flow, centered on graphical tools such as Integrated Synthesis Environment (ISE)\cite{72} and PlanAhead\cite{71} with the a new, improved, version of MaxCompiler: the user will have to care only about writing the DFE code and the host code. All the following synthesis steps, will be carried out by MaxCompiler.
CHAPTER 5

PROPOSED APPROACH

In Chapter 4 the Maxeler software environment has been presented, showing the starting point of this work. This Chapter will present the analysis that took place to decide a well suited methodology to create a design flow, based on the Maxeler platforms, able to support Partial Reconfiguration, and the considerations that led to the foundations of the solution.

The first section of this Chapter will present the analysis of the Maxeler architecture, that has been carried out to understand if PR could have been in some way compatible with the pre-existing architecture. The second section will show the final wrap up of the proposed design, necessary to summarize those which are the guidelines of the final solution.

5.1 Current system analysis

As shown in Chapter 4, originally Maxeler did support neither Partial Reconfiguration nor a hierarchical design: the entire flow was static and "flat". At the beginning of the work there was not certainty that main goal could have been fully accomplished, and an analysis has been made to understand in which part, and possibly in which way, of the Maxeler flow PR should have been inserted.

The first step to implement partial reconfigurability has been to decide which part of the DFE should have been made partially reconfigurable and which granularity level should have been targeted.

Since all the computations, inside the DFE, is done by the Kernels, it has been decided that the most suitable DFE part for Partial Reconfiguration were the Kernels. Manager has the only role to coordinate the Kernels, and making Manager partially reconfigurable did not seem to be a suitable solution:
Manager are not responsible for the execution of any arithmetic/logic computation. Usually, users are more interested to dynamically modify the behavior of the device;

As the name suggest, the main role of Managers is that of managing the DFE. Partially reconfiguring a Manager would imply the presence of some new upper layer in the hierarchy, responsible for the managing of the Manager reconfiguration. This choice seemed to be redundant and complex to accomplish.

After the decision to target Kernels has been made, the next question was if allowing to partially reconfigure an entire Kernel or just some fraction of the Kernel. Reconfiguring just a part of a Kernel instead of an entire one did not seem to comport any difference: in fact, if someone needs to modify just slightly the behavior of a Kernel (e.g. dividing something by three instead than by five), it is possible to accomplish that by using a scalar input (which have been presented in Section 4.1.4). On the other hand, if the modification is very small, but with differences from a logical point of view (e.g. substituting $a + b$ with $a - b$), one can break-up the design of the Kernel in many smaller Kernels that, put together, perform the same functionality.

Figure 25 shows the difference between a monolithic design, in which a very large Kernel performs many operations, and a modular one, where there are multiple Kernels that perform a limited number of operations.

The decision of targeting Kernels simply refers to the "Maxeler point of view", which is an abstraction over the physical Xilinx FPGA. In a standard Maxeler design, the user is not aware of the place where Kernels are located inside the FPGA, but in the Xilinx flow, when using PR, the user must define the area where the Reconfigurable Partitions must be placed. Therefore, there were two possible choices:

- Hiding the process of Reconfigurable Partitions area definition to the user, automating it inside the MaxCompiler;
• Letting the user decide in which part of the FPGA to place the Reconfigurable Partition.

Automating the area constraints definition would require the presence of some tool, able to optimize the choice of the area on the base of the Kernel characteristics. The implementation of such a tool was not among this Thesis goals, and it has been decided to let the user define the area constraints, leaving the automation of the process to a future work. To keep the area constraints definition simpler, it has been decided to express area constraints in terms of clock regions\[^{73}\]. A clock region is a part of the FPGA where it is assured zero skew clock
distribution. Clock regions in the FPGAs belonging to the Virtex-6 family have 12 global clock domains, and they are 40 CLBs tall spanning half of the die.

Figure 26: Clock Region in a Xilinx XC6VSX475T, the board employed as CFPGA in the MAX3 DFE.

5.1.1 MaxCompiler proposed modification

MaxCompiler consists in a Java library of very large dimensions, which takes care of the entire building process. MaxCompiler is one of the key factors in the success of Maxeler systems, and this implies that the modifications to support PR had to be tailored to its pre-existent structure: it was not possible to re-think entirely the Maxeler design flow to support PR, but all the modifications had to be as limited as possible to what was already present, keeping well in mind how the whole flow is implemented in order to preserve the same rationale.

Once it has been decided to employ single Kernels as minimum (and maximum as well) level of granularity, the next step has been to figure out how the Partial Reconfiguration should
have been accessed by the user: in fact, since this work targets an application conceived for commercial purpose, and not merely academic purposes, it has been essential to focus on the way PR would have been made accessible to the user, especially focusing on an interface as friendly as possible and in a design which allows to the user to exploit all the potentiality offered by PR without caring about the implementation details.

Kernels design itself did not need any modification, because there are not difference in the way a reconfigurable module and a static module are implemented. The only differences between a design without PR and a partially reconfigurable design are:

- The concept itself of *Reconfigurable Partition*. In some part of the MaxCompiler design this different type of "entity" has to be declared and instantiated;
- The design flow, that, as showed in Section 4.2.1, requires a different use of the Xilinx tools.

In the "standard" Maxeler design flow, the entire build process starts with the `build` method in the Manager, which gives birth to the whole compilation flow, from the VHDL translation to the `.max` file generation. Moreover the Manager, as shown in Section 4.1.3, is responsible for the Kernels instantiation and the definition of the streams connecting the Kernels and the host application. This makes clear that the Manager is responsible for the two most important design modification needed by PR, and it has been decided to implement a new type of Manager to support PR at high level (namely, at the level of the DFE Java code).

Drilling down through the levels of abstraction, the final step of the proposed approach has been to establish how to physically partially reconfigure the device. As shown in Section 2.2.2.1, in the IFPGA is present a VHDL state machine which controls the CFPGA configuration, carried out through the SelectMAP: the BIT file, contained inside the `.max` file is loaded from the host, it passes through the IFPGA, and the VHDL state machine is responsible for its loading into the CFPGA. Thus, the possibly simplest solution was to replicate a similar behavior:
• The .max file should contain not only the full BIT file, but all the partial BIT files as well;

• The VHDL state machine should be modified to support a new command to partially reconfigure the CFPGA;

• The PR should be exploited through the SelectMAP.

5.2 Conclusions

The entire proposed solution is thus founded on five main principles:

• Creating a new Manager class to support PR;

• Keeping as much as possible the structure of the "standard" MaxCompiler flow;

• Automating as much as possible to entire process, so that the user can exploit all the benefits of PR without running into troubles due to the higher complexity of the PR flow;

• Preserving all the Xilinx PR concepts, such as Configurations, Reconfigurable Partitions and Reconfigurable Modules;

• Using, at a physical level, similar solutions to the ones used by Maxeler in the case of a full configuration to reconfigure the CFPGA.

In this way, we are now able to classify the PR schema proposed by following the five points presented in Section 2.1.2:

• *who* controls the reconfiguration is the VHDL state machine, located in the IFPGA;

• *where*: The IFPGA is located into the MAX3 board. Although the PR is managed inside the DFE, it is a kind of *external configuration*, because the reconfigured device (the CFPGA) is not the one managing the reconfiguration process;
• *when* the modules are generated: Xilinx flow forces the user to define which modules she
want to synthesize at compile time. So, the modules are generated at compile time, but
the user can place them in the device, by partially reconfiguring it, at run time;

• *which* is the granularity of the reconfiguration: From a Maxeler point of view, the granu-
ularity level are the single Kernels, while from a Xilinx point of view the granularity level
are the Clock Regions;

• *dimension*: It is a bi-dimensional reconfiguration, because it is possible to reconfigure a
portion of the column of reconfigurable elements without stopping the rest of the column.

Modifying the MaxCompiler flow and supporting PR is a task which presents numerous
issues, as MaxCompiler lacks of concept such as Partitioning and Hierarchical Design[43][74].
User must be able to define reconfigurable partitions and reconfigurable modules, and the flow
is no more linear but instead it becomes in some way *iterative*: results from one iteration can
be necessary for some iterations in the future (*e.g.* importing partition), and this requires to
keep track of all the intermediate results and of the place where they are stored. Moreover,
resulting file system structure is much more complex with respect to a static design, with many
more ramifications.
CHAPTER 6

PROPOSED IMPLEMENTATION

In Chapter 5, a possible solution to support PR in the MaxCompiler flow has been presented. This solution has been implemented during the author’s visit to Maxeler Technologies, and some details of the implementation will be shown in this Chapter.

Introducing Partial Reconfiguration in the Maxeler design flow required to modify a relatively large part of the MaxCompiler library and to create new Java classes to support PR. Explaining in detail all the implementation steps is not the goal of this Chapter.

The first section of this Chapter will provide a description of the MaxCompiler user interface, relative to PR: basically, it consists of what the user can see of the implemented work. In the second section, the new design flow will be described with more details, without explicitly specifying the source code.

6.1 User interface

Due to the differentiation between the code which runs in the DFE and the code which runs in the CPU, MaxCompiler modifications required to support Partial Reconfiguration concern both the Java side of the application and the host code, more precisely SLiC interface.

- On the Java (i.e. DFE) side, MaxCompiler libraries need to be modified to allow the creation of Reconfigurable Partitions and the support to different design flow needed to generate a partially reconfigurable design;

- On the host code side, some features must be added to the SLiC interface in order to partially reconfigure the device. In fact, being the concept of PR intrinsically dynamic, the only way for the user to modify the DFE configuration at run-time is to call some SLiC function from the host code to execute a partial reconfiguration of the DFE.
For this reason, Section 6.1.1 will show the user interface regarding the DFE code. The result of this step will be the .max file, containing the DFE configuration data. Section 6.1.2 on the other hand will show how the user can dynamically modify the configuration of the DFE from the host code, making use of the results of the previous phase.

6.1.1 DFE code

As shown in Section 5.2, the linchpin of the modification of the MaxCompiler is the Manager. To make a partially reconfigurable as similar as possible to a "standard" design, it has been implemented a new class called *PRManager*, which extends the Custom Manager class.

When using PR, the user must create a Manager which extends PRManager. Inside the PRManager, in order to define a Reconfigurable Area, the user must create a *Reconfigurable-Block*, which corresponds to a Reconfigurable Area. The Reconfigurable Block constructor takes as inputs the coordinates of the clock regions assigned to that Reconfigurable Area and the Kernels (one or more) that will be part of that Reconfigurable Area. This means the one Reconfigurable Block can occupy one or more clock regions, and not a fraction of a single clock region.

For this reason, the maximum number of Reconfigurable Blocks correspond to the number of clock regions, which corresponds to 18 in the MAX3 CFPGA. In the FPGA there will be exactly one Kernel per Reconfigurable Block at the same time: the Kernels taken as input by the ReconfigurableBlock constructor represent the possible configurations of that Reconfigurable Block. After the Reconfigurable Kernels have been instantiated, the user can define the connections among them and the host code in the same way of a Custom Manager.
1. public class TestFilterManager extends PRManager {
2. TestFilterManager() {
3. super(MAX3BoardModel.MAX3424A, "TestFilter", Target.MAXFILE_FOR_HARDWARE);
4. GrayScaleKernel k1 = new GrayScaleKernel(makeKernelParameters("GrayScaleKernel"));
5. GaussianBlurKernel k2 = new GaussianBlurKernel(makeKernelParameters("GaussianBlurKernel"));
6. EdgeLaplaceKernel k3 = new EdgeLaplaceKernel(makeKernelParameters("EdgeLaplaceKernel"));
7. ThresholdKernel k4 = new ThresholdKernel(makeKernelParameters("ThresholdKernel"));
8. ReconfigurableBlock b1 = addReconfigurableBlock("reconf1", 0, 8, 0, 8, k1, k3);
9. ReconfigurableBlock b2 = addReconfigurableBlock("reconf2", 0, 7, 0, 7, k2, k4);
10. Stream x = addStreamFromHost("input");
11. Stream z = addStreamToHost("output");
12. b1.getInput("input") <== x;
13. b2.getInput("input") <== b1.getOutput("output");
14. z <== b2.getOutput("output");
15. }

Listing 4: Example of a PRManager. There are four Kernels and a Reconfigurable Block. Each Kernel can independently be loaded in the Reconfigurable Block, which is connected to the host application through an input stream and an output stream.

As Listing 4 shows, there is not any difference in the way Kernels are created. As shown in Section 4.1.3 a Custom Manager make use of of KernelBlocks. In a similar way, PRManager can use ReconfigurableBlocks, that are connected with each other and with the CPU in the same fashion of the KernelBlocks. Since PRManager inherits from CustomManager, it is possible to instantiate KernelBlocks also in a PRManager: they are used to place Kernels in the static part of the design.
After the PRManager has been instantiated, the user must define the *Configurations*. A Configuration represents the assignment of exactly one Kernel to each Reconfigurable Block. Specifying the Configurations is very important, because later on the design flow for each pair Kernel - Reconfigurable Block in a Configuration, the corresponding partial bitstream will be generated.

Listing 5 shows the creation of four configurations for the Kernels created in Listing 4. There is just one Reconfigurable Area, with four possible Kernels inside it. Thus, four Configurations are created, which assign one time every Kernel to the Reconfigurable Block, that will correspond to four partial BIT files. It is not mandatory to create a Configuration for every possible combinations of Kernels-Reconfigurable Blocks, it depends on the application needs.
public class FasiFilterBuilder {
    public static void main(String[] args) throws Exception {
        FasiFilterEngineParameters engineParameters =
                new FasiFilterEngineParameters(args);
        FasiFilterManager m = new FasiFilterManager(engineParameters);
        PRConfiguration c1 = new PRConfiguration("conf1");
        PRConfiguration c2 = new PRConfiguration("conf2");
        PRConfiguration c3 = new PRConfiguration("conf3");
        PRConfiguration c4 = new PRConfiguration("conf4");
        c1.setKernel("reconf1", "GrayScaleKernel");
        c2.setKernel("reconf1", "GaussianBlurKernel");
        c3.setKernel("reconf1", "EdgeLaplaceKernel");
        c4.setKernel("reconf1", "ThresholdKernel");
        m.build(c3, c2, c4, c1);
    }
}

Listing 5: Example of a Builder. There are four Configuration, one for each Kernel that can be placed in the Reconfigurable Block reconf1, meaning that four partial BIT files will be produced. The building process will start with the call of the build method.

Finally, user must call the PRManager build(Configuration ... conf) method, which starts the building process.

Figure 27 represents the PR flow in the Dataflow Engine from the user perspective.
Figure 27: DFE flow from the user perspective.

6.1.2 Host code

In order to exploit Partial Reconfiguration from the host code, it has been added to SLiC interface the function `max_reconfig_partial_bitstream(device, Kernel_name)`. By calling this function the user can reconfigure the Reconfigurable Area that the selected Kernel belongs to, loading that Kernel partial bitstream. Kernels names are univocal, and the same Kernel can not be placed in two different Reconfigurable Blocks, avoiding in this way possible name clashes. `max_reconfig_partial_bitstream` allows not only to load into the design Reconfigurable Modules created with MaxCompiler and which are encapsulated into the `.max` file, but also to load partial bitstream created externally with the standard Xilinx flow. In this case, instead of specifying the Kernel name, user must specify the path to the partial bitstream.

Loading partial bitstreams into the FPGA in this way can be dangerous, because the user could load partial bitstreams that are not compatible with the design and that could potentially damage the device. For this reason, the user should make use of this feature only if she is totally sure the partial bitstream she is loading can not harm the system.

From the user point of view, then, Partial Reconfiguration does not bring any big change to the standard Maxeler design: implementing a partially reconfigurable design is very similar to implementing a "static" design, and she does not notice any of the differences due to the
MaxCompiler modification.

6.2 Design flow implementation

MaxCompiler flow, as shown in Section 4.1.1 is a linear flow: starting from the Java code passing through the VHDL translation and finally to the use of all the Xilinx tools the design follows a continuous transformation where the different DFE’s components (Kernels and Manager) are processed all together.

Partial Reconfiguration requires a modular design and the use of partitions: it is not possible to process all the components together, because there are different flow stages and different synthesis levels.

There are two main differences are one of vertical type and the other of horizontal type:

- The first distinction is the one between the static part and the dynamic part. It is important to know that in this case static means ”everything that is not reconfigurable”, so both the Manager and the Kernels placed inside a KernelBlock. The static part is one hierarchical level above the dynamic part, therefore bringing to a vertical distinction in the design;

- The second distinction is due to the fact that in a PR design the tools NGDBuild, Map, PAR and Bitgen are not used just one time: they are called as many as the number of configurations in the design. This is an horizontal distinction, because it implies the existence of many building stages at the same level.

Moreover, Partial Reconfiguration requires the use of a PXML file, defining each configuration, and to modify the UCF file adding the area constraints which limits the mapping of the Reconfigurable Areas.

The Java class which leads the entire Partial Reconfiguration process is the PRManager, which is the class where most of the MaxCompiler modifications have been implemented.
It is possible to divide the build process in three main stages:

- In the first one the Java classes defining Kernels and Managers are translated to VHDL code, and they are synthesized producing the netlist files. This part accounts for a quite small fraction of the entire build process time;

- In the second stage, MaxCompiler performs some checks for verifying the design correctness and MPPR, pr_verify and Bitgen are called, producing the full bitstream and all the partial bitstreams. This stage is the most expensive in terms of time, as the Xilinx tools used require much time;

- The third and final stage consist in the creation of the .max file and in the compilation of the source code, done with gcc/g++. Time taken varies greatly depending on the complexity of the source code, but it is usually significantly quicker than the previous stage.

The following subsections will explain in more detail the three stages.

6.2.1 First Stage

When the user calls build(Configuration ... conf) the PRManager starts the entire build process. First, it checks if Kernels inside the same Reconfigurable Block are compatible, by calling the method verifyPRKernelCompatibility. This method checks the number and type of the inputs and outputs of the Kernels in the same Reconfigurable Block, and it performs many other checks to see if the resource usage of the Kernels is the same. Examples of other checks are the ones to verify that Kernels in the same Reconfigurable Area have the same number, name and type of scalar inputs and outputs (the same holds for mapped ROMs and RAMs), and that inside a Reconfigurable Area there are not any Global Clock Buffer. Other two methods perform important checks, not related to possible compatibility issues among Kernels in the same Reconfigurable Area:
• The first one consist in making sure that the same Kernel it is not part of two different Reconfigurable Blocks, because this would lead to name clashes when calling the function max_reconfig_partial_bitstream. Obviously, this does not mean that it is not possible to employ two Kernels with exactly the same functionalities in two different Reconfigurable Area, in that case one must simple instantiate two times the same Kernel using two different names;

• The second one checks that in every Configuration there is not a Reconfigurable Block with no Kernel assigned to it. An empty Kernel is an error from a logical point of view, if the user goal is to create an "idle" Kernel, she should instantiate a Pass-Through Kernel, namely a Kernel which pass as output the same stream it receives as input, without any modification/delay.

It is also important to avoid that more than one Kernel is assigned to a given Reconfigurable Block in the same Configuration (because a Reconfigurable Block can contain just one Kernel at the same time), and this check is implemented in the method which adds Kernel to a Reconfigurable Block.

Then, it is called the function to transform the Java Kernels and Manager code into VHDL. In the static part, there is a Black Box for each Reconfigurable Area. Black Boxes have the same type and number of inputs and outputs of the Kernels inside that Reconfigurable Block: this is the first step needed to exploit Hierarchical Design. After the VHDL code has been produced, XST and NGCBuild are called: the static part and the Kernels in the Reconfigurable Blocks are synthesized independently from each others, producing a large number of netlist files.

6.2.2 Second Stage

Then, calling the method buildConfigurations, it starts the second phase of the process. For each configuration, it is automatically created a PXML file which contains all the information needed to guide the next design steps:
• The static part of the design is set to "implement" only for the first configuration, all the following configuration will import it from the first configuration;

• The Reconfigurable Module (that correspond to the Kernels that are placed inside a Reconfigurable Block) are implemented only the first time they appear in a Configuration, the following times they will be imported.

Moreover, the Reconfigurable Area constraints specified when instantiating the Reconfigurable Blocks are included int the UCF file.

The next step consists in calling, for each configuration MPPR, pr_verify and Bitgen. MPPR (multiple place and route) runs Map and PAR different times with different cost tables. A cost table is a random seed to placement[75]. Its affect on placement allows for a multi-pass search through the possible distribution of results quality. This distribution of possible results approximates a Bell Curve where there are few exceptionally good results, few exceptionally bad results, and many results with average QOR. Cost tables are most often used to find the optimal result that allows an ambitious design to meet timing requirements, since different cost tables assign weighted values to relevant factors such as constraints, length of connection, and available routing resources, leading to different results. Once the best cost table for that design is found, it can be reused as long as design changes are small and good QOR continue. If not, it is necessary to search the Bell Curve for a new best cost table. The user can set the number of cost tables to use in the PRManager, in order to find the best placement result.

The pr_verify[45] utility is used to compare routed NCD files from two or more configurations created for a Partial Reconfiguration design to validate that all imported resources match. For this reason it is called for every Configuration but the first one, because when the first Configuration is build there is not yet any other Configuration to compare to. Needed resources include:

• **Global Clock Spines**: Each global clock must have clock spines routed within the same clock regions in all configurations;
- **Regional Clock Spines**: Each regional clock must have clock spines routed within the same clock regions in all configurations;

- **Proxy logic**: Proxy logic, although logically part of the static design, must be placed at the same locations within the Area Groups allocated for the Reconfigurable Partitions;

- **Partition Interfaces**: Each RP must have the same ports in and out of the RM in each configuration.

The `pr_verify` results are exported in `pr_verify_results.log`. If `pr_verify` does not fail, it means that it is possible to import the results from previous Configurations in the actual one. The last Xilinx tool called is Bitgen, which generates one full partial bitstream and as many partial bitstreams as the number of Reconfigurable Blocks.

### 6.2.3 Final stage

The final stage of the building process consists in creating the `.max` file and compiling the CPU code. The CPU code compilation did not require to be modified (it consists simply in C/C++ or FORTRAN code), while the `.max` file has now changed. The `.max` file for a static design contains an array storing the full BIT file. The BIT file is loaded into the DFE as soon as the application starts.

In the case of a design containing partially reconfigurable areas there is an array storing the full BIT file of the default Configuration, plus as many arrays as the number of Kernels that can be placed in a Reconfigurable Block, containing their partial bitstreams. Of course, even if the same Kernel is used in more than one Configuration, the `.max` fill will contain just one partial bitstream describing it. The partial bitstreams have the same name of the Kernel they are describing, and for this reason, in the host code, it is sufficient to specify the name of the Kernel we want to reconfigure for loading into the FPGA its partial bitstream: when the function `max_reconfig_partial_bitstream` is called, the selected partial BIT file is taken
from the \texttt{.max} file and loaded into the DFE.

\section*{6.2.4 State Machine}

The modification to the Maxeler infrastructure that have been done did not concern only the Java part of MaxCompiler. The function \texttt{max\_reconfig\_partial\_bitstream} is implemented in C in the SLiC interface, and its insertion required also to modify the IFPGA (described in Section 2.2.2.1).

The IFPGA is configured with a full bitstream stored in a flash memory when the device is turned on. In the IFPGA is implemented a VHDL state-machine\cite{76,77} which guides the configuration of the CFPGA: when it receives the command for configuring the CFPGA, it starts loading the full bitstream and manages the occurrence of errors.

To support PR it has been added a new command, to load the partial bitstream. It works in a slightly different way than the command for the full configuration, because of the distinctions between a full and a partial reconfigurations (e.g. the \texttt{DONE} signal is not asserted).

Thus, when calling from the host code the function \texttt{max\_reconfig\_partial\_bitstream} a command is passed to the VHDL state-machine in the IFPGA to load the selected partial bitstream. The IFPGA then reports if either the partial bitstream has been loaded with success or an error has occurred.

Since the full configuration of the CFPGA is carried out by the VHDL state machine in the IFPGA through the SelectMAP\cite{78} configuration interface, the PR makes use of the SelectMAP as well.

\section*{6.2.5 File system}

During the build process, MaxCompiler automatically creates a complex file system were all the intermediate build results are stored.
In the case of a partially reconfigurable design the file system is pretty complex, due to the many iterations of the Xilinx tools which every time take as input the results of some previous iteration (e.g. when a Configuration import some partition from a previous one).

Figure 28 represents a graphical example of the file system.

The main build directory contains the the sub-directory scratch, where all the building step are executed. Moreover, in that directory is run the synthesis of the static part. For each Kernel in a Reconfigurable Block it is created a subdirectory where it is synthesized independently. This correspond to the first stage of the build process. Then, it is created a directory for each Configuration, and there runs NGDBuild. Then, it is created a subdirectory for each Cost Table, where Map and PAR are called. Finally, when the best Cost Table has been found, pr_verify and Bitgen are run in the Configuration’s directory. All the Xilinx tools run in a directory called xilinx.pnr, to separate their input and outputs files from the rest of the files.

Since the build process is broken up in different stages performed in different directories, it is necessary, during the entire process, to provide the required files in the directories where they are supposed to be: for instance, both NGDBuild and MPPR requires the xpartition.pxml file, but they are run in different directories. This means that the xpartition.pxml file (which is the same for every Cost Table in the same Configuration) must be linked to all the directories where it is needed. In the same way the result of Map and PAR of the best Cost Table, which are stored in the Cost Table directory, are linked in the Configuration directory. This is done because when importing a partition the corresponding PXML file must include the directory containing the NGDBuild, Map and PAR results. Therefore, they must be present in the same directory, and it seemed more reasonable to link the best Cost Table results to its corresponding Configuration directory than linking the NGDBuild result to the winning Cost Table directory. All the files that need to be present in more than one directories are linked instead of copied, in order to reduce the used disk space.
Figure 28: File system example, with relative building phase on the right.
6.3 Summary

This Chapter described how the proposed methodology has been implemented. The first section of the Chapter focused on the user interface, showing that, from the user point of view, accessing Partial Reconfiguration is incredibly simple. PRManager looks very similar to a standard Custom Manager, the only difference for the user is that, in the case she wants to instantiate a Reconfigurable Partitions, she has to create a ReconfigurableBlock instead of KernelBlock, and that she has to specify the Configurations. Reconfigurable Blocks can be connected to other Blocks and to the host application in the Manager in the same way of Kernel Blocks.

The second section focused in lower-level details, explaining how the building flow is structured. In the case of a design containing Reconfigurable Blocks the building process is much more complex with respect to the standard case. Moreover, the building process takes more time, because Xilinx tools such as NGDBuild, MAP, Par and Bitgen must be run multiple times, one for each Configuration. The increase in complexity of the design flow results also in an increase in complexity of the structure of the directories where the build results are contained. While the user does not suffer of such increase in complexity, putting together a so complex structure presented several issues during the design phase.
CHAPTER 7

VALIDATION

The previous Chapter showed how the design flow presented in this work has been implemented. The fact that the work will be released in the next MaxCompiler release demonstrates how effective is the proposed solution. In any case, to show a practical example of the work, a demo application has been designed.

Dataflow computing performance can be maximized by targeting applications processing a large number of data of uniform type, performing repeatedly the same type of operations. A type of application which satisfies these requirements is a video filtering application. In fact, videos consist on a large set of pixels, that can be streamed to the DFE where it is possible to perform some simple operation. For these reasons, the application demo presented falls in the video filtering field.

The goal of this application is to show that now PR is supported and integrated in the Maxeler flow, and that building flow works as expected. This will allow, in the future, to develop much more complex and interesting applications, both from a commercial and an academic point of view.

7.1 The application: Canny edge detection

The application consists in a Canny\cite{79} edge detector, that applies sequentially four different filters to outline the edges of an image by finding sharp changes in the image brightness.

The applied filters are:

1. A Gray scale conversion filter, which transforms a color image in an image where only the intensity of the pixels is preserved, resulting in an image composed exclusively of shades of gray;
2. A *Gaussian Blur* filter, to reduce the image’s noise removing impurities;

3. A *Laplacian* filter, to detect the edges;

4. Finally, a *Threshold* filter, which refines the results.

These filters must be applied in the same order in which they have been listed, to produce the correct result. To better understand the result of the Canny edge detector Figure 29 shows two different images, where the first one is the original image and the second one the resultant image after the application of the Canny edge detector algorithm.

![Figure 29: Comparison between the original image (a) and the same image after the Canny edge detector filter has been applied (b). In the image (b) all the edges of the original image have been strongly emphasized.](image-url)
Edge detectors are an essential part of many computer vision systems: the edge detection process serves to simplify the analysis of images by drastically reducing the amount of data to be processed, while at the same time preserving useful structural information about object boundaries.

7.2 Implementation

This section will show how the Canny edge detector has been implemented. The first part of the section will give an overview of the application structure, while the following sections will provide more detailed informations about the implementation.

7.2.1 Structure overview

As in all the Maxeler applications, the code is divided in two parts:

- An *host* part which runs on the Intel CPU;
- An *hardware* part, which runs on the DFE.

The host code is written in C and it is responsible for opening the input video file, sending the data to the DFE, receiving the data from the DFE, and finally to write the result in a video. The hardware code consists on the implementation of a Manager and four Kernels, where each Kernel represents a different filter.

To exploit Partial Reconfiguration, the project contains two Reconfigurable Blocks:

- The block *A* can contain the modules representing the Gray Scale filter and the Laplacian filter;
- The block *B* can contain the modules representing the Gaussian Blur filter and the Threshold filter.
When the program starts, the block A is configured with the Gray Scale filter, while the block B is configured with the Gaussian Blur.

The program flow can be seen as composed of two iterations:

- In the first iteration, input data (the raw data) are streamed from the host code to the DFE, passing through block A and then block B. After that, block A is reconfigured with the Laplacian filter and block B is reconfigured with the Threshold filter;
- In the second iteration, the output of the first iteration flows through the DFE, passing through block A and the block B. The output of block B goes back to the host code, where it is written to a video file.

Figure 30 represents the entire flow, where the loop Host-DFE-Host is covered two times.

To better understand how the whole application works, some knowledge about the digital videos structure is needed: a digital image is made of pixels, and each pixel is the combination of the three components R, G, B, which represent the colors red, green and blue. Each one of this three component is encoded with a byte, which stores a value which goes from 0 to 255 and represents the intensity of the color in the pixel. This bytes can be referred as raw data, and for this reason we can think of a digital image as a matrix of raw data. On the other hand, a video can simply be looked as sequence of images, where each image is called a frame.

For this reason processing an image or processing a video (disregarding the complications inherent to a real-time applications) is basically the same thing: we can extract the frames from the video and processing them as "standard" images, and then putting together the processed images creating a new video.

### 7.2.2 Host Code

To manage the video opening and writing the OpenCV\textsuperscript{[80][81]} libraries have been used. OpenCV (Open source Computer Video) is an open source library for real time computer vision.
The usage of OpenCV is due to the fact that it strongly simplifies the phases of grabbing frames from a video and extracting the raw data from the frame pixels, and then to reconstruct a video starting from a sequence of frames.

In the `main` function of the host code, the first step is to open the input video calling a function named `loadVideo`. The function `loadVideo` extracts, iterating on the frames, the raw data that compose the image, and store them in an array.
Inside the function `loadVideo` the input video is opened calling the OpenCV function `cvCaptureFromAVI`. Subsequently, the frames are extracted one after the other by using the function `cvQueryFrame`. Each frame can be seen as a simple image, that is a 2D matrix of pixels. From each frame the raw data composing the pixels are extracted row by row, and finally stored in an array of 8 bits unsigned integers. This array will be the input of the first iteration of the DFE.

Listing 6 shows a portion of the main function, more precisely the part where the SLiC interface’s functions are employed to starts the DFE computations. Initially, the .max file is initialized, by calling the function `TestFilter_init()`. Then, the CFPGA is fully configured with the call of the function `max_load(maxfile,"*")`. At this point, an action[^48] structure is created, with fields defining for how long the Kernels must run, which are the inputs and the outputs, and the size of inputs and outputs. After that, the DFE starts its execution:

- The function `TestFilter_run(engine, &myaction)` starts the first iteration: input data are streamed to the DFE, the DFE performs its computation, and the output data are written back to an array in the host application;
- By calling two times the function `max_reconfig_partial_bitstream` the two Reconfigurable Blocks are partially reconfigured, with the two final Kernels. The DFE is ready to the second iteration;
- A new call to `TestFilter_run(engine, &myaction)` starts the second iteration. This time, the data streamed back to the host are those that will define the output video.

Finally, the function `max_unload(engine)` entails that the DFE is released by the CPU process and returns to the pool of DFEs managed by MaxelerOS.
Listing 6: A portion of the main code, where the SLiC functions are called.

Once the hardware code has finished to run, the output is used to build a new video file, calling the function writeVideo. The function writeVideo takes as input an array of raw data. As the first step, it calls the OpenCV function cvCreateVideoWriter, which creates a new Audio Video Interleave (AVI) file, with the codec XviD\textsuperscript{[82]}. This video file has the same frame rate of the video opened with loadVideo, and the frames have the same width and height of the frames of the input video. Then, it iterates through the RAW data array reconstructing sequentially the frames and writing them on the video file previously created.
So, as previously explained, the role of the host code is limited to the phases of video opening and writing. All the part regarding the actual filtering is implemented in the DFE.

7.2.3 Hardware Code

The hardware part of the application is composed of six files:

- Four Kernels, responsible for the arithmetic part of the DFE code, namely the filtering of the video;
- A Manager, which instantiates the Reconfigurable Blocks and the Kernels and manages the streams among the Kernels and the host application;
- A Builder, which contains the main function, instantiates the Manager, and starts the building process.

All this all files are written in Java with some additional features (e.g. improved operands overloading) and the termination is .maxj.

The remaining part of the section will be dedicated to the description of the DFE files, starting from the Kernels. The order in which Kernels are presented follows the same order in which the Kernels process the data.

7.2.3.1 GrayScale Kernel

The first Kernel crossed by the data flow is the Kernel implementing the Gray Scale conversion. Every pixel in an image is made of three component: the red component, the green component and the blue one. In the Gray Scale filter each of this three components values is set to their average:

\[ R_{new} = G_{new} = B_{new} = \frac{R_{old} + G_{old} + B_{old}}{3} \]  

(7.1)
In this way the intensity of the pixel is preserved, and its color varies from black at the weakest intensity to white at the strongest. Figure 31 shows the resultant image of applying the gray scale conversion filter to image (a) in Figure 29.

![Figure 31: Conversion to scale of gray of the Lena image.](image)

The second Kernel is the one implementing the Gaussian Blur.

### 7.2.3.2 Gaussian Blur Kernel

The Gaussian blur, as the name suggest, is the result of blurring an image by a Gaussian function. It is useful because it helps reducing the image noise (the variation of brightness or color information in images, usually due to electronic noise) and thus smoothing the image. Applying a Gaussian Blur to an image is the same as convolving the image with a Gaussian function, whose equation in the two dimensional case is:
\[ G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \]  

(7.2)

where \( x \) is the distance from the origin in the horizontal axis, \( y \) is the distance from the origin in the vertical axis, and \( \sigma \) is the standard deviation of the Gaussian distribution.

The result of this formula is a surface whose contours are concentric circles with a Gaussian distribution from the center point.

The entire image would need to be included in the calculation of each pixel’s value, but in this case a 3 \( \times \) 3 convolution matrix is applied to the image, where each pixel’s new value is set to a weighted average of that pixel’s neighborhood. The original pixel’s value receives the heaviest weight and neighboring pixels receive smaller weights as their distance to the original pixel increases, and in this way edges boundaries are well preserved.

The kernel of Gaussian values used is:

\[
\begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
1 & 2 & 1
\end{bmatrix}
\]

The original pixel is the one multiplied by the central value of the matrix (4), receiving the maximum weight. On the other hand, the most distant pixels are multiplied by 1, the smallest value in the matrix. The result is then averaged by dividing by 16, the sum of all the coefficients.

The third Kernel is the one implementing the Laplace filter.

### 7.2.3.3 EdgeLaplace Kernel

The edge points of an image can be detected by finding the zero crossings of the second derivative of the image intensity, but calculating the second derivative is very sensitive to noise. It is for this reason that before applying the edge detector filter it is used Gaussian blur, to
reduce the noise. In the edge detection algorithm based on the gradient, a pixel location is declared an edge location if the value of the gradient exceeds some threshold, because that is the point where there is a strong difference in intensity between two pixels. When the first derivative is at a maximum the second derivative is zero, and this is why the Laplacian method searches for zero crossings in the second derivative of the image to find edges.

Figure 32: Image (a) shows a generic function, in this case the arctangent function. Image (b) corresponds to the first derivative, with shows a maximum in the gradient when the x coordinate is equal to 0. The image (c) represents the second derivative (that it is possible to obtain by applying the Laplace operator), and it crosses the x axis in the same point of the maximum of the first derivative.
The filter used in the EdgeLaplace Kernel applies a $3 \times 3$ convolution matrix, with the following Laplacian kernel:

\[
\begin{bmatrix}
0 & 1 & 0 \\
1 & -4 & 1 \\
0 & 1 & 0
\end{bmatrix}
\]

If the result of applying the Laplacian kernel is lesser then 0, it is inverted (obviously it is not possible to define a pixel with a negative color value).

The fourth and last Kernel used, is that implementing the Threshold filter.

### 7.2.3.4 Threshold Kernel

The Threshold Kernel is the simplest among all the implemented filters: If the raw data represents a very dark color (lesser than 10), then it it is set to 0 (representing the black color), otherwise it is set to 255 (which represents the white color).

In this way, the resulting image is composed of just two colors: the image’s edges are white, while all the rest is black. Using this filter helps in obtaining are more defined image, where all the edges are emphasized with respect to the rest of the image.

### 7.2.3.5 Manager

The Manager, on the other hand, is responsible for instantiating the Kernels, the Reconfigurable Block, and defining the way in which Kernels and host code can communicate:
public class PartialFilterManager extends PRManager {
    PartialFilterManager(PartialFilterEngineParameters engineParameters) {
        super(engineParameters);
        getCurrentKernelConfig().optimization.setUseGlobalClockBuffer(false);
        GrayScaleKernel k1 = new GrayScaleKernel(makeKernelParameters("GrayScaleKernel"));
        GBKernel k2 = new GBKernel(makeKernelParameters("GBKernel"));
        ELKernel k3 = new ELKernel(makeKernelParameters("ELKernel"));
        ThresholdKernel k4 = new ThresholdKernel(makeKernelParameters("ThresholdKernel"));
        ReconfigurableBlock b1 = addReconfigurableBlock("reconf1", 0, 8, 0, 8, k1, k3);
        ReconfigurableBlock b2 = addReconfigurableBlock("reconf2", 0, 7, 0, 7, k2, k4);
        Stream x = addStreamFromHost("input");
        Stream z = addStreamToHost("output");
        b1.getInput("input") <= x;
        b2.getInput("input") <= b1.getOutput("output");
        z <= b2.getOutput("output");
    }
}

Listing 7: Manager code

In this particular case the Manager has three main tasks:

- Instantiating the four Kernels;
- Defining two Reconfigurable Blocks, specifying in which area they must be placed and which Kernels they can contain. For this application it is enough to declare a single clock region per Reconfigurable Block;
- Managing the communication between the host code and the DFE through the streams.

The last class composing the hardware code is the builder.

7.2.3.6 Builder

```
public class PartialFilterBuilder {
    public static void main(String[] args) throws Exception {
        PartialFilterEngineParameters engineParameters = new PartialFilterEngineParameters(args);
        PartialFilterManager m = new PartialFilterManager(engineParameters);
        ReconfigurableAreaConfiguration c1 = new ReconfigurableAreaConfiguration("conf1");
        ReconfigurableAreaConfiguration c2 = new ReconfigurableAreaConfiguration("conf2");
        c1.setKernel("reconf1", "GrayScaleKernel");
        c1.setKernel("reconf2", "GaussianBlurKernel");
        c2.setKernel("reconf1", "EdgeLaplaceKernel");
        c2.setKernel("reconf2", "ThresholdKernel");
        m.build(c1, c2);
    }
}
```

Listing 8: Builder code

The Builder contains the main function, instantiates the Manager, and defines the Configurations. There are two Configurations, corresponding to the two iterations through the DFE:

- The first Configuration assigns the GrayScale Kernel to the Reconfigurable Block `reconf1`, and the GaussianBlur Kernel to the Reconfigurable Block `reconf2`;
• The second Configuration assigns the EdgeLaplace Kernel to the Reconfigurable Block reconf1, and the Threshold Kernel to the Reconfigurable Block reconf2.

Finally, it calls the build(Configuration ... conf) Manager’s method, starting the entire building process.

Since the Configuration c1 is the first one passed to the build method, this means that when the application starts the full bitstream loaded into the CFPGA will contain the GrayScale Kernel inside the Reconfigurable Block reconf1, and the GaussianBlur Kernel in the Reconfigurable Block reconf2.

Besides the full bitstream, the .max file will contain a partial bitstream per Kernel.

7.2.4 Considerations

This application demonstrates that what has been presented in Chapter 6 works as expected. PR is now perfectly integrated in the MaxCompiler, and it is easily accessible to the user. Moreover, tests showed that partially reconfiguring a single clock region takes about 40-50 milliseceonds, a small time even for applications requiring high speed performance.
Presented work enhances the potentialities of Maxeler platforms introducing the concept of Partial Dynamic Reconfiguration. Maxeler systems are designed to increase dramatically the performance of a given application. Partial Reconfiguration allows to build more flexible systems, able to modify their behavior at run-time to adapt to the environment and to user needs.

Besides improvements in the application’s raw speed (depending on the application), a partially reconfigurable design does not suffer for FPGA capacity constraints as much as a static one. The presented work has proven to be effective and successful: not only the the video filtering application showed in Chapter 7 demonstrates that the solution works as expected, but the fact that the entire work will be released in the next MaxCompiler release is a clear certification of its effectiveness. Despite this, the work presents, obviously, space for improvements.

Among the imperfections, the only one which constitutes a true limitation is the fact that when a Reconfigurable Block is being reconfigured, all the Kernels in the system must be blocked. The reason for that is that when a Reconfigurable Block is being reconfigured, Xilinx requires to issue a local reset to the Reconfigurable Logic of that Reconfigurable Block as soon as it has been reconfigured.

When this thesis has been written Maxeler platforms did not allow to reset only a single Kernel, the only possibility was to issue a logical reset regarding all the Kernels. As a consequence, when one of the Reconfigurable Block is being reconfigured, all the other parts of the circuit have to stop their computation and halt.

One of the greatest strength of PR is the possibility to keep on running the rest of the system while reconfiguring just a portion of it, and for this reason the Maxeler requirement to
stop the entire device while reconfiguring a Reconfigurable Block is clearly a drawback in terms of performance. This limitation derives from a Maxeler platform limitation, and solving this problem would require a not trivial work.

Therefore, the most important step to improve the MaxCompiler flow regarding PR would be to modify the system by inserting a call to a function to reset a single Kernel, internally to MaxCompiler. Such a function would be probably useful not only for Partial Reconfiguration but also for different applications in which some Kernel (but no every Kernel) needs to be logically reset.

Secondly, an area that requires improvements is the one relative to the instantiation of a Reconfigurable Block.

In fact the decision about which portion of the FPGA must be reserved to a given Reconfigurable Block is the only not fully automated part in the MaxCompiler flow when using Partial Reconfiguration.

As shown in Section 6.1 the user, when she decides to instantiate a Reconfigurable Block, must specify the area constraints in terms of clock regions. For this reason, even if the Kernels that will be placed in that Reconfigurable Block are very simple and small (in terms of occupied area), at least an entire Clock Region will be required for them to be instantiated. Furthermore, the number of clock regions (18 in the case of Xilinx XC6VSX475T, the CFPGA in the MAX3 board) is an upper bound on the number of Reconfigurable Blocks in the design.

In a design where capacity constraints are an issue, or where there is a large number of Reconfigurable Blocks, assigning at least one clock region per Reconfigurable Block can be a limitation. It would be possible to allow for defining the Reconfigurable Blocks area constraints in more precise terms, but a better solution would be to hide the area constraints definition to the user automatizing this process.

This would be possible adding to the MaxCompiler flow tools as ReBit[83]. ReBit, among various other functionalities, allows to generate the UCF. Given an input Extensible Markup
Language (XML) file describing how the design is partitioned (similar to the PXML) ReBit is able to produce an UCF file containing the Reconfigurable Blocks’ area constraints.

In addition to the limitations due to the Xilinx FPGAs, when using Partial Reconfiguration with Maxeler platforms there are some other more subtle problems, due to how *Mapped Registers* and *Mapped Memories* are implemented.

- "Mapped Register" is the name given to the FPGA registers storing the values of scalar inputs and outputs;
- Mapped Memories correspond to ROMs and RAMs, that can be used inside the Kernels.

Both Mapped Registers and Mapped Memories, despite the fact they are used inside a Kernel, are placed by MaxCompiler externally to the Kernel. This means that they are part of the full bitstream instead of the partial ones.

Therefore, Kernels in the same Reconfigurable Block must have the same type and number of Mapped Memories and Mapped Registers, otherwise the application would fail when a Kernel try to access them.

The modification to MaxCompiler needed to overcome this limitation is not trivial: Mapped Registers (MR) and Mapped Memories (MM) are external to the Kernels because they represent a way to allow the dynamic (for their values can be modified at run-time) communication between Kernels and the Manager. So, it would be necessary to revise their entire implementation.

Anyway, it does not represent a strong limitation from the user point of view: simply, all the Kernels in the same Reconfigurable Block must employ a super set of all the scalar I/O and of the ROMs and RAMs, even if not using them. The only drawback of this solution is the increase in the occupied area of the design and therefore the reduced maximum theoretical utilization.

Finally, the concept of PR could be extended to all the Maxeler DFE supporting it, namely to all the Maxeler DFE composed of FPGAs which support it.
In fact, the presented work targets MaxWorkstation, whose DFE is the MAX3 board, and for this reason MaxWorkstation is in this moment the only Maxeler platform which allows to exploit Partial Reconfiguration. The modifications done to MaxCompiler would theoretically allow to employ PR on all the Maxeler platforms, but there are differences on the way the DFE is partially reconfigured: for instance, if some Maxeler DFE does not contain any IFPGA, then the reconfiguration mechanism based on the IFPGA’s VHDL state machine is not more suitable.
CITED LITERATURE

1. http://www.spiralgen.com/technology.html. [Citations at pages vi and 3]

2. Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2. [Citation at page xii]


6. AMD accelerated processors for desktop PCs. http://www.amd.com/us/products/desktop/apu/Pages/apu.aspx. [Citation at page xii]

7. NVIDIA Tegra 3. http://www.nvidia.com/object/tegra-3-processor.html. [Citation at page xii]


9. What is an FPGA. http://www.altera.com/products/fpga.html. [Citation at page xii]

11. Xilinx Inc. http://www.xilinx.com. [Citation at page xii]
13. Cray Inc. http://www.cray.com/Home.aspx. [Citation at page xii]
15. Maxeler Technologies Ltd. website. http://www.maxeler.com. [Citation at page xiii]
22. Cray XK7 specifications. http://www.cray.com/Products/XK/Specifications.aspx. [Citation at page 1]


32. Inc., X.: Two Flows of Partial Reconfiguration: Module Based or Difference Based, 2003. [Citation at page 11]


34. Hymel, R., George, A.D., Lam, H.: Evaluating Partial Reconfiguration for embedded FPGA applications. [Citation at page 12]

35. Virtex-6 family overview, 2012. [Citations at pages 13 and 34]

36. Inc., X.: *Virtex-6 FPGA Configurable Logic Block*. Xilinx Inc., 2012. [Citation at page 15]


40. Dyer, M., Wirz, M.: Reconfigurable system on FPGA, 2002. [Citation at page 20]

42. Tapp, S.: Configuration quick start guidelines. Tech. rep., 2003. [Citation at page 22]


49. Maxworkstation. http://www.maxeler.com/products/desktop/. [Citation at page 33]

50. Ferrandi, F., Santambrogio, M.D., Sciuto, D.: A design methodology for dynamic reconfiguration: The Caronte architecture. In IPDPS, IEEE Computer Society. [Citation at page 46]


61. Pell, O., Bower, J., Dimond, R., Mencer, O., Flynn, M.J.: Finite difference wave propagation modeling on special purpose dataflow machines. *IEEE Transactions on Parallel and Distributed Systems*, 2012. [Citation at page 52]


70. Ltd, M.T.: *MaxCompiler Manager Compiler Tutorial*. Maxeler Technologies Ltd., 2012. [Citation at page 62]


72. Inc., X.: *ISE In-Depth Tutorial*. Xilinx Inc., 2011. [Citation at page 75]

73. Inc., X.: Virtex-6 FPGA Clocking Resources, 2012. [Citation at page 78]

74. Zeh, C.: Incremental design reuse with partitions, 2007. [Citation at page 82]

75. [Citation at page 92]


78. Virtex-6 fpga configuration, 2012. [Citation at page 94]


80. The OpenCV tutorials, 2012. [Citation at page 101]


82. XviD video codec website. [Citation at page 104]
83. Bonetto, A., Cazzaniga, A., Durelli, G., Pilato, C., Sciuto, D., Santambrogio, M.D.: An open-source design and validation platform for reconfigurable systems. In *22nd International Conference on Field Programmable Logic and Applications*, 2012. [Citation at page 114]


VITA

Matteo Mastinu

EDUCATION

• Maturità Scientifica (equivalent to High School Diploma) at Liceo Scientifico Mariano IV d’Arborea (final grade: 100/100), 2007.

• Laurea (equivalent to Bachelor of Science) in Ingegneria Informatica at Politecnico di Milano (final grade: 105/110), 2010.

• Attending Laurea Magistrale (equivalent to Master of Science) in Ingegneria Informatica at Politecnico di Milano (GPA: 29.59/30).

• Attending Master of Science in Computer Science at University of Illinois at Chicago (final grade: 3.85/4.00).

PROFESSIONAL EXPERIENCE

• Visiting student at Maxeler Technologies Ltd., 1 Down Place London W6 9JH, UK (Reference: Oliver Pell), 2012.