Multiobjective Reconfiguration-Aware Scheduling
on FPGA-Based Heterogeneous Architectures

BY
ENRICO ARMENIO DEIANA
B.S., Politecnico di Milano, Milan, Italy, September 2011

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, 2015

Chicago, Illinois

Defense Committee:
John Lillis, Chair and Advisor
Wenjing Rao
Marco D. Santambrogio, Politecnico di Milano
To my family

and friends
ACKNOWLEDGEMENTS

There are many people who made this work possible. They helped me during the difficulties and drove me through the right path to solve the different issues that a research thesis such as this one encounters.

I want to thank Marco Rabozzi for being an active part in the realization of this work and for being just brilliant. I have always lived by “work smarter not harder”, but he taught me how to “work smarter and harder”, I will never forget it.

A thank you to my advisor, Marco Domenico Santambrogio, who believed in me and gave me guidance, inspiration and support through these years. Thank you for all the beautiful experiences that you made possible, thank you for the NECST group conferences, thank you for the support during the UIC experience.

Another thank you goes to all the PoliMi-UIC students that shared with me this great oversea experience. Especially Emanuele Del Sozzo and Luca Graziani with whom I spent the Spring 2014 semester as research assistant.

Thank you to Prof. Jon A. Solworth for giving me the opportunity to work as research assistant in an interesting project with exceptionally talented people.

Last but not least, a huge thank you to my family for supporting me spiritually and financially all these years in Milan and in the USA.
TABLE OF CONTENTS

CHAPTER                      PAGE
1 INTRODUCTION                1
   1.1 Heterogeneous architectures . . . . . . . . . . . . . . . . . . . 1
   1.1.1 Multiprocessor Systems on Chip . . . . . . . . . . . . . . . . . 2
   1.1.2 Field Programmable Gate Arrays . . . . . . . . . . . . . . . . . 3
   1.2 The concept of reconfiguration . . . . . . . . . . . . . . . . . . 5
   1.3 Addressed problem . . . . . . . . . . . . . . . . . . . . . . . . . 7

2 SCHEDULING PROBLEM          9
   2.1 Problem description . . . . . . . . . . . . . . . . . . . . . . . . 9
   2.2 Formal definition . . . . . . . . . . . . . . . . . . . . . . . . . 15
   2.2.1 Target architecture definition . . . . . . . . . . . . . . . . . . 15
   2.2.2 Application definition . . . . . . . . . . . . . . . . . . . . . . . 16
   2.2.3 Problem definition . . . . . . . . . . . . . . . . . . . . . . . . . 17

3 STATE OF THE ART            18
   3.1 Exact approaches . . . . . . . . . . . . . . . . . . . . . . . . . . 18
   3.1.1 Banerjee et al. ILP formulation . . . . . . . . . . . . . . . . . . 20
   3.1.2 Redaelli et al. ILP formulation . . . . . . . . . . . . . . . . . . 21
   3.1.3 Fekete et al. exact algorithm . . . . . . . . . . . . . . . . . . 22
   3.2 Heuristic approaches . . . . . . . . . . . . . . . . . . . . . . . . . 23
   3.2.1 Banerjee et al. KLFM heuristic algorithm . . . . . . . . . . . . 23
   3.2.2 Banerjee et al. PARLGRAN heuristic algorithm . . . . . . . . . . 24
   3.2.3 Redaelli et al. Napoleon heuristic algorithm . . . . . . . . . . . 26
   3.3 Final notes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4 ACO SCHEDULER               29
   4.1 ACO metaheuristic . . . . . . . . . . . . . . . . . . . . . . . . . . 29
   4.2 Modifications to the ACO-based scheduler . . . . . . . . . . . . . . . 32

5 PROPOSED SCHEDULER          33
   5.1 MILP formulation . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
   5.1.1 Sets and parameters . . . . . . . . . . . . . . . . . . . . . . . . . 34
   5.1.2 Variables identification . . . . . . . . . . . . . . . . . . . . . . . 37
   5.1.3 Semantic constraints . . . . . . . . . . . . . . . . . . . . . . . . . 39
   5.1.4 Problem constraints . . . . . . . . . . . . . . . . . . . . . . . . . 43
   5.1.5 Cutting planes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
   5.1.6 Objective function . . . . . . . . . . . . . . . . . . . . . . . . . . 46
<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Iterative scheduler</td>
<td>49</td>
</tr>
<tr>
<td>5.3 Computation of the normalization terms</td>
<td>50</td>
</tr>
<tr>
<td>5.4 Computation of the number of hardware regions</td>
<td>56</td>
</tr>
<tr>
<td>6 EXPERIMENTAL EVALUATION</td>
<td>59</td>
</tr>
<tr>
<td>6.1 Synthetic benchmark generation</td>
<td>59</td>
</tr>
<tr>
<td>6.2 Iterative scheduler and ACO scheduler comparison</td>
<td>59</td>
</tr>
<tr>
<td>6.2.1 Schedule makespan</td>
<td>60</td>
</tr>
<tr>
<td>6.2.2 Schedule peak power</td>
<td>64</td>
</tr>
<tr>
<td>6.2.3 Schedule energy consumption</td>
<td>67</td>
</tr>
<tr>
<td>6.3 Pareto curve of the multiobjective iterative scheduler</td>
<td>69</td>
</tr>
<tr>
<td>6.4 Floorplan of the schedule</td>
<td>71</td>
</tr>
<tr>
<td>6.4.1 Floorplannable schedule</td>
<td>72</td>
</tr>
<tr>
<td>6.4.2 Feasible/Infeasible noise</td>
<td>76</td>
</tr>
<tr>
<td>7 CONCLUSIONS</td>
<td>82</td>
</tr>
<tr>
<td>7.1 Contributions and limitations</td>
<td>82</td>
</tr>
<tr>
<td>7.2 Future work</td>
<td>83</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>85</td>
</tr>
<tr>
<td>VITA</td>
<td>88</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. SCHEDULE MAKESPAN COMPARISON</td>
<td>61</td>
</tr>
<tr>
<td>II. ALGORITHM RUNTIME W.R.T. MAKESPAN</td>
<td>62</td>
</tr>
<tr>
<td>III. AVERAGE SCHEDULE MAKESPAN IMPROVEMENT</td>
<td>63</td>
</tr>
<tr>
<td>IV. AVERAGE ALGORITHM RUNTIME W.R.T. MAKESPAN</td>
<td>63</td>
</tr>
<tr>
<td>V. SCHEDULE PEAK POWER COMPARISON</td>
<td>64</td>
</tr>
<tr>
<td>VI. ALGORITHM RUNTIME W.R.T. PEAK POWER</td>
<td>65</td>
</tr>
<tr>
<td>VII. AVERAGE SCHEDULE PEAK POWER IMPROVEMENT</td>
<td>66</td>
</tr>
<tr>
<td>VIII. AVERAGE ALGORITHM RUNTIME W.R.T. PEAK POWER</td>
<td>66</td>
</tr>
<tr>
<td>IX. SCHEDULE ENERGY COMPARISON</td>
<td>67</td>
</tr>
<tr>
<td>X. ALGORITHM RUNTIME W.R.T. ENERGY</td>
<td>68</td>
</tr>
<tr>
<td>XI. AVERAGE SCHEDULE ENERGY IMPROVEMENT</td>
<td>69</td>
</tr>
<tr>
<td>XII. AVERAGE ALGORITHM RUNTIME W.R.T. ENERGY</td>
<td>69</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>11</td>
</tr>
<tr>
<td>3.</td>
<td>12</td>
</tr>
<tr>
<td>4.</td>
<td>14</td>
</tr>
<tr>
<td>5.</td>
<td>25</td>
</tr>
<tr>
<td>6.</td>
<td>26</td>
</tr>
<tr>
<td>7.</td>
<td>53</td>
</tr>
<tr>
<td>8.</td>
<td>54</td>
</tr>
<tr>
<td>9.</td>
<td>55</td>
</tr>
<tr>
<td>10.</td>
<td>56</td>
</tr>
<tr>
<td>11.</td>
<td>71</td>
</tr>
<tr>
<td>12.</td>
<td>73</td>
</tr>
<tr>
<td>13.</td>
<td>75</td>
</tr>
<tr>
<td>14.</td>
<td>76</td>
</tr>
<tr>
<td>15.</td>
<td>77</td>
</tr>
<tr>
<td>16.</td>
<td>78</td>
</tr>
<tr>
<td>17.</td>
<td>79</td>
</tr>
<tr>
<td>18.</td>
<td>81</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ACO</td>
<td>Ant Colony Optimization</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block RAM</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GPCPU</td>
<td>General Purpose Central Processing Unit</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HLA</td>
<td>High Level Analysis</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>ICAP</td>
<td>Internal Configuration Access Port</td>
</tr>
<tr>
<td>ILP</td>
<td>Integer Linear Programming</td>
</tr>
<tr>
<td>IOB</td>
<td>Input/Output Block</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>KLFM</td>
<td>Kernighan-Lin Fiduccia-Matthysse</td>
</tr>
<tr>
<td>LP</td>
<td>Linear Programming</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS (Continued)

LUT Look-Up Table. vii, 4

MILP Mixed-Integer Linear Programming. vii, 12, 19, 29, 32–35, 37, 39, 45, 46, 49, 50, 52, 54, 56, 57, 59, 60, 63, 82–84

MPSoc Multiprocessor System on Chip. vii, 1–3, 82

PDR Partial Dynamic Reconfiguration. vii, 6, 20, 21, 27

PR Partial Reconfiguration. vii

RAM Random Access Memory. vii, 2, 83

RCPS Resource Constrained Project Scheduling. vii, 9, 29

SoC System on Chip. vii, 1–3

TGFF Task Graph For Free. vii, 59

TSP Travelling Salesman Problem. vii, 29
SUMMARY

Designing applications for heterogeneous systems, like Multiprocessor Systems on Chip with Field Programmable Gate Arrays is a complex task. In order to exploit all the capabilities of these systems, such as Partial Dynamic Reconfiguration and hardware acceleration, the designer still has to develop large parts of the system unassisted, establishing the design choices mostly on his experience.

In this work we present a Mixed Integer Linear Programming formulation for mapping and scheduling applications on heterogeneous and reconfigurable devices taking into account partial dynamic reconfiguration, module reuse and reconfiguration prefetching.

Since the better these techniques and features are exploited, the better is the resulting schedule, and taking into account that the space of the solutions is considerable, it is necessary the use of automatic tools in order to help the system designer in building the best possible application design with respect to his needs, which often are not limited to improve the overall execution time (like most of the other schedulers in the literature do), but to also consider the peak power and energy consumption of the design. This means addressing a specific Resource Constrained Project Scheduling problem, which takes into consideration the reconfiguration aspects with a focus on power and energy metrics that are becoming crucial in designing applications targeting heterogeneous architectures.

Following this view we propose an iterative off-line scheduler, based on a MILP model, which provides the following contributions:
SUMMARY (Continued)

- the possibility to specify different performance metrics among execution time, peak power and energy consumption, or a linear combination of them;

- the possibility to consider the features and techniques offered by current FPGA-based devices (partial dynamic reconfiguration, module reuse, reconfiguration prefetching);

- the opportunity to easily trade-off the quality of the desired scheduling solution with respect to the execution time of the algorithm.

The remainder of this work is organized as follows:

- Chapter 1 introduces the main concepts of heterogeneous architectures, reconfiguration and an overview of the problem that we are addressing;

- Chapter 2 gives a more detailed description of the scheduling problem;

- Chapter 3 describes the most significant approaches in the literature that address the scheduling problem;

- Chapter 4 describes an already existing ACO-based scheduler and the modifications we made in order to make it comparable with our approach;

- Chapter 5 defines in details the heuristic and exact algorithms that we are proposing and the MILP formulation that constitutes the base of our approach;

- Chapter 6 reports the results of the benchmark that we used to test our approach;

- Chapter 7 highlights the contributions and limitations of our approach and proposes possible future works.
CHAPTER 1

INTRODUCTION

In this chapter we introduce the reader to the topics we will address in the thesis and the motivations that led us to this work.

In Section 1.1 we will introduce the concepts of heterogeneous architectures, reconfigurable computing, System on Chips (SoCs), Multiprocessor System on Chips (MPSoCs) and Field Programmable Gate Arrays (FPGAs). Section 1.2 describes the concept of reconfiguration, its features and the different kinds of it that are present in reconfigurable devices. In the end, Section 1.3 defines the scheduling problem that we are addressing, with peculiar care on the necessity of having an automatic tool in order to help the system designer.

1.1 Heterogeneous architectures

As the demand for high performance in processing systems was growing, the clock frequency increased until 2006, when the main manufacturers of processing units (i.e., Intel, AMD, IBM etc.) could not increase the processor frequency anymore because of power and overheating issues. This led to multiprocessor systems, the use of dedicated hardware (i.e., Digital Signal Processors (DSPs), Application-Specific Integrated Circuits (ASICs)) and reconfigurable logic (i.e. FPGAs) to overcome the previous limitations. Since we aim at scheduling applications for heterogeneous architectures, we mainly focus our attention on high performance architectures (e.g., Maxeler supercomputers [1]) and, mostly, on embedded reconfigurable devices (e.g.,
heterogeneous MPSoC [2]). Heterogeneous high performance computing systems are usually a combination of multiprocessor units and reconfigurable logic such as FPGAs interconnected in some way (e.g., via PCI express). While, heterogeneous embedded devices often have multiprocessors and reconfigurable logic on the same chip (in this case they are heterogeneous MPSoCs).

In the following subsections we will describe more in details our main target architecture (i.e., heterogeneous MPSoCs) and the used reconfigurable logic (i.e., FPGA).

1.1.1 Multiprocessor Systems on Chip

A heterogeneous SoC is a combination of general purpose and application specific processing units.

The general purpose model is at the base of common personal computers. This type of architecture was first proposed by the mathematician John von Neumann in 1945 [3]. It is based on the concept of stored-program computing that stores instructions and data (i.e., a program) in memory, usually a Random Access Memory (RAM), and by mean of its Instruction Set Architecture (ISA) executes the program. The peculiarity of the general purpose model is its ability of executing arbitrary programs by changing the stored instructions and data in memory.

On the other hand, the application specific model waives the flexibility of the general purpose model in order to achieve better performances with low power consumption. This model is used by the so called ASIC [4] that are Integrated Circuit (IC) built for executing only one specific functionality.
Combining the two approaches leads to a heterogeneous SoC that, in case of multiple processing elements, becomes a MPSoC. These devices have different kinds of processing units on their chip, like general purpose processors and a reconfigurable logic or DSPs. If along with the general purpose processors there is a reconfigurable IC, like an FPGA, then we have a reconfigurable device. Furthermore, these devices usually have different connections and peripherals like USB, FireWire and ethernet connected together by a bus.

Notice that there are also homogeneous MPSoCs that only have one kind of processing units, like general purpose processors.

1.1.2 Field Programmable Gate Arrays

FPGAs are tightly connected with the concept of reconfiguration, unlike ASICs. Indeed, an FPGA can be reconfigured any number of times, while an ASIC is built to perform a single functionality that has been fixed at chip design time. However, both are integrated circuits and are configured using a Hardware Description Language (HDL).

An FPGA can be seen as a set of resources distributed on a matrix. These resources are:

- Input/Output Blocks (IOBs) that are blocks on the edges of the FPGA matrix and are used to input and output signals from and to the IC;

- interconnections that are used to connect the different resources on the FPGA and outside it through the IOBs;

- Configurable Logic Blocks (CLBs) that are the basic blocks realizing the logical functionalities;
• DSPs that are more complex blocks specialized in digital signal processing;

• Block RAMs (BRAMs) are blocks of memory that can be easily and quickly accessed by the tasks on the FPGA.

Notice that CLBs can be further decomposed in:

• Look-Up Tables (LUTs) whose output can realize any logical functionality given different inputs. They usually have 4 inputs and 1 output or 6 inputs and 2 outputs;

• latches that are circuits used to store information;

• multiplexers that, given a set of inputs, select what to output.

BRAMs, CLBs, DSPs are not randomly distributed on the FPGA, but they are grouped in tiles as it is shown in Figure 1..
Finally, an FPGA is called homogeneous if it only has IOBs, CLBs and interconnections among the previous resources. Otherwise, if it also contains BRAMs and DSPs, is called heterogeneous.

1.2 The concept of reconfiguration

Reconfiguration can be categorized with respect to many of its features. In order to reconfigure an FPGA it is necessary to change a file, the content of such file is called bitstream.
and is stored in the reconfiguration memory of the FPGA. Depending on the reconfigurable device, the minimum portion of area that can be reconfigured at once determines whether the reconfiguration is complete or partial. Indeed, if it is needed to change the whole bitstream for reconfiguring the FPGA, we have complete reconfiguration. While, if we can change only part of it, we have partial reconfiguration. Partial reconfiguration can also be one dimensional (i.e., an entire slice of the FPGA has to be reconfigured) or two dimensional (i.e., specific rows and columns of the FPGA matrix can be selected for reconfiguration) [5].

The reconfiguration on FPGAs is mainly performed within the FPGA itself through an interface that, for Xilinx devices, is called Internal Configuration Access Port (ICAP) [6]. However, for some FPGAs, it may happen that an external device takes care of the reconfiguration from outside the FPGA. Notice that some FPGAs can perform multiple reconfigurations at the same time using different reconfiguration controllers.

Moreover, the reconfiguration can be static (i.e., once the FPGA has been configured it is necessary to restart it for changing its configuration) or dynamic (i.e., the FPGA configuration can be changed at runtime).

Finally, the granularity of the resources that can be reconfigured can change with respect to the considered FPGA. If we can change the single CLB or IOB configuration, then we have smallbits reconfiguration. Otherwise, if we can change groups of FPGA resources, then we have module based reconfiguration. In this work we will focus on module based, partial, dynamic reconfiguration, also known as Partial Dynamic Reconfiguration (PDR).
Reconfiguration is needed when two subsequent scheduled tasks are mapped on the same hardware region on the FPGA and they perform different operations (i.e., do not share the same implementation). A hardware region (or module) is a rectangular portion of the FPGA that, in the application lifetime, holds one or more application tasks. In our case, since we consider module based reconfiguration, a module dictates the reconfiguration granularity.

Along with the reconfiguration go the concepts of reconfiguration prefetching and module reuse (or resource sharing) that are exploited in order to get a better schedule. The reconfiguration prefetching is used to improve the concealment of the reconfiguration task in the schedule, since it implies that the task requiring the reconfiguration does not have to start right after the end of the reconfiguration. This means that the reconfiguration task can be scheduled at any time between the end of the old task and the start of the new task that required the reconfiguration, in this way the schedule has more freedom on when performing the reconfiguration and then it can better hide the reconfiguration task.

Module reusing is exploited when two successive scheduled tasks on the same hardware region use the same implementation (and then the same resources). In this case it is sufficient to configure the FPGA only once and then the reconfiguration is not needed.

1.3 Addressed problem

In this work we address two different problems, but tightly intercorrelated at the same time: the mapping problem and the scheduling problem.

The mapper decides for each task which implementation and component has to be chosen, while the scheduler set the starting time of each task. The decisions made by the mapper
and the scheduler have to be compliant to the given objective function, which, again, is a linear combination of three different metrics: schedule execution time, peak power and energy consumption.

If not specified otherwise, in the remainder of the thesis, we will refer to the conjunct mapping and scheduling problem as simply scheduling.

We will further describe the scheduling problem with more details in Chapter 2.
CHAPTER 2

SCHEDULING PROBLEM

The scheduling problem or, more precisely, the Resource Constrained Project Scheduling (RCPS) problem is well known in the literature [7, 8]. There are many variants of it, so in this chapter we are going to describe the one that we are addressing. Indeed, in our case, besides the scheduling of tasks on limited hardware and software resources, we need to consider also the mapping problem and the role of reconfiguration that often has to be hidden by the execution of other tasks in order to further improve the schedule with respect to its objective function.

2.1 Problem description

Since we have to deal with the mapping problem and reconfigurations among tasks, we are not approaching the standard RCPS. So, in this section we are going to describe more in details the variant that we are addressing.

The inputs for the schedule on heterogeneous architectures consist of:

- an architectural template (e.g., ZedBoard with Zynq-7000 SoC) containing information on the General Purpose Processors (e.g., dual core ARM Cortex-A9), the programmable logic (e.g., Xilinx XC7Z020 [9]) and information on the reconfiguration controller such as the reconfiguration speed with respect to the area that has to be reconfigured and the needed power to perform the reconfiguration;
• a description of the application in terms of a taskgraph, that is a Directed Acyclic Graph (DAG) where each node represents a task of the application and the directed edges represent the data dependencies among tasks;

• an objective function with respect to optimize the schedule.

Each task can have several hardware and software implementations with different execution time and peak power. The hardware implementations are also characterized by the resources they need on the programmable logic, such as the number of BRAMs, CLBs, DSPs and their relative bitstream size. Notice that different tasks that fulfill the same functionalities can have more than one implementation in common.

For each task the scheduler has to choose an implementation, mapping it on a hardware region (i.e., a portion of the programmable logic) or a software one (i.e., one of the available processors) and then scheduling it in a time slot according to a user defined objective function. Moreover, the scheduler must take into account the reconfigurations that may be required among consequent tasks mapped on the same reconfigurable region. This means that the scheduler has to consider if the reconfiguration can be hidden by other tasks in execution and how much it costs (in terms of time, power and energy) with respect to the given objective function. Notice also that the reconfiguration is not put within the task that requires it, but it is considered as a different task with its own properties (i.e., reconfiguration prefetching is exploited). So, in order to clarify how we plan to deal with reconfiguration tasks, let us make a simple example.

Given the taskgraph, as a Directed Acyclic Graph (DAG), in Figure 2., we represented each computing region where the corresponding task is mapped with a color.
If task 1 is mapped into a hardware region and task 4, that has been scheduled after task 1, is going to be mapped into the same region with a different implementation with respect to the one used by task 1, then a reconfiguration task is placed between task 1 and task 4 as it is shown in Figure 3.
As we will see more in details in Chapter 5, we propose an iterative approach based on a Mixed-Integer Linear Programming (MILP) formulation of the problem that simultaneously maps and schedules the tasks of the application. As we stated before, our approach considers the reconfigurations as dynamic tasks with their own dependencies that are added to the taskgraph when required.
In order to reduce the problem complexity we do not directly perform the floorplan of the reconfigurable regions, however, the resource requirements of the tasks mapped in hardware are taken into account to minimize the probability to achieve an infeasible solution and avoid schedules that exceed the available resources of the FPGA that are clearly infeasible.

In the end, let us remember that the schedule objective function is a linear combination of three performance metrics (i.e., makespan, energy consumption and peak power of the schedule) with respect to the schedule is optimized. These metrics are computed in the following way:

- **makespan**: is the total execution time of the schedule;
- **energy**: is the sum of the energy (i.e., $energy = makespan \cdot power$) of the chosen implementations of the scheduled tasks;
- **power**: is the maximum sum of peak power of overlapping tasks in time.

For clarifying the meaning of the makespan and peak power metrics, let us make another example. Given the schedule in Figure 4., we can see how the schedule makespan is its length (in time) of 6889 clock cycles, while the maximum peak power is indicated by the second line and is 8169 mW.
Figure 4.: Schedule metrics
2.2 Formal definition

In this section we will describe more formally the model of the target architecture in Subsection 2.2.1, of the application in Subsection 2.2.2, of the mapping and scheduling problem in Subsection 2.2.3.

2.2.1 Target architecture definition

The architecture on which the application is going to run, can be simply modeled with a set of processing elements \( P \). Each element \( p \in P \) has a set \( Q \) of resources that can be categorized in:

- renewable resources \( R \), that can be used again after a task has finished its execution (e.g., if the FPGA allows the reconfiguration to be performed, then the resources of the reconfigurable logic can be considered renewable);

- nonrenewable resources \( N \), that once used by a task can not be used again (e.g., if special policies, like static allocation, are used for the local memory of a processor, then that resource is nonrenewable).

So, for each component \( p_k \in P \) there is a total amount of available resources \( P^q_k \) of type \( q \in Q \) (i.e., the capacity of that processing element with respect to that kind of resource). Hence, tasks can exploit only those components whose capacity satisfies their resource demand.
2.2.2 Application definition

We model the application as a DAG $G = (T, E)$. Where the set of nodes $T$ represents the application tasks, and the set of edges $E \subseteq T \times T$ represents the precedences among tasks (i.e., an edge $e(t1, t2) \in E$ means that task $t2$ can be executed only after the end of task $t1$).

Now, since different tasks could perform the same operations, we introduce the concept of job $j$ and the set of all the application jobs $J$. A job can have more than one task that executes the same operations of the job, while a task can be associated with only one job.

An implementation $i$ of a task $t$ is defined by its needed resources $q \in Q$, execution time and peak power. Notice that a task, and then a job, can have different implementations that could not be available on every processing element $p \in P$ (e.g., a task could not have hardware implementations if it can not be synthesized). We define the set of all the available implementations of the application as $I$.

Given the sets that we have defined above, we need to introduce on those sets some functions. The first one is $\gamma : I \rightarrow P$ that, given an implementation, returns the corresponding processing component $p_k = \gamma(i)$. In this way it is not possible, by construction, to assign jobs to components that are not in the architecture, and then the number of available resources can not be exceeded. Notice that, in order to model the sharing of resources, different jobs can share more than one implementation.

In order to associate the execution time and power for an implementation we need to define the functions $\delta_{\text{time}} : I \rightarrow \mathbb{N}$ and $\delta_{\text{power}} : I \rightarrow \mathbb{N}$. The implementation energy can be easily calculated as $\delta_{\text{time}}(i) \cdot \delta_{\text{power}}(i)$. 
In the end, we need the function $\sigma : I \times Q \rightarrow \mathbb{N}$ that for each pair implementation, resource returns the quantity of the given resource required by that implementation.

### 2.2.3 Problem definition

Now we are ready to formally define the two problems we are addressing: the mapping and scheduling of tasks. The mapping can be formalized in this way $M : T \rightarrow I$ (i.e., for each task $t$ the chosen implementation $i$ is returned and then $\gamma(i)$ is the component where the task will be executed). The scheduling, instead, can be formalized as $S : T \rightarrow \mathbb{N}$ (i.e., for each task $t$ associates the corresponding start time). 
CHAPTER 3

STATE OF THE ART

We can split the algorithms proposed in the literature in two main categories. The first one includes the heuristic approaches [10–18], while the second one includes the exact approaches [10, 12, 13, 19–23]. We will discuss the most significant exact and heuristic algorithms in, respectively, Section 3.2 and Section 3.1.

Some of the proposed approaches solve not only the mapping and scheduling problem, but also the floorplanning problem (i.e., placing the hardware regions, where hardware implementations will be instantiated, on the FPGA) at the same time. We, on the other hand, are going to address directly only the mapping and scheduling problem. Again, since these two problems are tightly correlated, if not specified otherwise, we will refer to the problem of simultaneous mapping and scheduling as simply scheduling.

3.1 Exact approaches

Most of the exact approaches proposed in the literature make use of a Linear Programming (LP) model. Such models are characterized by sets and parameters (i.e., the data that the problem needs), variables (i.e., what the model has to compute), linear constraints (i.e., the description of the problem and semantic of the variables) and a linear objective function (i.e., the metrics with respect to optimize the solution).
The canonical form of an LP model is:

\begin{align*}
\text{objective function} : & \quad \max (c^T x) \\
\text{constraints} : & \quad Ax \leq b \quad (3.1) \\
& \quad x \geq 0
\end{align*}

While the standard form is:

\begin{align*}
\text{objective function} : & \quad \max (c^T x) \\
\text{constraints} : & \quad Ax = b \quad (3.2) \\
& \quad x \geq 0
\end{align*}

It is always possible to go from the canonical to the standard form introducing some new variables.

The vector \( x \) represents the problem variables, \( b \) and \( c \) are vectors of coefficients, \( A \) is the matrix of the constraints coefficients and \((.)^T\) is the transpose operator.

An LP model can be further specified as follows. If all the variables in \( x \) have the integer constraint (i.e., \( x \in \mathbb{Z}^n \)), then we have an Integer Linear Programming (ILP) model. While, if only some of the variables in \( x \) are under the integer constraint, then we have a MILP model.

Among the exact approaches in the literature, there are some of them that we want to deepen \([10, 12, 13, 19, 20]\).
3.1.1 Banerjee et al. ILP formulation

The approach proposed by Banerjee et al. in [10] is an ILP formulation that takes into account PDR and reconfiguration prefetching with a single reconfigurator able to perform only one dimensional reconfiguration. PDR is considered, since the algorithm takes into account the possibility to put different tasks, with different implementations, on the same hardware region in different time instants. Reconfiguration prefetching is also taken into account, since the reconfiguration task is not considered within the task that needs the reconfiguration, but as a standalone task. During the scheduling phase they also take into consideration the floorplanning problem, in order to build a feasible solution.

The model considers a certain number of tasks that can have both hardware and software implementations. The resources considered are a general purpose processor, used for software implementations, and the columns that form the programmable logic. In fact, since only one dimensional reconfiguration is allowed, only the number of columns is required to define the resources needed by a hardware implementation. Again, the formulation also considers the reconfiguration prefetching; indeed, if for a task is chosen a hardware implementation, then two variables keep track of the beginning of the reconfiguration and the beginning of the actual task (that has to be after the reconfiguration). Furthermore, for tasks mapped on hardware regions, the leftmost column that the task occupies is used for non overlapping constraints and hardware resource availability is also respected.

The major limitations of this approach are the use of a single one dimensional reconfigurator and the fact that module reuse is not exploited at all. The first limitation is not in line with
current heterogeneous architectures that exploit two dimensional reconfiguration. The second limitation makes the model losing the chance to reuse the same hardware implementation among tasks that share it and hence avoiding reconfiguration. Both assumptions make the retrieved schedule suboptimal with respect to what it could be exploiting the previous features. Moreover, like most of all the exact approaches, finding the optimal solution is very time consuming.

3.1.2 Redaelli et al. ILP formulation

The model proposed in [19] overcomes the limitations of the previous ILP formulation by Banerjee et al. [10]. It introduces the possibility to reuse an already configured module without reconfiguring it first, if the previous implementation on that hardware region was equal to the one that has to be placed. This model also uses all the already cited features and techniques of FPGA-based devices, such as PDR and reconfiguration prefetching.

In order to exploit module reuse, the model utilizes two binary variables. The first one is set to 1 if two tasks have the same implementation, while the second one is set to 1 if a task makes use of an already configured hardware region. However, this approach still relies on one dimensional regonfigurator. In a later work, Redaelli et al. [13] introduced in the ILP model the variables that represent also the rows of the programmable logic (and not only the columns). So, the position of a task is set taking the leftmost column and the bottommost row. In this way it is possible to keep track of the position of tasks during time and reconfigure only the region occupied by the task instead of the whole slice of programmable logic that includes all the rows for a set of given columns.
The major limitation of the proposed ILP model is the dimension that even a simple problem can have using this formulation. Indeed, each time instant (i.e., clock cycle) is taken into consideration by two binary variables $p_{i,h,k,m}$ and $t_{i,h}$. The first one is set to 1 if task $i$ is on the reconfigurable logic at time $h$ and on the region $(k,m)$, which is identified by the bottommost row $k$ and leftmost column $m$. The second one is set to 1 if the reconfiguration of task $i$ starts at $h$.

Since the number of these variables grows with the time horizon of the schedule, using two binary variables for every time instant makes the model extremely big and difficult to solve. Furthermore, the estimated time horizon has to be provided in advance and then it has to be conservative (i.e., the worst possible schedule makespan). This makes even simple and small problems like a five rows by five columns FPGA and ten tasks problem very time consuming to solve (i.e., up to 27 days).

Moreover, the problem is oversimplified since tasks can only be placed on the programmable logic. Indeed, general purpose processors are not taken into account here.

In a following work [12] they extended the model to take into account also software processors. However, the time required to solve a small instance of the problem is still considerable and the objective function optimizes only the schedule length.

3.1.3 **Fekete et al. exact algorithm**

The algorithm proposed by Fekete et al. in [20] is an exact approach that does not make use of an ILP formulation. It allows to find the optimum with respect to two metrics. The first one is the usual minimization of the schedule makespan, while the second one is used to find the
minimum number of FPGA resources needed to schedule the tasks in a given amount of time. The algorithm models the tasks as three dimensional rectangles. The first two dimensions $x, y$ represent the area that the task occupies on the programmable logic, while the third dimension $t$ is the time spent by the task on the FPGA; in this way it avoids the overlapping of tasks. Even in this case scheduling and floorplanning problems are solved at the same time and also the communication among tasks is taken into account.

However, the algorithm supports only one dimensional reconfiguration and does not allow reconfiguration prefetching, since the reconfiguration time is simply added to the task execution time. Moreover, an arbitrary number of reconfigurations can take place at the same time, which is not a realistic assumption, since often reconfigurations are the bottleneck of the system.

3.2 Heuristic approaches

The algorithms that use some kind of heuristic to address the scheduling problem usually return suboptimal schedules in a relatively small amount of time. There are numerous types of heuristic algorithms, some of them are simply list-based (i.e., they schedule a task with respect to a list that is ordered using some priority measure), others exploit metaheuristics such as tabu search, simulated annealing, Ant Colony Optimization (ACO) or genetic algorithms.

Among the heuristic approaches in the literature, we want to discuss [10–13] in particular.

3.2.1 Banerjee et al. KLFM heuristic algorithm

Banerjee et al. propose, in the same paper of the already introduced ILP formulation [10], a heuristic approach to overcome the considerable amount of time needed by the exact approach.
In this work a Kernighan-Lin Fiduccia-Mattheyses (KLFM) heuristic [24, 25] is introduced. The algorithm separates the hardware software partitioning phase from the scheduling phase. For each task that is ready to be scheduled (i.e., its parents have already been scheduled) a move is made (i.e., an implementation is chosen) following the heuristic and then it is scheduled using a simple list-based scheduler. The list-scheduler uses a single metrics for the priority function of software tasks, which is the path length (i.e., the critical path priority function); while, for hardware tasks, besides the longest path, other metrics are taken into account to compose the priority function, and these metrics are: the area occupied by the task, the earliest start time and earliest finish time of the task: \( \text{priority function} = -A \cdot \text{columns} - B \cdot \text{EST} + C \cdot \text{path length} - D \cdot \text{EFT} \). The floorplanning problem is addressed too, so that only feasible schedules can be returned.

The limitations of this approach are the same of the ILP formulation discussed in the same paper [10]. So, even in this case only one dimensional reconfigurator is taken into consideration and module reuse is not exploited.

3.2.2 Banerjee et al. PARLGRAN heuristic algorithm

In another work Banerjee et al. [11] propose a different heuristic approach, called PARLGRAN, to address the scheduling problem. The algorithm tries to add as many copies of data parallel tasks as possible, in order to maximize the number of concurrent operations on the same input data and then minimize the schedule makespan. Besides that, it mainly focuses on two antifragmentation techniques for placing tasks without wasting space on the reconfigurable logic. In this way it ensures feasible solutions and further reduces the makespan of the
schedule. The first antifragmentation technique is a simple fragmentation reduction that tries to put each hardware region on opposite sides of the FPGA. The second one exploits slack in reconfiguration controller trying to reorganize the placement for avoiding useless latencies before a reconfiguration. In Figure 5. and Figure 6. is shown, respectively, a portion of a schedule before and after the use of the second antifragmentation technique (note that E1, E2, E3 are application tasks, while R2, R3, R4 are reconfiguration tasks).

Figure 5.: Schedule before antifragmentation
The main limitations of this algorithm are, again, the use of one dimensional reconfigurator and the unexploited possibility to reuse previous configuration of the programmable logic in different time instants.

3.2.3 Redaelli et al. Napoleon heuristic algorithm

Redaelli et al. in the same paper of the already discussed ILP formulation [13], propose a heuristic algorithm, called Napoleon, to overcome the time consuming issues of the ILP model.
As for the exact approach, mapping, scheduling and flooplanning are addressed at the same time in order to keep track of the limited resources on the FPGA and then returning a feasible, placeable solution. Features and techniques such as two dimensional reconfiguration, module reuse and reconfiguration prefetching are all exploited to produce the best schedule possible with respect to the heuristic. The algorithm also allows the use of more than one reconfigurator. Moreover, antifragmentation techniques are used to further reduce the schedule length. In particular it uses the farthest placement criteria that aims to place each hardware region as far as possible from the center of the reconfigurable logic, so that bigger regions can be placed more easily occupying the center of the FPGA.

However, as for the ILP formulation in [13], the tasks can be scheduled only on the programmable logic, so general purpose processors are not considered. In [12] they extended the heuristic, but still it only considers the schedule length as optimization metrics.

3.3 Final notes

It is worth to notice that all the previous proposed approaches currently use as quality metrics the schedule makespan only, while other performance metrics such as peak power or energy consumption are not considered. On the other hand, [18, 23] focus on power minimization. The approach in [23] uses an ILP model with time constraints on the schedule makespan and targets heterogeneous architectures composed by a General Purpose Central Processing Unit (GPCPU) and co-processors, so reconfigurable logic (i.e., FPGAs), and hence PDR, are not considered. While [18] uses cluster techniques for mapping operations to microprocessors or
ASICs. Even in this case though, programmable logic components and reconfiguration are not taken into account.

Furthermore, as we will see more in detail in Chapter 5, we do not address the floorplanning problem, so it might happen that the schedule we return is not feasible. Notice that we keep into consideration the total amount of hardware resources in order to avoid solutions that are clearly infeasible, but, for geometrical reasons about the placement of hardware regions, the scheduled hardware implementations could not be placed on the FPGA anyway. However, state of the art floorplanner such as [26, 27] can be used to verify the feasibility of our solution. In case the hardware tasks of the schedule would not be placeable on the FPGA, we can simply gradually diminish the total available resources (i.e., BRAMs, CLBs, DSPs) of the programmable logic given to the scheduler (giving the floorplanner more flexibility for placing the regions) until we can fit all the hardware regions. We will consider this issue more in detail in Chapter 6 Section 6.4.
CHAPTER 4

ACO SCHEDULER

The ACO metaheuristic has been initially used to solve the Travelling Salesman Problem (TSP) [28]. From then, it has been utilized to address other combinatorial optimization problems, such as the RCPS problem. Indeed, in literature, there are many examples that use ACO for solving the scheduling problem, for instance [29–34].

In this chapter we will focus on the state-of-the-art ACO-based scheduler [30]. In Section 4.1 we briefly describe how the ACO metaheuristic works, while in Section 4.2 we introduce our modifications to the work in [30]. Indeed, since in Chapter 6 we will directly compare our MILP-based iterative scheduler with this approach [30], we had to slightly modify it adding the power and energy metrics as quality measure for the schedule (since, previously, it considered only the reduction of the schedule makespan). Hence, as in our approach, we developed the possibility to define a linear combination of schedule makespan, peak power and energy consumption as objective function for [30].

4.1 ACO metaheuristic

ACO is a naturally inspired algorithm that tries to simulate the behavior of an ant colony searching for food. Each path that an ant follows from the nest to the food is a possible solution of the problem that the metaheuristic is addressing. The goodness of a path is given by its length. Each ant chooses a random direction from the nest and, following the chosen path, it
deposits a pheromone trail. Since pheromones decay as time goes by, the shortest path will have more pheromones than the others, so that the other ants are more prone to follow that path (i.e., the best solution) than other ones and then more pheromones will be released into that path. So, the amount of pheromones on a path is a measure of the goodness of that path for reaching the objective.

At each iteration of the algorithm, a generation of \( N \) agents (i.e., the ants) is created. Each agent explores the solution space and, incrementally, build a solution computing all the possible choices at each step. Then, these choices are sorted with respect to the value of two heuristics:

- a local heuristic that assigns a score to each choice considering only the current state of the exploration;
- a global heuristic that basically models the pheromone trail and then measures the goodness of the choices taken so far.

To each choice is associated a certain amount of pheromones that is represented by a probability. These values are stored into a matrix, called the pheromone matrix.

When the algorithm starts its execution, the pheromone matrix has uniformly distributed values (i.e., all the choices have equal probability to be chosen); after each iteration the matrix is updated reinforcing the good choices (i.e., higher amount of pheromones) and penalizing the others (i.e., lower amount of pheromones).

When it is time for the agent to make a choice, a probability for each choice is computed as follows:
\[ p_{x,y} = \frac{(\tau_{x,y})^\alpha \cdot (\eta_{x,y})^\beta}{\sum_{l \in \Omega_x} (\tau_{x,l})^\alpha \cdot (\eta_{x,l})^\beta} \] (4.1)

Where \( x \) represents the current iteration, \( y \) is the candidate choice, \( \eta \) is the local heuristic value related to the problem (computed when a probability is generated) and \( \tau \) is the global heuristic value (related to the pheromones). The terms \( \alpha \) and \( \beta \) weight, respectively, the global and local heuristic and then affect the choice of the agent. In the end \( \Omega_x \) represents the set of all the available choices at iteration \( x \).

At the end of each iteration only \( K \) out of \( N \) agents of the generation (with \( K \leq N \)) are considered for updating the pheromone matrix. The pheromones are updated using the following formula:

\[ \tau_{x,y} = (1 - \rho) \cdot \tau_{x,y} + \epsilon \]

Where \( \rho \) is the evaporation rate (i.e., how much the amount of pheromones decays) and \( \epsilon \neq 0 \) is a term that keeps track of the goodness of a solution and then avoids penalization for good choices. The best solution (i.e., the best path) is then identified when the global heuristic dominates the local one.

It is worth to notice that Equation 4.1 computes only the probability of admissible choices, avoiding those ones that violate a constraint and then reducing the infeasible solutions.
4.2 Modifications to the ACO-based scheduler

Since we had the chance to get the source code of the ACO-based scheduler described in [30] (it has been developed at the Politecnico di Milano), we decided to adapt it in order to make our MILP-based scheduler and the ACO based scheduler comparable. For getting a fair comparison between our approach and the ACO one, we had to match the interfaces. In fact, we had to add the peak power and energy consumption metrics computation and the possibility to specify a more complex objective function as a linear combination of the already available schedule makespan metrics and the new peak power and energy consumption metrics.
CHAPTER 5

PROPOSED SCHEDULER

We propose two algorithms, both making use of the same MILP formulation, that solve the scheduling problem. The first is an heuristic algorithm, while the second one looks directly for the optimal solution.

Since the size of the model rapidly increases with respect to the number of tasks, the resulting formulation becomes challenging to solve. To overcome this issue, the first algorithm iteratively schedules only a subset of the application tasks at each iteration, keeping the model relatively small and easier to solve. This iterative algorithm can be tuned to schedule simultaneously any number of tasks of the application with respect to the provided objective function.

The second algorithm is a particular case of the first one. In fact, if all the application tasks are provided, it will find the optimal overall schedule. However, this option is usually time consuming for taskgraphs bigger than five nodes. For this reason, we added the possibility to warm start the MILP model providing as input an initial solution that can be taken from the iterative algorithm proposed or from any other heuristic such as [30]. This solution is then improved towards the optimum with respect to the given objective function.

Each instance of the problem is solved by using a state-of-the-art solver called Gurobi [35], which gives you the opportunity to stop the searching for the optimal solution at any time or to set a fixed time limit.
5.1 MILP formulation

In this section we present the MILP model used by the iterative and exact algorithms. Since the model is quite substantial we will present it following this order: first we introduce the constants (i.e., the parameters and sets), second the variables, third the semantic constraints related to the description and meaning of the previous variables and then the cuts constraints used for reducing the solution space. In Subsection 5.1.6 we add the real variables, parameters and constraints that are solely needed to define the objective function.

5.1.1 Sets and parameters

In order to define the MILP model we need to introduce several sets related to the problem:

\[ T := \text{set of tasks to schedule}; \]

\[ RT := \text{set of reconfiguration tasks}; \]

\[ AT := \text{set of all tasks}; \]

\[ I^s := \text{set of software implementations}; \]

\[ I^h := \text{set of hardware implementations}; \]

\[ I := \text{set of all the available implementations}; \]

\[ C^s := \text{set of software components}; \]

\[ C^h := \text{set of hardware components}; \]

\[ C := \text{set of all the available components}; \]

\[ P := \text{set of tasks’ dependencies (i.e., } (t1, t2) \in P \text{ if and only if } t2 \text{ depends directly on } t1); \]
$P^+ :=$ the transitive closure of $P$ (i.e., $(t_1, t_2) \in P^+$ if and only if $t_2$ depends on $t_1$);

$R :=$ set of FPGA resources (i.e., CLB, DSP, BRAM).

Notice that we do not know beforehand which is the number of reconfiguration tasks that will be performed in the final schedule. However, the set $RT$ must have a fixed number of elements that cannot vary during the optimization of the MILP model. In order to solve this issue, we consider the worst case scenario in which each task, except for the first one, requires a reconfiguration before its execution, so that overall $|T| - 1$ reconfigurations are performed.

Within the set $RT$ we consider $|T| - 1$ elements, while a special binary variable $(rt_{rt,t})$ for each element will be used to determine if the element represents a required reconfiguration or not.

The following are the parameters needed for the MILP model:

$T_{max} :=$ maximum time for the execution of the schedule;

$T_{c} :=$ minimum time unit;

$time_i :=$ execution time of implementation $i \in I$;

$power_i :=$ power consumption of implementation $i \in I$;

$energy_i :=$ energy consumption of implementation $i \in I$;

$map_{t,i,c} := 1$ if task $t \in T$ can be mapped on component $c \in C$ with implementation $i \in I$, 0 otherwise;

$res_{i,r} :=$ resources of type $r \in R$ required by implementation $i \in I^h$;

$maxRes_r :=$ number of resources of type $r \in R$ within the FPGA;
\(bit_r\) := average bitstream size for a resource of type \(r\);

\(bit_{\text{max}}\) := maximum bitstream size for reconfiguration;

\(T_{\text{rec}}\) := reconfiguration time for each unit of bitstream;

\(P_{\text{rec}}\) := power consumption for reconfiguration tasks.

In order to simplify the description of the formulation, it is useful to define the following sets derived from \(\text{map}_{t,i,c}\):

\(TIC\) := set of triplets \((t, i, c)\) where \(t \in T, i \in I, c \in C\) such that \(\text{map}_{t,i,c} = 1\);

\(TI\) := set of couples \((t, i)\) where \(t \in T, i \in I\) such that \(\exists c \in C : \text{map}_{t,i,c} = 1\);

\(TC\) := set of couples \((t, c)\) where \(t \in T, c \in C\) such that \(\exists i \in I : \text{map}_{t,i,c} = 1\).

Moreover, starting from the definition already presented, we are able to compute additional sets that will be exploited to give a better characterization of the model variables:

\(CP\) := component precedence set: contains all the couples of tasks \((t_1, t_2) : t_1, t_2 \in T\) such that it is possible to schedule \(t_2\) right after \(t_1\) on the same hardware component;

\(OT\) := overlapping tasks set: contains all the couples of tasks \((t_1, t_2) : t_1, t_2 \in AT\) such that there exists a schedule in which \(t_1\) and \(t_2\) overlap in time;

\(CT\) := compatible tasks set: contains all the couples of tasks \((t_1, t_2) : t_1, t_2 \in T\) such that both tasks have at least one common hardware implementation (\(\exists i \in I_h : (t_1, i), (t_2, i) \in TI\)).

More formally the sets can be computed as:
\[ CP = \{(t_1, t_2) \mid t_1, t_2 \in T \land t_1 \neq t_2 \land (t_2, t_1) \notin P^+ \land \\
(\exists c \in C^h \mid (t_1, c), (t_2, c) \in TC)\} \]

\[ OT = \{(t_1, t_2) \mid t_1, t_2 \in AT \land t_1 \neq t_2 \land \sim (t_1, t_2) \in RT) \land \\
(t_1, t_2) \notin P^+ \land (t_2, t_1) \notin P^+\} \quad (5.1) \]

\[ CT = \{(t_1, t_2) \mid t_1, t_2 \in T \land t_1 \neq t_2 \land \\
(\exists i \in I^h, c \in C^h \mid (t_1, i, c), (t_2, i, c) \in TIC)\} \]

Notice that for the computation of \( OT \) we took into account a single reconfigurator, so that no two reconfiguration tasks can overlap in time by definition.

### 5.1.2 Variables identification

Thanks to the sets and parameters described in Subsection 5.1.1, we are now able to introduce the variables required for the MILP model:

\[ b_t := \forall t \in AT: \text{ real variable in the range } [0, T_{max}] \text{ defining the begin time of task } t; \]

\[ e_t := \forall t \in AT: \text{ real variable in the range } [0, T_{max}] \text{ defining the end time of task } t; \]

\[ mic_{t,i,c} := \forall (t, i, c) \in TIC: \text{ binary variable set to } 1 \text{ if task } t \text{ is mapped to component } c \text{ with implementation } i; \]

\[ mi_{t,i} := \forall (t, i) \in TI: \text{ binary variable set to } 1 \text{ if task } t \text{ is assigned to implementation } i; \]

\[ mc_{t,c} := \forall (t, c) \in TC: \text{ binary variable set to } 1 \text{ if task } t \text{ is mapped to component } c; \]
$oc_{c,r} := \forall c \in C^h, r \in R$: real non negative variable ($\geq 0$) defining the amount of resources of type $r$ needed by hardware component $c$;

cp_{t1,t2} := \forall (t1, t2) \in CP$: binary variable set to 1 if task $t2$ is executed right after task $t1$ on the same hardware component;

cft_{t,c} := \forall (t, c) \in TC : c \in C^h$: binary variable set to 1 if task $t$ is the first task executed on hardware component $c$;

rtt_{rt,t} := \forall t \in T, rt \in RT$: binary variable set to 1 if task $t$ requires reconfiguration $rt$ prior to its execution;

rtc_{rt,c} := \forall rt \in RT, c \in C^h$: binary variable set to 1 if reconfiguration task $rt$ is performed on hardware component $c$;

bitc_c := \forall c \in C^h$: real non negative variable ($\geq 0$) defining the bitstream size for hardware component $c$;

ba_{t1,t2} := \forall (t1, t2) \in OT$: binary variable set to 1 if task $t1$ begins after the beginning of task $t2$ or at the same time of $t2$;

bb_{t1,t2} := \forall (t1, t2) \in OT$: binary variable set to 1 if task $t1$ begins before the end of task $t2$ or at the end of $t2$;

bo_{t1,t2} := \forall (t1, t2) \in OT$: real variable in the range $[0, 1]$ set to 1 if the beginning of task $t1$ overlaps in time with task $t2$ (i.e., task $t1$ begins when $t2$ is in execution);
\( mibo_{t1,t2,i} := \forall(t1,t2) \in OT, (t2,i) \in TI: \) real variable in the range \([0,1]\) set to 1 if \( bo_{t1,t2} = 1 \) and \( mi_{t2,i} = 1 \).

All the time intervals considered in the model are closed on the left and open on the right, meaning that instant \( b_t \) represents the first time instant in which task \( t \) is in execution, while instant \( e_t \) represents the first instant in time right after the end of the execution of task \( t \).

As an example, if the time domain is discretized into clock cycles, we have \( T_\epsilon = 1 \) while the computation of a task \( t \) with \( b_t = 2 \) and \( e_t = 5 \) is performed during clock cycles 2, 3 and 4.

Moreover, in order to speed up the execution time of the MILP solver, variables \( oc_{c,r} \), \( bitc_c \), \( bo_{t1,t2} \), \( mibo_{t1,t2,i} \) are declared as real even though their values should be integer. This can be done without changing the semantics of the model thanks to the constraints in Subsection 5.1.3 that relate these variables to the other integer variables of the formulation.

### 5.1.3 Semantic constraints

The following are the constraints that are needed to guarantee the semantics of the model variables.

Guarantees the semantics for \( bb \), \( ba \) and \( bo \):

\[
\forall(t1,t2) \in OT : \]

\[
b_{t1} \geq e_{t2} - bb_{t1,t2} \cdot T_{max} \]

\[
b_{t1} \leq b_{t2} - T_\epsilon + ba_{t1,t2} \cdot T_{max} \quad (5.2)
\]

\[
bo_{t1,t2} \geq ba_{t1,t2} + bb_{t1,t2} - 1
\]
Ensures that a task is mapped exactly on one implementation and one component:

$$\forall t \in T : \sum_{(t,i,c) \in TIC} mic_{t,i,c} = 1 \quad (5.3)$$

Computes $mi$ and $mc$ from $mic$:

$$\forall (t,i) \in TI : m_{i,t} = \sum_{(t,i,c) \in TIC} mic_{t,i,c}$$

$$\forall (t,c) \in TC : m_{c,t} = \sum_{(t,i,c) \in TIC} mic_{t,i,c} \quad (5.4)$$

Computes the end of a task with respect to the selected implementation:

$$\forall t \in T : e_t = b_t + \sum_{(t,i) \in TI} mic_{t,i} \cdot time_i \quad (5.5)$$

Ensures that there is at most one first task executed on a hardware component:

$$\forall c \in C^h : \sum_{(t,c) \in TC} cft_{t,c} \leq 1 \quad (5.6)$$

Relates $cft$ with $mc$ (i.e., if a task is the first on a hardware component, it must be mapped to that component):

$$\forall (t,c) \in TC \mid c \in C^h : cft_{t,c} \leq mc_{t,c} \quad (5.7)$$
Relates \( cp \) with \( mc \) (i.e., \( cp_{t_1,t_2} = 0 \) if \( t_1 \) and \( t_2 \) are mapped on different components):

\[
\forall (t_1,t_2) \in CP, \forall c_1 \in C : \\
mc_{t_1,c_1} + \sum_{(t_2,c_2) \in TC : c_2 \neq c} mc_{t_2,c_2} + cp_{t_1,t_2} \leq 2
\] (5.8)

Relates \( cft \) with \( cp \) and \( mc \) (i.e., if a task is mapped to a hardware component then it is the first task or another task is executed before it):

\[
\forall (t,c) \in TC | c \in C^h : \\
mc_{t,c} \leq cft_{t,c} + \sum_{(t_2,t) \in CP} cp_{t_2,t}
\] (5.9)

Relates \( cp \) with the scheduling of the tasks:

\[
\forall (t_1,t_2) \in CP : \\
b_{t_2} \geq e_{t_1} - (1 - cp_{t_1,t_2}) \cdot T_{max}
\] (5.10)

Ensures that a hardware component occupies not less than the resources needed by the tasks mapped on it:

\[
\forall c \in C^h, \forall r \in R, \forall t \in T : \\
oc_{r,c} \geq \sum_{(t_i,c) \in TIC} mic_{t_i,c} \cdot res_{i,r}
\] (5.11)
Computes the bitstream size of a hardware component:

\[ \forall c \in C^h : \text{bit}_{c} = \sum_{r \in R} o_{c,r} \cdot \text{bit}_r \]  

(5.12)

Ensures that a task requires at most one reconfiguration and that a reconfiguration configure no more than one task:

\[ \forall rt \in RT : \sum_{t \in T} rtt_{rt,t} \leq 1 \]  

(5.13)

\[ \forall t \in T : \sum_{rt \in RT} rtt_{rt,t} \leq 1 \]

Guarantees the semantic of rtc:

\[ \forall rt \in RT, \forall c \in C^h, \forall t \in T | (t,c) \in TC : \]  

\[ rt_{c,t} \geq rtt_{rt,t} + mc_{t,c} - 1 \]  

(5.14)

Ensures that a reconfiguration task ends always after its beginning:

\[ \forall rt \in RT : e_{rt} \geq b_{rt} \]  

(5.15)
Ensures that if a reconfiguration task is performed on a component, the reconfiguration can not last less than required:

\[
\forall rt \in RT, \forall c \in C^h : \\
e_{rt} \geq b_{rt} + (bit_{c} - (1 - rtc_{rt,c}) \cdot bit_{max}) \cdot T_{rec}
\] (5.16)

5.1.4 Problem constraints

Here we define the constraints tightly related to the scheduling problem.

Ensures the dependencies among the tasks:

\[
\forall (t1, t2) \in P : b_{t2} \geq e_{t1}
\] (5.17)

Avoids overlap among tasks mapped on the same component (we exploit the fact that if \(bo_{t1,t2} = bo_{t2,t2} = 0\) there is no overlapping among tasks \(t1, t2 \in OT\)):

\[
\forall (t1, t2) \in OT, \forall c \in C \mid (t1, c), (t2, c) \in TC : \\
bo_{t1,t2} + mc_{t1,c} + mc_{t2,c} \leq 2
\] (5.18)

Ensures non overlapping also with respect to \(cp\) (i.e., given a task \(t\), there is at most one previous task and one subsequent task on the same hardware component):

\[
\forall t \in T : \\
\sum_{(t1,t2) \in CP} cp_{t,t2} \leq 1 \quad \sum_{(t2,t) \in CP} cp_{t2,t} \leq 1
\] (5.19)
Avoids overlapping between the potential reconfiguration tasks by enforcing a sequential order (to state this inequality we assume the reconfiguration tasks assigned to unique natural numbers in the interval \([1, |T| - 1]\)):

\[
\forall rt \in RT \mid rt > 1 : b_{rt} \geq e_{rt-1}
\] (5.20)

Ensures that the hardware components do not exceed the resources provided by the FPGA:

\[
\forall r \in R : \sum_{c \in C^h} oc_{c,r} \leq maxRes_r
\] (5.21)

Ensures that between two subsequent tasks \(t_1, t_2 \in T\) mapped on the same hardware component with different implementations a reconfiguration must be performed to configure task \(t_2\):

\[
\forall (t_1, t_2) \in CP, \forall i_1 \in I^h \mid (t_1, i_1) \in TI \land (t_1, t_2) \in CT :
\]

\[
\sum_{rt \in RT} rtt_{rt,t_2} \geq c_{t1,t_2} + mi_{t1,i_1} + \sum_{(t_2,i_2) \in TI, i_2 \in I^h \land i_1 \neq i_2} mi_{t2,i_2} - 2
\] (5.22)

\[
\forall t \in T :
\]

\[
\sum_{rt \in RT} rtt_{rt,t} \geq \sum_{(t_2,t) \in CP \land (t_2,t) \notin CT} cp_{t2,t}
\]
Guarantees that a reconfiguration between tasks \( t_1 \in T \) and \( t_2 \in T \) is executed after \( t_1 \) and before \( t_2 \):

\[
\forall rt \in RT, \forall t \in T : \\
b_t \geq e_{rt} - (1 - rt_{t,t}) \cdot T_{max}
\]

(5.23)

\[
\forall rt \in RT, \forall (t, t_2) \in CP : \\
e_t \leq b_{rt} + (2 - rt_{t,t_2} - cp_{t,t_2}) \cdot T_{max}
\]

Notice that the proposed model currently does not directly consider the delay due to communication among tasks. However, Equation 5.17 can be easily modified to take into account a fixed \( \lambda_{t_1,t_2} \) communication time among tasks \( t_1 \) and \( t_2 \) that is not dependent on the selected implementations:

\[
\forall (t_1, t_2) \in P : b_{t_2} \geq e_{t_1} + \lambda_{t_1,t_2}
\]

(5.24)

### 5.1.5 Cutting planes

In the MILP model we used Equation 5.18 to avoid overlapping among tasks on the same component. This constraint depends on variable \( bo \) that in turns depends on \( bb \) and \( ba \). The coherence of variables \( bb \) and \( ba \) is guaranteed by Equation 5.2. When the \( T_{max} \) parameter is too big, the solver may fail to attribute the correct values to \( bb \) and \( ba \) due to numerical errors, so that overlapping is not correctly detected. To overcome this issue we introduce an
additional non overlapping constraint that binds variables $bb$ and $ba$ directly to variables $mc$ while improving at the same time the description of the model:

\[ \forall (t_1, t_2) \in OT, \forall c \in C \mid (t_1, c) \in TC \land (t_2, c) \in TC : \]
\[ mc_{t_1,c} + mc_{t_2,c} - 1 \leq bb_{t_1,t_2} + bb_{t_2,t_1} \]
\[ mc_{t_1,c} + mc_{t_2,c} - 1 \leq ba_{t_1,t_2} + ba_{t_2,t_1} \]

The formulation can be further improved by removing symmetries that depend on the application domain. If we consider an optimal schedule, another solution with the same quality can be easily obtained by applying a permutation of the indexes of the hardware and the software components. To remove this high amount of equivalent solutions we can simply add constraints that prevent the mapping of some of the tasks to some of the components. If we consider both the hardware and software components and the tasks assigned to increasing number starting from 1, the symmetries can be removed as follows:

\[ \forall c \in C, \forall t \in T \mid t \leq c \land (t, c) \in TC : mc_{t,c} = 0 \]

5.1.6 Objective function

Thanks to the variables and the parameters of the MILP model we are able to compute and optimize the following three different metrics:

Execution time ($T_{cost}$): The overall execution time needed to complete the computation of all the tasks of the schedule including reconfiguration tasks.
Peak power \((P_{cost})\): The estimated peak power reached by the schedule, this value is computed considering the maximum overall power consumption reached within a single time unit;

Energy consumption \((E_{cost})\): The estimated energy consumption for the schedule, it is computed considering the specific implementation selected for each task and the energy needed for all the reconfigurations.

In order to define the metrics within the model, we consider \(T_{cost}, P_{cost}\) and \(E_{cost}\) as new non negative \((\geq 0)\) real variables. Since the objective is to minimize a suitable combination of these metrics, it is enough to provide lower bounds for variables \(T_{cost}, P_{cost}\) and \(E_{cost}\).

The lower bound for the execution time is given by the following constraint:

\[
\forall t \in AT : T_{cost} \geq e_t
\]  

(5.27)

In order to compute lower bounds for the peak power we do not need to consider all the time instants within the schedule, indeed, the peak power varies only when a task begins or ends its execution. Furthermore, when a tasks ends its execution, the peak power cannot increase and this allows us to take into account only the time instants in which a task begin its execution. For the beginning of each task \(t \in AT\) we consider the total power of all the tasks that are in execution at time \(b_t\) (i.e., all tasks \(t2 \in AT\) such that \(bo_{t,t2} = 1\)). The following are the lower bounds for \(P_{cost}\) considering both time instants derived from application tasks and reconfiguration tasks (variable \(rtt\) is used to remove not required reconfigurations):
∀t ∈ T : P_{cost} ≥ \sum_{(t,i) \in TI} m_{t,i} \cdot power_i + pow_{RT_t} + pow_{T_t}

∀rt ∈ RT : P_{cost} ≥ \sum_{t \in T} rtt_{rt,t} \cdot P_{rec} + pow_{T_{rt}}

Where:

\[ pow_{RT_t} = \sum_{rt \in RT, t2 \in T \mid (t,rt) \in OT} rtt_{rt,t2} \cdot P_{rec} \]  

\[ pow_{T_t} = \sum_{t2 \in T, (t2,i) \in TI \mid (t,t2) \in OT} mib_{t2,i} \cdot power_i \]

On the other hand, about the energy consumption, we are able to compute an exact value as:

\[ E_{cost} = \sum_{(t,i) \in TI} m_{t,i} \cdot time_i \cdot power_i + \sum_{rt \in RT} (e_{rt} - b_{rt}) \cdot P_{rec} \]  

(5.30)

Notice that for a reconfiguration rt that is not required, the solver is allowed to set \( b_{rt} = e_{rt} \) so that it does not impact on the energy cost. Overall, a possible objective function for the problem can be obtained with a linear combination of \( T_{cost}, P_{cost} \) and \( E_{cost} \):

\[
\min \left\{ q_1 \cdot \frac{T_{cost}}{T_{max}} + q_2 \cdot \frac{P_{cost}}{P_{max}} + q_3 \cdot \frac{E_{cost}}{E_{max}} \right\}
\]  

(5.31)
Where $T_{\text{max}}$, $P_{\text{max}}$ and $E_{\text{max}}$ are normalization terms representing the maximum value that $T_{\text{cost}}$, $P_{\text{cost}}$ and $E_{\text{cost}}$ can achieve respectively. While $q_1$, $q_2$ and $q_3$ are weights that can be set according to the designer preferences.

5.2 Iterative scheduler

In Section 5.1 we described the MILP formulation used by both the optimal scheduler and the heuristic scheduler. As we stated before, even though the model could be solved directly by a MILP solver to achieve the optimal solution, this approach can only be adopted for taskgraphs with a small number of nodes. Indeed, the size of the model rapidly increases with respect to the number of tasks, so that the resulting formulation becomes challenging to solve. In order to overcome this issue, we consider an approach that schedules a subset of the tasks at each iteration. The idea is to iteratively force the values of some of the integer variables of the MILP model (i.e., $\text{mic}, \text{cp}, \text{cft}, \text{rtt}$) in order to reduce the size of the formulation. At each iteration, the solver computes the optimal solution for the reduced formulation considering the objective
function provided by the designer. The pseudo code of the iterative scheduler is shown in Algorithm 1.

```
function provided by the designer. The pseudo code of the iterative scheduler is shown in Algorithm 1.

Algorithm 1: Iterative scheduling algorithm

5.3 Computation of the normalization terms

In order to correctly compute the objective function, it is needed to accurately estimate the normalization terms $E_{max}, P_{max}, T_{max}$. 
The computation of $E_{\text{max}}$ is straightforward. For each pair task, implementation we add up the energy of those ones that have the maximum energy (in terms of \textit{execution time} \cdot \textit{peak power}), considering also the energy consumed by a reconfiguration in case of hardware implementations.

The computation of $T_{\text{max}}$ is straightforward too. Again, for each pair task, implementation we add up the execution time of those ones that have the maximum execution time, considering also the time spent for the reconfiguration in case of hardware implementations.

Instead, the computation of $P_{\text{max}}$ is more complex. In order to get the best normalization term, we can not simply sum all the maximum power for each task, implementation pair, since this would lead to an overestimated normalization term. So, we decided to exploit the Dilworth’s theorem [36], in order to come out with a more accurate estimation. Through the Dilworth’s theorem we are able to measure the degree of parallelism of a DAG (i.e., the application taskgraph). With degree of parallelism, or width $w$ of the graph, we mean how many tasks can be executed at the same time, at most.

Before we introduce Dilworth’s theorem, we need to introduce the definition of chain:

**Definition 1.** a chain is a set of elements every two of which are comparable.

Now, we can introduce the Dilworth’s theorem:

**Theorem 1.** a partially ordered set (i.e., a DAG) has finite width $w$ if and only if it may be partitioned into $w$ chains.
The width is the degree of parallelism that we need in order to know how many tasks can run in parallel, given the taskgraph. At this point, we can compute $P_{\text{max}}$ adding up the worst $w$ tasks (in terms of power), instead of all of them.

In order to maximize the matching (i.e., set of edges without common nodes) of the bipartite graph of the transitive precedences we used another MILP formulation.

Let us introduce the needed sets:

$$T := \text{set of tasks, the DAG nodes;}$$

$$E := \text{the transitive closure of the DAG direct precedences (i.e., } (t_1, t_2) \in E \text{ if and only if } t_2 \text{ depends on } t_1).$$

The only variable that we need is:

$$edge_{t_1,t_2} := \forall (t_1, t_2) \in E: \text{real variable in the range } [0, 1] \text{ set to 1 if we consider the precedence relation between } t_1 \text{ and } t_2.$$

Now, we define the two constraints related to the problem:

$$\forall t \in T: \sum_{t_2 \in T} edge_{t,t_2} \leq 1 \text{ i.e., for each task there has to be at most one successor;}$$

$$\forall t \in T: \sum_{t_2 \in T} edge_{t_2,t} \leq 1 \text{ i.e., for each task there has to be at most one predecessor.}$$

The objective function is: $\text{max}(edge_{t_1,t_2})$. In this way, forcing the existence of at most one edge for each pair of tasks and maximizing the number of those edges, we get the maximum matching and then we can easily compute the chains and counting them we get the width.

For clarifying how this works, let us make an example. Given the following taskgraph:
We compute the graph of transitive precedences, obtaining the following:
Figure 8.: Bipartite graph of transitive precedences

Now, using the previous MILP formulation we maximize the matching of the bipartite graph of the transitive precedences.
Given the maximum matching, we can compute the chains of the DAG.

Figure 9.: Chosen precedences
Finally, we just need to count the retrieved chains in order to know the width of the graph.

5.4 Computation of the number of hardware regions

The MILP formulation grows not only with the number of tasks, but also with the number of hardware regions where these tasks can be scheduled. So, it is necessary to give to the MILP model only those hardware regions that can be actually used from the given tasks, taking into account the limitation of resources of the programmable logic (i.e., our FPGA).
This problem is a variant of the bin packing problem [37], considering multiple resources. Indeed, for each region (i.e., bin) we have to consider how many BRAM, CLB and DSP (i.e., resources) can fit with respect to the total number of available resources on the FPGA.

In order to solve the previous problem we used another MILP model that we are going to describe below. First of all we need the following parameters and sets:

\[ T := \text{set of tasks}; \]
\[ TI := \text{set of viable task, implementation pairs}; \]
\[ totBRAM := \text{total number of available BRAM on the FPGA}; \]
\[ totCLB := \text{total number of available CLB on the FPGA}; \]
\[ totDSP := \text{total number of available DSP on the FPGA}; \]
\[ BRAM_i := \text{number of required BRAM for the implementation } i; \]
\[ CLB_i := \text{number of required CLB for the implementation } i; \]
\[ DSP_i := \text{number of required DSP for the implementation } i. \]

The only variable we need to use is:

\[ z_{t,i} := \forall (t, i) \in TI: \text{binary variable set to 1 if we choose to put into a region the implementation } i \text{ of task } t. \]

The constraints are:

\[ \forall t \in T: \sum_{(t,i) \in TI} z_{t,i} \leq 1 \text{ i.e., for each task at most one implementation can be chosen;} \]
\[ \sum_{(t,i) \in TI} z_{t,i} \cdot BRAM_i \leq totBRAM \text{ i.e., the sum of the occupied BRAM of each chosen task, implementation pair can not exceed the total available BRAM on the FPGA;} \]

\[ \sum_{(t,i) \in TI} z_{t,i} \cdot CLB_i \leq totCLB \text{ i.e., the sum of the occupied CLB of each chosen task, implementation pair can not exceed the total available CLB on the FPGA;} \]

\[ \sum_{(t,i) \in TI} z_{t,i} \cdot DSP_i \leq totDSP \text{ i.e., the sum of the occupied DSP of each chosen task, implementation pair can not exceed the total available DSP on the FPGA.} \]

In the end, the objective function is simply: \[ max(z_{t,i}) \]. Maximizing the number of hardware regions occupied by an implementation of a task, gives us the maximum number of hardware regions that can be used at most by the given taskgraph.
CHAPTER 6

EXPERIMENTAL EVALUATION

In this chapter we show in Section 6.2 the experimental results of our MILP-based scheduler and the ACO scheduler on a set of synthetic generated benchmark that are described in Section 6.1. We also discuss the floorplannability of the returned schedules in Section 6.4

6.1 Synthetic benchmark generation

In order to generate the set of benchmark for the following tests we used the Task Graph For Free (TGFF) [38] framework. This graph generator has already been used in the literature [30, 32] because it can be easily controlled to generate pseudo-random task graphs with specific properties. For instance, for each task we forced the presence, on average, of four different implementations ensuring common hardware implementations among tasks in order to exploit module reuse. The power, execution time and needed resources for each implementation, the speed and power for the reconfiguration controller and the total available resources on the FPGA have been set to realistic random values. Furthermore, we generated the benchmark in a way that the bigger is the bitstream size of a task, the smaller is its execution time, since usually more resources imply a faster execution.

6.2 Iterative scheduler and ACO scheduler comparison

In this section we present an evaluation of our iterative scheduler algorithm with respect to the ACO scheduler presented in Chapter 4. The iterative algorithm has been implemented in
python using Gurobi 6.0 [35] for the optimization of the MILP model, while the experiments were performed on an Intel Core i7-2630QM under Linux.

As a first test suite we considered 36 taskgraphs having different numbers of nodes in the range \([10,50]\) with a step of 5 nodes at the time. As we stated previously, for each task we generated, on average, four different hardware/software implementations and we ensured the presence of common hardware implementations to allow the exploitation of module reuse.

In order to evaluate our approach we compared the results with the ACO based scheduler. For each problem instance we executed the iterative scheduler (IS-k) varying the number of considered tasks per iteration \((k\) from 1 to 5).

We will compare our iterative scheduler with the ACO one with respect to all the three metrics we defined (i.e., schedule makespan, peak power and energy consumption) in, respectively, Subsection 6.2.1, Subsection 6.2.2 and Subsection 6.2.3.

In the second test suite, for verifying the floorplannability of the schedules, we included also the 5 nodes taskgraphs. The results are shown in Section 6.4.

6.2.1 Schedule makespan

For the first comparison we used the schedule makespan as the metric of choice, Table I. reports the quality of the solutions achieved by IS-k and the ACO scheduler, while the execution time of the two algorithms is reported in Table II.
<table>
<thead>
<tr>
<th># Tasks</th>
<th>Average schedule makespan [clock cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACO</td>
</tr>
<tr>
<td>10</td>
<td>5364</td>
</tr>
<tr>
<td>15</td>
<td>11765</td>
</tr>
<tr>
<td>20</td>
<td>12665</td>
</tr>
<tr>
<td>25</td>
<td>29489</td>
</tr>
<tr>
<td>30</td>
<td>22721</td>
</tr>
<tr>
<td>35</td>
<td>22877</td>
</tr>
<tr>
<td>40</td>
<td>62160</td>
</tr>
<tr>
<td>45</td>
<td>37505</td>
</tr>
<tr>
<td>50</td>
<td>39845</td>
</tr>
</tbody>
</table>
TABLE II.: ALGORITHM RUNTIME W.R.T. MAKESPAN

<table>
<thead>
<tr>
<th># Tasks</th>
<th>Average runtime [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACO</td>
</tr>
<tr>
<td>10</td>
<td>9.23</td>
</tr>
<tr>
<td>15</td>
<td>18.57</td>
</tr>
<tr>
<td>20</td>
<td>28.36</td>
</tr>
<tr>
<td>25</td>
<td>35.95</td>
</tr>
<tr>
<td>30</td>
<td>52.14</td>
</tr>
<tr>
<td>35</td>
<td>69.01</td>
</tr>
<tr>
<td>40</td>
<td>76.08</td>
</tr>
<tr>
<td>45</td>
<td>95.96</td>
</tr>
<tr>
<td>50</td>
<td>119.4</td>
</tr>
</tbody>
</table>

On average, all the IS-k executions of the makespan benchmark produced better solutions with respect to the ones obtained by the ACO scheduler, within Table III. the average improvement over the ACO-based algorithm are reported. It is worth noting that IS-4 gives an average reduction of the makespan of 23.7% with an execution time that is comparable to the ACO scheduler. On the other hand, even though IS-5 leads to higher quality solutions, the
time needed to solve the MILP model is in general significantly higher due to the increased size of the formulation. In Table IV. is shown the average improvement and worsening of the execution time of the IS-k algorithm with respect to ACO.

**TABLE III.: AVERAGE SCHEDULE MAKESPAN IMPROVEMENT**

<table>
<thead>
<tr>
<th></th>
<th>IS-1</th>
<th>IS-2</th>
<th>IS-3</th>
<th>IS-4</th>
<th>IS-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>16.0%</td>
<td>19.0%</td>
<td>21.4%</td>
<td>23.7%</td>
<td>27.7%</td>
</tr>
</tbody>
</table>

**TABLE IV.: AVERAGE ALGORITHM RUNTIME W.R.T. MAKESPAN**

<table>
<thead>
<tr>
<th></th>
<th>IS-1</th>
<th>IS-2</th>
<th>IS-3</th>
<th>IS-4</th>
<th>IS-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>48.2%</td>
<td>59.6%</td>
<td>59.2%</td>
<td>-7.9%</td>
<td>-2253.1%</td>
</tr>
</tbody>
</table>
As we can see, for some taskgraphs, the optimal solution scheduling 5 nodes at the time has been extremely computational expensive, and that led to the -2253.1% worsening for the algorithm runtime.

### 6.2.2 Schedule peak power

In Table V. is reported the comparison of the schedule peak power returned by IS-k and ACO, while the execution time of the two algorithms during the test is reported in Table VI.

**TABLE V.: SCHEDULE PEAK POWER COMPARISON**

<table>
<thead>
<tr>
<th># Tasks</th>
<th>Average schedule peak power [mW]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACO</td>
</tr>
<tr>
<td>10</td>
<td>5646</td>
</tr>
<tr>
<td>15</td>
<td>5181</td>
</tr>
<tr>
<td>20</td>
<td>5089</td>
</tr>
<tr>
<td>25</td>
<td>4787</td>
</tr>
<tr>
<td>30</td>
<td>5822</td>
</tr>
<tr>
<td>35</td>
<td>6107</td>
</tr>
<tr>
<td>40</td>
<td>6468</td>
</tr>
<tr>
<td>45</td>
<td>7401</td>
</tr>
<tr>
<td>50</td>
<td>6699</td>
</tr>
</tbody>
</table>
The peak power benchmark results shown in Table VII. are more restrained: an average improvement up to 10.3% can be observed in the IS-3 case, while in the others the improvement is around 5.5%. This is due to the fact that, once a set of implementations for $k$ tasks is chosen, the peak power for those tasks can not be changed, hence the only option left to the solver is to place the next $k$ tasks in such a way that they do not overlap with the others respecting

<table>
<thead>
<tr>
<th># Tasks</th>
<th>ACO [s]</th>
<th>IS-1 [s]</th>
<th>IS-2 [s]</th>
<th>IS-3 [s]</th>
<th>IS-4 [s]</th>
<th>IS-5 [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8.66</td>
<td>1.26</td>
<td>0.72</td>
<td>0.84</td>
<td>1.0</td>
<td>0.95</td>
</tr>
<tr>
<td>15</td>
<td>18.9</td>
<td>3.17</td>
<td>2.4</td>
<td>2.38</td>
<td>3.57</td>
<td>3.95</td>
</tr>
<tr>
<td>20</td>
<td>29.48</td>
<td>6.53</td>
<td>4.07</td>
<td>4.26</td>
<td>7.49</td>
<td>9.32</td>
</tr>
<tr>
<td>25</td>
<td>35.32</td>
<td>10.82</td>
<td>6.88</td>
<td>7.62</td>
<td>9.24</td>
<td>17.21</td>
</tr>
<tr>
<td>30</td>
<td>56.03</td>
<td>32.92</td>
<td>22.04</td>
<td>25.19</td>
<td>37.45</td>
<td>144.67</td>
</tr>
<tr>
<td>35</td>
<td>68.65</td>
<td>38.84</td>
<td>23.51</td>
<td>23.49</td>
<td>49.72</td>
<td>47.54</td>
</tr>
<tr>
<td>40</td>
<td>76.82</td>
<td>46.32</td>
<td>23.57</td>
<td>24.6</td>
<td>34.29</td>
<td>41.72</td>
</tr>
<tr>
<td>45</td>
<td>92.1</td>
<td>71.85</td>
<td>41.4</td>
<td>47.02</td>
<td>61.35</td>
<td>62.46</td>
</tr>
<tr>
<td>50</td>
<td>116.71</td>
<td>147.02</td>
<td>81.02</td>
<td>88.47</td>
<td>160.26</td>
<td>192.42</td>
</tr>
</tbody>
</table>
Nevertheless, as it is shown in Table VIII., the execution time of our iterative scheduler is, on average, always smaller than the one of the ACO-based algorithm.

### TABLE VII.: AVERAGE SCHEDULE PEAK POWER IMPROVEMENT

<table>
<thead>
<tr>
<th></th>
<th>IS-1</th>
<th>IS-2</th>
<th>IS-3</th>
<th>IS-4</th>
<th>IS-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3%</td>
<td>7.3%</td>
<td>10.3%</td>
<td>5%</td>
<td>4.3%</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE VIII.: AVERAGE ALGORITHM RUNTIME W.R.T. PEAK POWER

<table>
<thead>
<tr>
<th></th>
<th>IS-1</th>
<th>IS-2</th>
<th>IS-3</th>
<th>IS-4</th>
<th>IS-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>39.8%</td>
<td>56.9%</td>
<td>54.7%</td>
<td>37.6%</td>
<td>13.9%</td>
<td></td>
</tr>
</tbody>
</table>
6.2.3 Schedule energy consumption

The comparison of the schedule energy consumption between IS-k and ACO is reported in Table IX., while the execution time of the two algorithms during the benchmark is reported in Table X..

<table>
<thead>
<tr>
<th># Tasks</th>
<th>Average schedule energy consumption [J]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACO</td>
</tr>
<tr>
<td>20</td>
<td>40.43</td>
</tr>
<tr>
<td>25</td>
<td>83.15</td>
</tr>
<tr>
<td>30</td>
<td>64.08</td>
</tr>
<tr>
<td>35</td>
<td>66.42</td>
</tr>
<tr>
<td>40</td>
<td>226.62</td>
</tr>
<tr>
<td>45</td>
<td>249.24</td>
</tr>
<tr>
<td>50</td>
<td>121.13</td>
</tr>
</tbody>
</table>
As we can see in Table XI., the energy consumption benchmark shows a steady improvement of about 38%. This is due to the fact that energy optimization diverts the problem towards a mapping problem rather than a scheduling problem, so the number of scheduled tasks at the same time (i.e. $k$) affects the results only slightly. From Table XI. we can see that the execution...
time of the iterative scheduler improves up to 51% with respect to the ACO-based algorithm in the IS-3 case, while it worsens for bigger values of $k$.

<table>
<thead>
<tr>
<th>TABLE XI.: AVERAGE SCHEDULE ENERGY IMPROVEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS-1 IS-2 IS-3 IS-4 IS-5</td>
</tr>
<tr>
<td>38.2% 38.6% 37.8% 38.1% 38.3%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE XII.: AVERAGE ALGORITHM RUNTIME W.R.T. ENERGY</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS-1 IS-2 IS-3 IS-4 IS-5</td>
</tr>
<tr>
<td>35.9% 49.8% 51% -14% -135.2%</td>
</tr>
</tbody>
</table>

6.3 Pareto curve of the multiobjective iterative scheduler

As a second test case, we considered a problem instance with 20 tasks taken from the previous benchmark to analyze the impact of a multiobjective optimization driven both by
energy and time metrics. Specifically, we performed 9 different executions of IS-1 setting the objective functions weights $q^i_1, q^i_2, q^i_3$ for each iteration $i$ as follows:

$$
\forall i \in \{0.1, 0.2, \cdots, 0.8, 0.9\} :
$$

$$
q^i_1 = i, \quad q^i_2 = 0, \quad q^i_3 = 1 - i
$$

(6.1)

Less than a minute was needed to obtain the solutions for all the executions. With a post processing step we removed those results that were dominated by other in terms of both the makespan and the overall energy consumption of the schedule. Figure 11. shows a pareto analysis considering the non-dominated solutions. A similar approach could also be followed by the application designer in order to quickly evaluate several promising solution on conflicting metrics. Moreover, even though we considered $E_{cost}$, $P_{cost}$ and $T_{cost}$ only within the objective function, these variables can be easily used in additional user constraints, to prune solutions that do not meet the application requirements.
As we underlined in Chapter 3 Section 3.3, even though we take into account the total amount of resources (i.e., BRAMs, CLBs, DSPs) available on the FPGA, some of the schedules that we return could be infeasible from the floorplanner point of view because of geometrical

6.4 **Floorplan of the schedule**

Figure 11.: Pareto analysis on energy consumption and execution time of a schedule
issues about the placing of the hardware regions on the programmable logic. So, in this section, we report the results of the tests we made on the floorplan feasibility. In Subsection 6.4.1 we describe the procedure to follow in order to get a floorplannable schedule in an iterative way, while in Subsection 6.4.2 we describe the phenomenon that forces us to use that procedure for getting a schedule that can be implemented on the FPGA. For these tests we used a state-of-the-art floorplanner [27].

6.4.1 Floorplannable schedule

In most cases forcing the scheduler to consider only a portion of the FPGA resources, instead of the whole programmable logic, makes the returned schedule more likely to be floorplannable. This happens because, using less resources than the actual ones available, allows the floorplanner to have more operating space for placing the hardware regions on the FPGA.

So, we can get a floorplannable schedule following this method: we run the scheduler for the first time with 100% of the FPGA resources, then we run the floorplanner on the returned schedule and verify whether the schedule is feasible or not. If it is feasible, then we can stop, since, as we see in Figure 12., the trend goes to better schedules when the amount of given resources is higher (notice that the number of points between the two curves in the figure does not match because in some cases the same solution is returned by the scheduler, hence there are overlying points). If it is not feasible, we can decrease the amount of FPGA resources by 10%, 5%, 1%, or less for more precision, that we give to the scheduler and repeat the process. We used a 5% decrement from 100% to 50% in order to cover as many different cases as possible with respect to the FPGA that we were using (i.e., Xilinx XC7Z020 [9]). Indeed, with 14
BRAM tiles, 133 CLB’s tiles and 11 DSP tiles, decreasing them by 5% we cover all the cases from 14 to 7 tiles in the BRAM case, from 11 to 5 tiles in the DSP case and a decent different amount from 133 to 66 tiles in the CLB case. Furthermore, from 50% down to 0% of FPGA given resources we observed that all the schedules were floorplannable.

![Figure 12.: Makespan VS given and used resources of a 5 nodes schedule](image-url)
This approach is simple but effective, indeed, as we can see in Figure 13., some applications are already floorplannable with 100% of given resources and many of the 40 test cases become feasible when we give the scheduler 85% of the FPGA resources. While, in the worst case, we have to reach the 70% of given resources to get a floorplannable schedule. On average, instead, most of the schedules are floorplannable when 87% of the resources are given to the scheduler. These tests have been made in the same way the system designer is supposed to get the best feasible schedule for his application. So, with a decrement of 5% of the FPGA resources we can reach a good floorplannable schedule with 4 iterations of the algorithm (on average), or 7 iterations in the worst case.
The reason we can not use a clever approach (like, for instance, binary search on the interval of the FPGA resources given to the scheduler) is a phenomenon that we call feasible/infeasible noise, which is described in Subsection 6.4.2. Notice that we performed these tests preprocessing
the amount of FPGA resources of the application implementations in order to reduce the feasible/infeasible noise. Even this preprocessing step is described in Subsection 6.4.2.

6.4.2 Feasible/Infeasible noise

As we can see in Figure 14., the feasibility of a schedule (with respect to the floorplanning phase), given different amount of FPGA resources, is not a monotonic function in general. Indeed, we expect that under a certain threshold the schedule will be always floorplannable, but this property is not always guaranteed (e.g., in Figure 14. for 80% and 50% of given resources the schedule is not feasible even though it is for 90%).

![Figure 14.: Feasible/Infeasible noise](image)

The main reason for this phenomenon resides in the fact that we are performing the scheduling and floorplanning in two distinct phases instead of doing everything at the same time. In-
deed, the scheduler does not have the information about the real amount of resources that the chosen implementations occupy on the FPGA region; such information is known only to the floorplanner. The reason behind the difference, in terms of FPGA resources, among implementations and hardware regions is geometrical and resides in the fact that the floorplanner has to compute rectangular regions.

Let us clarify this phenomenon with an example. We have an application (and then a schedule) with two tasks $t_1$ and $t_2$. Let us suppose that we have an FPGA with 4 BRAMs, 4 DSPs and 8 CLBs (as in Figure 15.).

![Figure 15.: Given FPGA](image_url)
Given the 100% of the FPGA resources, the scheduler chooses for $t_1$ and implementation that requires 4 CLBs and 4 DSPs, while for $t_2$ an implementation that requires 4 BRAMs and 4 CLBs. Given the regular geometry of the FPGA, the floorplanner returns a floorplan as the one reported in Figure 16.

Figure 16.: Floorplan with 100% of the FPGA resources
Now, if we give to the scheduler 75% of the FPGA resources (i.e., 3 BRAMs, 6 CLBs and 3 DSPs), the scheduler can not choose the previous implementations and then goes for the second best choice it has, for instance, an implementation requiring 1 BRAM, 3 CLBs and 3 DSPs for $t_1$ and another one requiring 2 BRAMs and 3 CLBs for $t_2$. From the scheduler point of view these two implementations are within the given resources, but from the floorplanner point of view the schedule is not feasible, as we can see in Figure 17.

![Figure 17.: Floorplan with 75% of the FPGA resources](image-url)
Indeed, the real occupied resources of the implementation chosen by the scheduler for $t_1$ are 3 BRAMs, 6 CLBs and 3 DSPs (and not the required 1 BRAM, 3 CLBs and 3 DSPs) for geometrical reasons of the floorplanner (i.e., the regions have to be rectangular) that are unknown during the scheduling phase. So, there is no room left for the implementation of $t_2$ that requires 2 BRAMs and 3 CLBs, then even if the schedule is legit from the scheduler point of view, it is not from the floorplanner stand point.

As we saw in the previous example, diminishing the FPGA resources given to the scheduler (in order to leave more space to the floorplanner for positioning the scheduled regions) does not always ensure a floorplannable schedule.

We can mitigate this feasible/infeasible noise, by precomputing a table where each entry is the amount of BRAMs, CLBs and DSPs for each possible rectangular region on the FPGA. Then, before scheduling the application, we modify the required amounts of BRAMs, CLBs and DSPs of each implementation taking the entries of the previous table that satisfy all the requirements (in terms of resources) and then taking the minimum of each resources of those entries. Notice that, since we are taking the minimum of different entries, the new required amount of resources of the implementation may not be a real region on the FPGA (while the entries are), so we can still have feasible/infeasible noise.

Going forward in trying to eliminate this issue would mean to choose an entry (i.e., a region) from the table of all the possible regions for each implementation, making a choice that the floorplanner is supposed to do, and then overestimating the amount of resources needed by each implementation. This would lead to poor solutions.
Reducing the feasible/infeasible noise also leads to get a floorplannable schedule earlier. Indeed, as we can see comparing Figure 18. and Figure 13., if we do not perform this preprocessing step we find a feasible schedule at 82% of the given resources on average, while it was 87% for the modified taskgraphs, and we have to go till 55% in the worst case scenario, rather than 70%.

Figure 18.: First feasibility percentage of given resources without noise reduction
CHAPTER 7

CONCLUSIONS

In this chapter we show our contribution to the scheduling problem with the heuristic iterative scheduler and the optimal MILP-based scheduler that we presented in Chapter 5. In Section 7.1 we summarize the features of the two algorithms that we proposed highlining their limits, while in Section 7.2 we introduce new features and improvements that could be exploited to better this work.

7.1 Contributions and limitations

The heuristic iterative algorithm and the optimal MILP-based one solve the scheduling problem for reconfigurable devices such as heterogeneous MPSoCs. Both of them exploit the same MILP formulation, indeed the optimal algorithm is a particular case of the heuristic one when the number of task to consider at the same time $k$ is equal to the total number of tasks of the application.

The best way to exploit our approaches would be to use the iterative algorithm with $k$ equal to the one that performs better in Chapter 6, taking into account which metrics has to be optimize. Then, if the result is not satisfying, different values of $k$ can be tested or the returned solution can be given to the optimal scheduler and let it optimizes that solution until is acceptable.

Our main contributions are listed below:
• we have introduced a MILP formulation to address the scheduling problem that can be used both with a heuristic iterative algorithm and an optimal algorithm;

• we avoided the exploration of solutions that exceed the total amount of available resources on the FPGA;

• we introduced an objective function with respect to the schedule is optimized that takes into account three different metrics (execution time, peak power and energy consumption of the schedule) as a linear combination.

However, this work can still be improved in many ways. First of all, we do not take into considerations the time, power and energy required by the data transfer among tasks. Indeed, the communication is only addressed as data dependencies (i.e., precedences) among tasks. Moreover, the formulation of the MILP model still has numerical issues if the execution time of the tasks is too big and it could be further improved with other cuts to the solution space.

7.2 Future work

An interesting future work would be to include the different kinds of communication that are allowed among tasks on heterogeneous architectures. For instance, there could be data communication among software tasks through cache and RAM memory, among hardware tasks through a bus or BRAM (using protocols like AXI4-Stream and AXI4 [39]) and, finally, among hardware and software tasks through RAM (using Direct Memory Access (DMA)).

Another extension could be the integration of the floorplanning problem in order to exclude by construction also those solution that cannot be placed on the FPGA. However, this would lead to a even more complex MILP formulation, so it would be worth it to consider all the
advantages and disadvantages of such extension with respect to our approach that keeps the scheduling and floorplanning problems separated.

In the end, using a different approach to the problem, it could be possible to make the MILP model smaller in terms of variables and constraints. Furthermore, additional cuts can be found to resize the solution space. Both improvements would lead to the possibility of dealing with bigger taskgraphs in less time.
CITED LITERATURE


VITA

Enrico Armenio Deiana

Education

B.S., Engineering of Computing Systems
Politecnico di Milano
September 2011

M.S., Engineering of Computing Systems
Politecnico di Milano
July 2015

M.S., Computer Science (current)
University of Illinois at Chicago, Chicago, IL
August 2015