RUNTIME CORE ALLOCATION MANAGEMENT FOR 2D SELF-PARTIALLY AND DYNAMICALLY RECONFIGURABLE SYSTEMS

BY
MASSIMO MORANDI
Bachelor in Computer Engineering, Politecnico di Milano, 2006

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

Chicago, Illinois
Victorious warriors win first and then go to war, while defeated warriors go to war first and then seek to win.

Sun Tzu
To my Family and Friends

Massimo
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Setting</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Creation and Management of Self-Reconfigurable Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Core Allocation Management</td>
<td>7</td>
</tr>
<tr>
<td>2 PROBLEM DEFINITION</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Two Dimensional Reconfiguration</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Problems with 2D Reconfiguration</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Fragmentation</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2 Core Allocation Manager Choices</td>
<td>14</td>
</tr>
<tr>
<td>2.2.3 Communication Infrastructure</td>
<td>15</td>
</tr>
<tr>
<td>2.2.4 Configuration Bitstreams Generation</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Desired Features for the Core Allocation Manager</td>
<td>20</td>
</tr>
<tr>
<td>3 RELATED WORKS</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Definitions and Metrics</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Fitting Strategies</td>
<td>26</td>
</tr>
<tr>
<td>3.2.1 General Strategies</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2 Focused Strategies</td>
<td>27</td>
</tr>
<tr>
<td>3.3 KAMER method</td>
<td>28</td>
</tr>
<tr>
<td>3.4 KNER method</td>
<td>30</td>
</tr>
<tr>
<td>3.5 CUR method</td>
<td>32</td>
</tr>
<tr>
<td>3.6 2D-Hashing method</td>
<td>34</td>
</tr>
<tr>
<td>3.7 Global Comparison</td>
<td>36</td>
</tr>
<tr>
<td>4 PROPOSED APPROACH</td>
<td>39</td>
</tr>
<tr>
<td>4.1 The Global Workflow</td>
<td>39</td>
</tr>
<tr>
<td>4.1.1 Input Management and Validation Phase</td>
<td>42</td>
</tr>
<tr>
<td>4.1.2 Solution Identification Phase</td>
<td>43</td>
</tr>
<tr>
<td>4.1.3 System Generation Phase</td>
<td>44</td>
</tr>
<tr>
<td>4.1.4 Bitstreams Generation Phase</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Area Constraints Definition</td>
<td>46</td>
</tr>
<tr>
<td>4.3 Core Allocation Management</td>
<td>47</td>
</tr>
<tr>
<td>4.3.1 Proposed Solution for Core Allocation Management</td>
<td>50</td>
</tr>
<tr>
<td>5 IMPLEMENTATION</td>
<td>52</td>
</tr>
<tr>
<td>5.1 Area Constraints Definition</td>
<td>52</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1 User Preferences Management</td>
<td>52</td>
</tr>
<tr>
<td>5.1.2 Constraints Automatic Generation and Alteration</td>
<td>53</td>
</tr>
<tr>
<td>5.2 Core Allocation Management</td>
<td>55</td>
</tr>
<tr>
<td>5.2.1 Supported Scenarios</td>
<td>56</td>
</tr>
<tr>
<td>5.2.2 Cores Representation</td>
<td>57</td>
</tr>
<tr>
<td>5.2.3 Reconfigurable Device State Representation</td>
<td>58</td>
</tr>
<tr>
<td>5.2.4 Rectangles, Split operation, Merge operation</td>
<td>59</td>
</tr>
<tr>
<td>5.2.5 The Algorithms</td>
<td>60</td>
</tr>
<tr>
<td>6 EXPERIMENTAL RESULTS</td>
<td>67</td>
</tr>
<tr>
<td>6.1 Comparison with similar literature solutions</td>
<td>68</td>
</tr>
<tr>
<td>6.2 Completion times when varying device size</td>
<td>70</td>
</tr>
<tr>
<td>6.3 Comparison between multiple-shapes and single-shape</td>
<td>74</td>
</tr>
<tr>
<td>7 CONCLUSIONS</td>
<td>80</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>82</td>
</tr>
<tr>
<td>VITA</td>
<td>89</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>COMPARISON OF PLACEMENT METHODS</td>
<td>37</td>
</tr>
<tr>
<td>II</td>
<td>COMPARISON WITH SIMILAR LITERATURE SOLUTIONS</td>
<td>69</td>
</tr>
<tr>
<td>III</td>
<td>COMPLETION TIMES WHEN VARYING DEVICE SIZE</td>
<td>72</td>
</tr>
<tr>
<td>IV</td>
<td>SINGLE SHAPE VS MULTIPLE SHAPES USING FIRST FIT</td>
<td>75</td>
</tr>
<tr>
<td>V</td>
<td>SINGLE SHAPE VS MULTIPLE SHAPES USING BEST FIT</td>
<td>76</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview of the target self reconfigurable architecture</td>
</tr>
<tr>
<td>2</td>
<td>Optimization due to 2D Reconfiguration</td>
</tr>
<tr>
<td>3</td>
<td>Fragmentation in 2D Reconfiguration</td>
</tr>
<tr>
<td>4</td>
<td>Placement Decisions</td>
</tr>
<tr>
<td>5</td>
<td>Problems with Simple Static Bus</td>
</tr>
<tr>
<td>6</td>
<td>Example of maximally empty rectangles when placing two cores</td>
</tr>
<tr>
<td>7</td>
<td>All Maximally Empty Rectangles vs only Non-Overlapping</td>
</tr>
<tr>
<td>8</td>
<td>Finding a feasible location with CUR</td>
</tr>
<tr>
<td>9</td>
<td>Overview of the target self reconfigurable architecture</td>
</tr>
<tr>
<td>10</td>
<td>Overview of the complete workflow</td>
</tr>
<tr>
<td>11</td>
<td>General Data Flow Chart for the Proposed Solution</td>
</tr>
<tr>
<td>12</td>
<td>Improvement in CRR with FF and BF</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Empty Space Manager for Blind Schedule Scenario .......................... 64
2. Empty Space Manager for Dynamic Schedule Scenario ..................... 65
3. Procedure findPlacement(Core C) ............................................. 66
SUMMARY

A reconfigurable system has the capability of changing its configuration according to the user needs. In the FPGA case, the configuration is represented by a set of functionalities implemented on the reconfigurable device; those functionalities can be configured, i.e. their implementation is allocated on the reconfigurable fabric, and deconfigured, i.e. deallocated upon request. 2D Self partial and dynamical reconfiguration is a powerful approach to reconfiguration, providing flexibility and possibility for high performances, but has the drawback of high complexity in reconfigurable system management when compared to simpler approaches.

The main concerns of this thesis work are the definition of suitable area constraints for Cores that will be configured on the reconfigurable device and the creation of an efficient allocation manager to perform placement choices at runtime. The first aspect includes the choice of a shaping policy, to define reasonable area constraints for Cores. The second aspect includes the management of empty space inside the reconfigurable fabric, fundamental to further allow Core placement decisions and the exploitation of different placement policies (criteria used to choose where to place a Core among many acceptable solutions), to achieve high efficiency; those policies can be either general, such as First Fit or Best Fit, for example, or focused, such as Routing Aware of Fragmentation Aware. The aim is to obtain good results according to several metrics, which include: Core Rejection Rate, Fragmentation, Throughput, Management Overhead and Routing Efficiency. Those metrics are often in contrast one with the other, and it is important to find a good compromise between them.
CHAPTER 1

INTRODUCTION

One of the most important problems in current computing is the continuous search for the best compromise between flexibility and performance: generally every step in the direction of versatility tends to be a step back in terms of efficiency and vice-versa. A possible solution to this problem is offered by Field Programmable Gate Arrays. FPGAs allow the creation of reconfigurable systems (2), which can be considered a good compromise between special purpose hardware - like Application Specific Integrated Circuits (ASICs) - and general purpose computers. A reconfigurable system, as the name implies, has the capability of changing its configuration according to the user needs. In the FPGA case, the configuration is represented by a set of functionalities implemented on the reconfigurable device; those functionalities can be configured, i.e. their implementation is allocated on the reconfigurable fabric, deconfigured, i.e. deallocated, or, by combination of the previous two operations, moved upon request.

Reconfigurable systems, while providing new possibilities for hardware/software co-design (1; 43), also introduce a series of new problems related to their implementation and management. This issue is of particular relevance for systems that exploit partial and dynamical reconfiguration (29; 30; 31; 32). In partial reconfiguration, the functionality of single portions of the reconfigurable device can be modified, while the remaining parts of the device remain unchanged. Furthermore, with dynamic reconfiguration, the device portions that are not di-
rectly involved in the reconfiguration can continue their computation without any interruption due to the reconfiguration process.

The common approach that is adopted to exploit those features is the definition of predetermined area portions on the device, reconfigurable slots, in which components implementing different functionalities of the specification, can be configured independently from the rest of the device. Such a scenario involves partitioning the specification, i.e. subdividing it into partitions each identifying a functionality. We will refer to Core as the implementation of a functionality, independent of the position and shape that the functionality assumes. Moreover, we will call RFU (Reconfigurable Functional Unit) the implementation of a functionality after its shape has been fixed. A configuration bitstream is a binary file that describes an RFU implemented in a specific position and is used to configure it on the reconfigurable device. To obtain partial reconfiguration it is necessary to provide a separate configuration file, called partial bitstream or partial reconfiguration bitstream, for each functionality desired on the device and for each location into which the functionality might be mapped (3; 4).

1.1 **Problem Setting**

The setting for this work is self, partial and dynamical reconfiguration, in both its mono-dimensional and bi-dimensional aspects. Self reconfigurable systems are completely independent in their management, thus they have the need to internally host reconfiguration management functionalities, such as core allocation, and to store or be able to autonomously obtain configuration bitstreams when needed. The 1D reconfiguration paradigm only allows dynamical reconfiguration of columns spanning the whole device vertically, while the 2D paradigm gives
the possibility of configuring fractions of the FPGA of arbitrary rectangular shape; the 2D approach is more powerful because of added flexibility but it is also more complex to be managed.

Self, partial and dynamical reconfiguration is a powerful approach to reconfiguration; it gives flexibility and possibility for high performances but has the drawback of increased complexity in its management when compared to simpler approaches. This increase in complexity becomes even more relevant when exploiting the 2D reconfiguration paradigm.

With a self partial and dynamical approach to reconfiguration it is possible to obtain higher flexibility and better performances when compared to simpler approaches; however, the price for this improvement lies in the increased difficulties in reconfigurable system management which becomes significantly more complex. This problem becomes even more relevant when also exploiting the 2D paradigm, which provides even higher flexibility and yields an even more complex handling due to one more degree of freedom, represented by the additional dimension. An automated or semi-automated way to support this kind of systems would help the designers by raising the level of abstraction at which they must work in architecture definition.

1.2 Creation and Management of Self-Reconfigurable Systems

This work is, in fact, part of a larger project, which aims to create a complete workflow to help the designer in the creation and management of self partially and dynamically reconfigurable systems; the provided support is related to the definition of area constraints for Cores, the creation of an efficient runtime core allocation manager and finally the generation of a solution to obtain internal and fast relocation of cores. Extensive simulation of the scenarios
created by different choices is also included in the flow, as a way to obtain feedback on their
goodness.

The scope of this thesis covers the online Core Allocation Management part of the problem,
which includes the creation of a solution to manage the empty space on the reconfigurable device
and the definition of policies to choose where to place cores among many possible locations; part
of the work is also devoted to the definition of initial area constraints for Cores and criteria to
subsequently improve them, exploiting the simulation feedback on the behaviour of the system
with different shapes.

The goal of the global workflow is to provide automated support for the creation and
the online management processes of self partially and dynamically reconfigurable systems; in
particular, this involves computing at runtime, if necessary, new core placements and allowing
their relocation by means of an hardware filter. We consider the input application as composed
of a set of functionalities that have to be implemented on the target reconfigurable device. In
such a context we will refer to the implementation of each functionality as a Core; each core
is described by one or more partial reconfiguration bitstream, depending on whether multiple
shapes are allowed or not.

With the global workflow we propose a novel approach to automate the creation and manage-
ment of self, partially and dynamically reconfigurable systems. The main points of innovation
for the proposed workflow are:

- Automation: the flow is designed to allow automatic creation of a reconfigurable archi-
tecture, from specification to implementation, with minimal effort from the designer
• Flexibility: the user can vary the level of automation of the system, providing external support whenever desired

• Integration: the flow includes support for area constraints definition, core allocation management and relocation, all in an integrated environment

• Generality: the flow is designed to support different reconfiguration models and communication infrastructures, allowing the user to create a wide range of reconfigurable architectures.

As previously introduced, we are particularly concerned with three aspects: defining suitable core area constraints, proposing an efficient way to manage online core placement and, finally, providing solutions for internal and fast core relocation.

The target self reconfigurable architecture is defined by a static region and a reconfigurable one, as shown in Figure 1; the figure shows, for simplicity, an architecture with 2 monodimensional reconfigurable slots, but the supported scenario includes 2D reconfiguration and a larger number of reconfigurable slots. The reconfigurable region is constituted by several reconfigurable slots and the reconfiguration of those areas, with the needed cores, is performed by a general purpose processor embedded in the static region.

The flow used to implement such an architecture starts from basic information provided by the designer, such as target application and device, the chosen communication infrastructure and reconfiguration model; this information is validated, and then given as input to the subsequent step, that performs the critical choices for the creation of the final solution. First of all, the system tries to obtain a reasonable solution for area constraints definition, based on the distribution of cores given by the user, on the structure of communication channels, on the cho-
Figure 1. Overview of the target self reconfigurable architecture

sen model and on the device structure and size. Different placement policies (for example First Fit, Best Fit, Bottom-Left Fit etc.) for this first definition are then applied, to choose the most suitable one for the specific case. Once both steps are complete, a relocation solution tailored to the system is added and the effectiveness of the resulting system is evaluated. This process is repeated, everytime exploiting the feedback from the previous ones, until a good compromise between all the important factors for the final reconfigurable system is met. Once the solution identification phase gives its response, the obtained information is shown to the user that can evaluate the goodness of the proposed solution and decide whether to approve it and proceed with the flow or to feed the system with additional constraints or hints to improve the results. Then, in the final step of the flow, support for the reconfigurable system is generated, in the
form of constraints definitions, a runtime allocation manager and a relocation solution. In the final step, this information is re-combined with that provided by the user at the beginning (i.e., target application and device and reconfiguration model), and used to generate the complete self, partially and dynamically reconfigurable architecture, in the form of static part bitstreams, partial bitstreams, blank bitstreams for deallocation and processor code for runtime allocation management.

1.3 Core Allocation Management

As previously introduced, this thesis deals in particular with one aspect of the global flow: the Core Allocation Management. The main concern is to allow efficient usage of the FPGA area when exploiting self partial and dynamical reconfiguration. In particular, this goal requires the creation of an efficient allocation manager to perform placement choices at runtime. The goodness of the obtained results can be measured according to several metrics, which include:

- Core Rejection Rate: the percentage of cores for which placement fails
- Application Completion time: the time that is needed for the user-defined application to complete, can also be compared to time taken with infinite resources or with different amounts of resources (reconfigurable area)
- Fragmentation: the fraction of the total area that is unusable because too scattered, this can be considered a measure for potential CRR
- Management Overhead: the computational and memory overhead introduced by the allocation management
• Routing Efficiency: the wiring costs defined by placement choices

Those metrics are often in contrast one with the other, and it is important to find a good compromise between them. One first concern lies then in the choice of different shaping policies, to define reasonable area constraints for cores to be configured on the target device. These shapes must be defined offline, as they require the logic modules to be actually syntesized, but an observation of the online behaviour of the placement system with various shaping policies can help devising a better one for subsequent implementations.

Another important direction for development of this work is the management of empty space inside the reconfigurable fabric, fundamental to further allow core placement decisions (33). The information on the empty space can be either maintained in a complete or in a heuristic way. The first approach requires more memory and more time to search for the best placement of a core at runtime but guarantees that, if a feasible location exists then it will be found. The second approach does not give this guarantee but, on the other hand, requires significantly less memory and can give a critical speedup in the placement algorithm runtime, which must be as fast as possible to avoid hindering the performances of the dynamically reconfigurable system.

Finally, this thesis deals with definition, evaluation and application of different fitting strategies (policies used to choose where to place a core among many acceptable solutions), to obtain good results. There are several criteria, but they can mainly be divided into general strategies and focused strategies. General strategies are the simplest ones; they choose among available locations without directly trying to minimize a particular objective function but with simple considerations on height, width or position. Focused strategies are more advanced, and try to
choose among available locations the one that directly maximizes or minimizes some specific objective function. Some examples of general fitting strategies are First Fit or Best Fit; on the other hand, focused strategies can, for example, try to directly minimize routing costs or fragmentation.
CHAPTER 2

PROBLEM DEFINITION

This work deals with partial and dynamical reconfiguration; in a partially reconfigurable system (3; 4), the functionality of a fraction of the total configurable logic can be changed according to the user’s needs, while leaving the rest of the reconfigurable device unchanged. In a dynamically reconfigurable system (11), this partial change can happen while the rest of the systems keeps performing computation.

We further classify those systems according to where the reconfiguration manager is located and to when the scheduling and placement choices and the generation of configuration bitstreams are performed. In the first case we distinguish between self reconfiguration and external reconfiguration (14). A self reconfigurable system, (1; 31) is completely independent from the outside in its management; for this reason, it has to host a reconfiguration manager in part of its logic and to store or be able to autonomously obtain configuration bitstreams when needed. On the other hand, an externally reconfigurable system depends on a configuration manager which is located outside the system itself and does not necessarily need to store or autonomously generate all the configuration bitstreams it needs as it can receive them from the manager along with configuration commands.

In the second case, we distinguish between run-time reconfiguration (16) and compile-time reconfiguration, (20). A system implements run-time reconfiguration if the computations needed to change the device state happen at the time of reconfiguration, while the system is running;
these computations include scheduling choices - what to configure -, placement choices - where to configure - and necessary bitstream generation or manipulation. Conversely, in a system implementing compile-time reconfiguration, those computations are performed prior to the reconfiguration process; in particular the schedule of what will be configured is static, allowing to statically define also its placement and to pre-generate and store the needed bitstreams which depend on functionality and on location of the module to be configured.

2.1 Two Dimensional Reconfiguration

Two Dimensional Reconfiguration, (7; 8) i.e. the possibility of configuring fractions of the FPGA of arbitrary rectangular shape as opposed to only whole columns as in 1D reconfiguration (29), involves all the previously described kinds of reconfigurable system, with effects that may also significantly vary from one kind to another and are both positive and negative.

For what concerns the positive effects, 2D reconfiguration carries several advantages in area usage and performances. A functionality can be implemented by logic in many ways, which generally differ in size and latency; each functionality has area-optimal implementations that minimize the amount of logic it uses up on board and performance-optimal implementations, which minimize the maximum combinatorial path, and thus latency.

With 1D Reconfiguration, however, the designer has no possibility of exploiting this efficiently as modules have a locked height which, in general, does not permit to effectively reach the optimal shape either in area or in performance. Figure 2 shows an example of this: if a functionality has an optimal shape of 2 rows and 2 columns and the target FPGA has columns of height 5 then we are doomed to use up an area of 5x2, 2.5 times the optimal one, to maintain
performance optimality, or to stretch its implementation in 5x1 to use up only 1.25 times the optimal area but with a possibly significant loss in performance.

Figure 2. Optimization due to 2D Reconfiguration

With 2D reconfiguration the possibility of choosing a shape which is closer to the optimal one for a functionality can have a dramatically positive impact on a reconfigurable system: using up less logic and, at the same time, obtaining optimal performance for single functionalities is a great improvement and is what makes 2D reconfiguration so appealing.

2.2 Problems with 2D Reconfiguration

Along with the advantages described in the previous section, 2D reconfiguration also causes several problems in reconfiguration management. Those problems can be summarized globally as an increase in the complexity of reconfiguration management due to the additional degree of freedom of reconfiguration.
In particular, a 2D-reconfigurable system is more exposed to fragmentation, has a significantly more troublesome placement phase, needs a complex communication infrastructures to ensure module interaction and, finally, requires a huge number of different configuration bitstreams to be generated due to the combinatorial explosion of its possible evolutions. The following sections explore in detail those four different problems and give some examples to further understand them.

2.2.1 Fragmentation

With the possibility to choose both the height and width of a core to be configured, the designer can optimize area usage obtaining significant savings in the amount of logic available on the FPGA. However, even if part of the area that would instead be wasted with 1D reconfiguration can be saved, we cannot always ensure that those pieces of logic are distributed in a way that permits to effectively use them.

The free logic cells that can be obtained by optimizing the shape of functionalities tend to be small and not clustered together; often they are located in such a way that is practically impossible to make use of them except with really small cores. Those unusable fragments can significantly reduce the advantage in area usage that is obtained with 2D reconfiguration and must be taken into account when defining the minimal configurable slot for the system and when making placement decisions. An example of Fragmentation Aware placement method can be found in (34).

Figure 3 depicts a toy example useful to show how easily fragmentation appears when recurring to 2D reconfiguration. The example FPGA has 4 rows and 4 columns; there are 3
configured cores of respective size 4 cells, 3 cells and 4 cells. On the left is shown the result of allocating those cores in a 2D reconfigurable system, while the right figure depicts the same cores placed as columns of a 1D reconfigurable system. While the overall free area (marked as *Usable*) in the bi–dimensional case is more than in the mono–dimensional one (5 cells vs 4 cells), free area in the 1D example is more clustered and has a higher probability of being effectively used. On the extreme, if the smallest core available for the example system requires at least 3 cells, then we actually have more wasted space in the 2D example than in 1D (the 2 cells marked with "‘Usable?’" vs the "‘Unusable’" one).

### 2.2.2 Core Allocation Manager Choices

With a mono–dimensional approach, the placement of a core is univocally defined by the column where its configuration starts; in a system with *k* columns a core can be placed in at most *k* different ways, making placement decisions quite simple. When trying to exploit the bi–dimensional approach, instead, things become complex; the starting column is not sufficient anymore to define a location as the starting row is also needed along with the height and width of the core.

After the shape for a core has been defined by the designer according to desired optimizations, the row and column where to place it must be chosen by the Core Allocation Manager in such a way to ensure that the system performances meet the expectations (35). While technically it is not necessary to explicitly take care of clever placement decisions, as cores could be simply put in the first available area big enough to host them, this approach can lead to severe waste of combinatorial logic due to fragmentation and eliminate the area saving advan-
tage of 2D reconfiguration (34). Bad placement decisions can also prevent further allocation of needed functionalities and thus introduce a delay in scheduled computation which can make the performance optimizations obtained with more compact mappings completely useless.

Again, an example is provided to show how much placement choices can be critical, even in really small systems (Figure 4). The example FPGA has 3 rows and 3 columns; the cores are: $A$ (2x2), $B$ (2x1) and $C$ (1x3); $A$ and $B$ are needed from Step 1 of the schedule and $C$ is needed from Step 2. Choice $A$, depicted on the upper half, is better in that it allows the subsequent allocation of core $C$ when needed at the second step; choice $B$, instead, is not so good and causes a delay in the schedule when $C$ is needed but cannot be placed due to lack of space and is delayed until the completion of either $A$ or $B$.

In this really small example the optimal choice is evident; however, when considering several schedule steps and much bigger FPGA, where many cores can be placed together, the decision is not so easy and must be weighted accurately to obtain good results and minimize delay.

2.2.3 Communication Infrastructure

If cores can be freely placed in the reconfigurable area, the designer cannot easily define a statical infrastructure spanning the whole board to allow communication between modules as it could be done in 1D architectures (36). This problem also grows with the granularity of bi-dimensional reconfiguration: for each additional independent module that can be placed along the 2nd dimension there is need for one more attachment point to the communication infrastructure.
Figure 3. Fragmentation in 2D Reconfiguration

Figure 4. Placement Decisions
Figure 5 shows an example of this; the same cores have been placed in a 1D reconfigurable architecture and in a 2D one, using the same communication infrastructure. Again, it can be seen how the 2D approach permits a better area usage, leaving space available for an additional functionality, while the mono-dimensional one leaves unusable leftover space. On the other hand, if the 2D reconfigurable system is connected with a simple horizontal bus, as common in the 1D case, the area improvements are immediately canceled by the fact that some of the configured logic cannot be connected to the system (Cores A and D).

![Figure 5. Problems with Simple Static Bus](image)

The example shows also that the number of required buses to always ensure complete connection is in fact directly tied to the minimum granularity of reconfiguration offered by the device. If we consider the example FPGA with 15 independently configurable units, arranged in 3 rows and 5 columns, then the minimum number of horizontal buses needed to completely
connect the device is equal to the number of independent rows minus one (because they can be placed at the borderline between module, saving one of them), plus a vertical one that must be used to connect each bus with the others. In a similar way it can be chosen to create several vertical buses, leading to a minimum number equal to the number of columns minus one plus an horizontal one. Which one to choose between horizontal and vertical is up to the designer’s needs and strongly depends on the physical device structure, the example takes an horizontal one as that approach is the one viable with Virtex FPGAs, onto which previous related work has been done.

The critical part, then, lies in the definition of the minimum granularity that the designer wants to allow for 2D reconfiguration, as a too fine-grained choice could lead to really complex and expensive infrastructures while a too coarse-grained one could keep low infrastructure complexity but at a loss for area optimization which is the main reason for doing 2D reconfiguration.

2.2.4 Configuration Bitstreams Generation

To effectively exploit the features of a self dynamically reconfigurable system, it is necessary that configuration bitstreams are available on board when required for the reconfiguration process. As the timing of the bitstream creation phase is incompatible with the speed requirements of dynamical reconfiguration and the system must be autonomous from the outside, it is necessary to pre-generate and store the configuration files in the internal memory of the reconfigurable system.

In a compile-time 2D reconfiguration scenario, it is known beforehand what will be configured on board at each stage of the system evolution, along with where, in terms of row and
column coordinates, those 2D functionalities are going to be placed. This permits to exhaustively generate all the necessary bitstreams to further store them in the internal memory of the system; those bitstreams will then be used to reconfigure the device when needed. Furthermore, as all the evolutions of the system are known at compile-time, the designer can exploit difference bitstreams reconfiguration, generating small configuration files that globally describe only what changes from one state to another, without explicitly allocating or deallocating independent modules and using only a small amount of memory to store bitstreams.

On the other hand, in a run-time reconfiguration scenario, the designer only knows a set of cores that could possibly be placed on the device at some time in its evolution, and has no idea on where those functionalities will be mapped on the FPGA and what they are going to replace. This nullifies the applicability of the difference bitstreams approach and forces the use of module-based reconfiguration. With the module based approach, every functionality is described by a specific configuration file, defining an independent 2D core; those cores can then be allocated on the device using their characteristic bitstream and deallocated by using a bitstream defining a blank module.

In principle, a self run-time dynamically reconfigurable system using the described approach should store, in its internal memory, all the bitstreams that map every functionality that could be possibly hosted on the device into every possible 2D location. This consideration makes the exhaustive generation of configuration bitstreams computationally very expensive and prohibitive in terms of memory usage on the device; this problem is encountered also in the mono-dimensional case but, with the additional degree of freedom offered by 2D reconfigura-
tion, things become even worse. This is due to the fact that the possible placement choices for each core, which in the mono-dimensional case were $c$, e.g. the number of available vertical slots in the FPGA, become $c \cdot r$, with $r$ equal to the number of rows of the reconfigurable device.

In 2D reconfiguration, then, each functionality has $c \cdot r$ different mappings, and thus requires $c \cdot r$ different configuration bitstreams available to be freely configured at run-time. As done with 1D reconfiguration (37), this can be solved by means of relocation: one single bitstream is created for each different functionality that could possibly be allocated in the system and all the other configuration files will be obtained at run-time. This is achieved by manipulating the content of the bitstream, only altering a small part of it to change the final mapping position of the core on the device. Clearly, those manipulations, will not involve only the column address on the device, as was done previously, but should also alter information on the row, which is also contained in bitstreams for devices that support 2D reconfiguration.

2.3 Desired Features for the Core Allocation Manager

The problems that originate when evolving from 1D reconfiguration to a bi-dimensional scenario must be managed in such a way to allow an efficient implementation of the reconfigurable system. The main criterion to evaluate efficiency of the reconfigurable system implementation is performance which, in turn, depends on several aspects.

First of all, we have to consider the time overhead introduced by reconfiguration management. When a new module needs to be configured on board, the system has to check whether there is enough empty space or not and, if yes, choose a suitable location to place it; subsequently, the bitstream configuring the requested functionality must be relocated and sent to the
interface that performs the reconfiguration process; only upon completion of that, the module can start to compute.

As a consequence, it is important that the system is able to find a suitable location to place a new module, because if this does not happen the core it implements must be delayed, possibly exceeding its deadline. A first metric that can be used to evaluate this is fragmentation grade, which can be seen as the unlikelihood of being able to place new cores. The more fragmented is the configurable fabric, the less is the probability of being able to place a core when requested. Another, more direct, measure for goodness of a placement solution is the average Core Rejection Rate (CRR), e.g. the percentage of functionalities that cannot be placed on the device before their deadline expiration, if any. This value can be computed on a benchmark set including cores of various size, shape and distribution. One more possibility becomes available when considering a scenario with no deadlines but the only concern of completing as fast as possible, given limited resources (i.e. FPGA area); this measure compares actual application completion time versus the time that would be achieved in an infinite resources scenario (or among devices of different size).

For what concerns the pure reconfiguration management overhead, it is also important that the algorithm chosen for placement is fast, as the time spent to find a suitable location introduces an additional delay in the system computation. As we aim to obtain good performance, a second measure to be considered for quality of a solution is then its timing requirements, which can be evaluated asymptotically as time complexity in the number of placed cores or by measuring the effective time spent on benchmark sets.
Along with timing requirements, a placement solution also uses part of the reconfigurable device internal memory: at every moment in the system evolution, the reconfiguration manager must know exactly which parts of the FPGA are free and the shape of all the available locations to decide whether a module can be placed or not and where to place it. If internal memory is scarce, then it is important to measure also this aspect. Memory requirements for the online placement solution can be evaluated by computing its asymptotical spatial complexity in the number of already placed cores.

Finally, on a lower level of abstraction, the physical latencies of the system can be taken into account; when placing modules in different locations their internal state does not change, and thus neither their individual performance do, but the relative distances between interdependent functionalities may vary, leading to possible performance losses due to longer routing paths. This means that another metric can be considered: routing paths length; a possible measure for this is the sum of weighted distances, either euclidean or manhattan, between interdependent cores. Other, more complex metrics could also be considered, for example in a hierarchical routing scenario.
CHAPTER 3

RELATED WORKS

Several works concerning the defined problem are present in literature; however they mainly address individual aspects of the problem, such as core relocation, or core placement. Currently, there is no integrated solution to give the designer a complete framework for defining and handling a self, partially and dynamically reconfigurable system; the availability of such a solution could greatly increase the reward in exploiting this more complex approach.

Since the scope of this thesis is mainly the core allocation management aspect of our proposed solution to automate the creation and management of self dynamically reconfigurable systems and, at present time, no integrated solution exists, in this chapter we will focus on existing solutions for that specific aspect.

In literature, several solutions to the problem of runtime core allocation management for 2D reconfigurable systems can be found; this section presents the most relevant of them. Along with a brief explanation of the basic ideas behind each solution, the specific advantages and disadvantages of each one of them are analyzed. The main criteria used in the evaluation are quality of placement (or fragmentation grade), time and memory requirements, especially in terms of asymptotic complexity, and total modules interconnection cost.
3.1 **Definitions and Metrics**

Before examining integrated solutions for 2D reconfiguration management, some additional concepts relative to the individual aspects of fragmentation, placement and cores interconnection have to be defined.

First of all we can consider two different kinds of fragmentation metric (34): *absolute* metrics and *relative* ones. An absolute fragmentation metric only takes into account the distribution of empty space on the reconfigurable device, with no regard to the distribution of size and shape of the incoming cores; on the other hand, a relative fragmentation metric does also consider the probabilities that a core of a certain size or shape will be requested for reconfiguration at some time. The consequence is that absolute metrics have a more general value, being independent from the specific functionalities implemented on the reconfigurable system; relative metrics, instead, give a more accurate estimate of the likeliness that a core will fit on the device, which varies not only with the fragmentation grade of logic but also with the size and shape distribution of cores.

Fragmentation can be dealt with in either a reactive or proactive way. Reactive solutions try to reduce fragmentation after its insurgence; this resembles the concept of defragmentation and is typically done by moving modules placed in locations that are bad for fragmentation to a more suitable location. This approach has the obvious disadvantage that it introduces a huge overhead due to additional reconfiguration time of the core that is being moved and requires either preemption of the implemented core, which causes additional loss of performance, or a really complex system of logic replication to keep computation going which, by the way, can
only be used to move logic between adjacent locations (39). Another possibility to deal with already fragmented systems is footprint transform, which involves the repositioning of subcores inside a module to change its shape (38). Fragmentation can also be limited in a proactive way, trying to prevent its insurgence; this can be done with good placement techniques, that can either try to directly minimize it, like Fragmentation Aware placement (34), or indirectly by keeping used area as clustered as possible.

For what concerns modules interconnection, we can basically choose between two approaches: a *static* communication infrastructure and a *dynamical* one. A static infrastructure is always present on board and thus can easily allow communication between all the modules that are executing, even during deallocation of unused cores and allocation of new ones; the price for this simplicity is the high amount of area that must be devoted to the infrastructure; the structure must be composed of some sort of grid with a sufficiently small granularity to permit attachment of cores wherever they could be placed. Another disadvantage of this approach is in latency, which is higher due to the width of the communication network and the low scalability of a complete static infrastructure.

The alternative to this is a dynamical infrastructure, whose routing resources are only present where and when needed, connecting only cores that communicate one with the other. With this approach, some sort of backup path must be provided before deallocating or allocating part of the logic which includes routing resources to guarantee the connection of the remaining parts of the device. The advantage of this approach is a great amount of resource saving due to significantly smaller infrastructure and lower latency, provided that the placement algorithm
also tries to minimize relative distance between interdependent cores; the disadvantage is that configuring also routing resources at runtime adds further reconfiguration overhead.

Finally, regarding the placement manager (33), it is normally composed by an empty space manager and a fitter. The empty space manager keeps memory of the state of configurable area, providing an easy and fast navigation through placement possibilities to the fitter; for example, it can maintain a list of all empty slots along with their height and width. The fitter, instead, has the task of effectively choosing which of the available locations is the best according to some goodness metric, such as likelyhood of of being able to place new cores in the future, or absolute fragmentation, or weighted relative distance between interdependent cores.

3.2 Fitting Strategies

Once a technique has been defined for empty space management, some criterion to choose between suitable locations has to be defined. There are several criteria, but they can mainly be divided into general strategies and focused strategies.

3.2.1 General Strategies

General strategies are the simplest ones; they choose among available locations without directly trying to minimize a particular objective function but with simple considerations on height, width or position. The most common general fitting strategies are:

- First Fit (FF): chooses the first found free location that can accommodate the core, trying to save search time.
• Bottom Left Fit (BL): among all location that can accommodate the core, BL chooses the one closest to the bottom left corner of the device, heuristically trying to cluster used area.

• Best Fit (BF): chooses the smallest free location that can accommodate the core, trying to minimize leftover space.

• Worst Fit (WF): chooses the biggest free location that can accommodate the core, trying to leave large free slots as leftovers.

• Best Fit with Exact Fit (BFEF): among all free locations which can accommodate the core, BFEF chooses the smallest one which has exactly the same width or height as the core. If no such rectangle is found, the core is placed according to BF.

• Worst Fit with Exact Fit (WFEF): among all free locations which can accommodate the core, BFEF chooses the biggest one which has exactly the same width or height as the core. If no such rectangle is found, the core is placed according to WF.

3.2.2 Focused Strategies

Focused strategies are more complex, and try to choose among available locations the one that directly maximizes or minimizes a specific objective function. Some relevant examples of focused fitting strategies are the following:

• Fragmentation Aware (34) (FA): among all free locations which can accommodate the core, FA chooses the one which gives the least contribute to device fragmentation which is computed on the whole area according to some fragmentation metric.
• Routing Aware (40) (RA): among all free locations which can accommodate the core, FA chooses the one which gives the least contribute to total cost of inter-core routing resources. The cost is computed as a function of relative amount of needed communication, number of interconnected cores and some distance function, in particular euclidean or manhattan. Also more complex metrics can be considered, for example in the case of hierarchical routing.

• Least Interference (41) (LI): among all the locations which can accommodate the core, BL chooses the one that interferes with the least number of placed cores. This is particularly useful in a hybrid static-dynamic reconfiguration scenario to minimize the number of stopped cores during reconfiguration.

3.3 **KAMER method**

One of the first solutions for FPGA empty space management was proposed by Bazargan and others in 2000 (33). This solution is commonly referred as KAMER, which is an acronym for Keeping All Maximally Empty Rectangles. The basic idea behind this approach is to maintain a data structure that holds the current status of FPGA area as a set of rectangles; each rectangle represents a possible placement location. More in detail, only maximally empty rectangles are kept, as subrectangles do not give any additional utility in the placement process; the reason for this is the following: if a core can be placed in a specific rectangle, then it trivially can also be placed in every other rectangle that fully contains it. Figure 6 illustrates the concept with an example: two cores are placed in sequence on the device and the maximally empty rectangles are updated.
The main strength of this approach lies in its completeness. The KAMER algorithm guarantees that, if a sufficiently large free location for placement exists on the reconfigurable device, then it will be found. This is not a trivial property, as its enforcement has a significant cost in term of algorithmical complexity but, at the same time, guarantees a really high quality of placement; this high cost directly depends on the necessity of keeping track of every maximally empty rectangle at each time instant to guarantee completeness. This is quite expensive as the number of maximally empty rectangles in a reconfigurable system grows quadratically in the number of placed modules; the reason for this lies in the fact that, every time a new module is placed, a potentially large amount of new maximally empty rectangles is created. In fact it can be shown that this number is proportional to \( n \), the number of placed cores; if the addition of each module creates \( \vartheta(n) \) new maximally empty rectangles, then the total number of maximally empty rectangles is effectively proportional to the square of the placed modules.
This easily leads to determine that the asymptotical space complexity of the KAMER algorithm is \( \vartheta(n^2) \), as all the rectangles must be stored, and its asymptotical time complexity is also \( \vartheta(n^2) \) as, in the worst case, each one of those rectangles must be checked for placement. The criteria that the authors propose to be used in conjunction with this empty space management method are general ones, in particular First Fit and Best Fit.

Summing it up, the KAMER method permits to achieve a really high quality for placement as it guarantees that, if a placement location exists, then it will be found, by keeping complete information on the empty space in the form of all maximally empty rectangles. The drawback for this lies in its high complexity, which is quadratical both in time and space. The method is particularly valuable as benchmark to measure the performance of simplified algorithms as it can be considered the “optimal” in terms of quality.

### 3.4 KNER method

The creators of KAMER proposed also another, simplified, solution: KNER (33); again, the name is an acronym, which stands for Keeping Non-overlapping Empty Rectangles. This solution was created with the explicit purpose of reducing the complexity of KAMER, while still obtaining a reasonably good quality of placement, in terms of core rejection rate. The main difference between KAMER and KNER lies in the kind of rectangles that are taken in consideration when keeping track of the FPGA state. While both solutions only consider maximally empty rectangles for the reason stated before, KAMER keeps all those rectangles, while KNER puts a limitation on their number.
In particular, only non-overlapping rectangles are considered with the KNER method; 3.4 shows the difference between the KAMER and KNER approach in the basic case of a single core placed in the bottom left corner of an empty device.

Figure 7. All Maximally Empty Rectangles vs only Non-Overlapping

The choice of this restriction was driven by the need to strongly reduce the amount of information on the FPGA state to be maintained at each time step; the approach succeeds in that, passing from $n$ potential new rectangles created per every placed core to a small constant number per placed core. This is accomplished by performing an heuristic split decision of the remaining empty space every time a core is placed inside a rectangle.

More in detail, when a core is placed in a non-overlapping empty rectangle, the remaining space from that rectangle is split in a number of new non-overlapping rectangles that can be
0, 1 or 2 (depending on which dimensions are filled by the core: 0 if it fills up both height and width, 1 if it fills up either one and 2 if neither). This means that the number of rectangles to be kept can be, at most, twice the number of placed modules; on the other hand, with KAMER, every time a new core is placed, not only the chosen rectangle has to be split but, at worst, every rectangle on the device as they could all overlap, leading to the quadratical growth.

With this consideration, the asymptotical time complexity of KNER is $\Theta(n)$ as the number of checks to be performed in the worst case is linear in the number of placed cores. The space complexity is also $\Theta(n)$ as the number of stored rectangles is proportional to $n$. The price for this much better efficiency lies in the loss of completeness as the algorithm uses an heuristic in choosing how to partition rectangles in case of overlapping. This means that, in an unlucky situation, KNER could be unable to find a suitable location for placement, even if it exists, leading to an higher core rejection rate. Also for this method, the authors propose usage in conjunction with a general purpose fitting strategies such as First Fit or Best Fit.

### 3.5 CUR method

A completely different approach is the Contour of Union of Rectangles (CUR) method, which is based on computational geometry and is described in (40). The authors of the paper give an algorithm that matches the lower bound in time complexity for a non-heuristic solution of the 2D placement problem. The value of the lower bound which comes from previous studies in the field of computational geometry is $\Theta(n \log(n))$.

With the CUR approach, instead of representing empty space on the FPGA as a set of currently free rectangles, a different kind of information is kept in memory. The algorithm
creates a data structure holding the global contour of all the clusters of modules that are currently placed on the FPGA. The contour is simply a list of segments, which define the border between free and occupied space.

This information can be used to determine a suitable location for placement with some simple manipulations: all the placed modules widths (heights) are enlarged by half the width (height) of the core to be placed. An analogous manipulation is applied to the border of the device, which is shrunk by the same amount. After this modification, the core can be treated like a single point, which has no shape, and can be placed anywhere inside the space that remained free. Figure 8 illustrates this process, the striped area in the right picture represents feasible locations for the core which became a point.

![Figure 8. Finding a feasible location with CUR](image)

This solution was combined by the authors with a fitting strategy focused on minimizing routing costs, in the form of total distance between communicating modules. Given the way it
works, the CUR approach is particularly suitable to exploit a focused fitting strategy, especially one based on a distance metric, while it loses part of its value for general strategies: in particular, it does not adapt well to criteria such as Best Fit, Worst Fit and derivates.

For what concerns complexity, the number of segments that must be stored by this method grows linearly in the number of placed cores, leading to a space complexity of $\vartheta(n)$. The CUR method gives the best performance in terms of execution time for an algorithm that keeps complete information on the empty space and provides a quality of placement which is basically equal to that of KAMER.

However, a time complexity of $\vartheta(n\log(n))$, even if it is the lower bound for keeping all the information, may still be too high; the placement algorithm will be run every time a new core must be placed, and its running time will directly impact on the global performance of the reconfigurable system, as cores will wait for its completion before being actually placed. For this reason, along with complete information methods, also heuristic algorithms are being considered with the goal of finding the best compromise between the time overhead due to placement decisions and the cost of core rejection.

### 3.6 2D-Hashing method

The 2D-Hashing method, as its name implies, is based on a hash table structure, maintained by the empty space manager and accessed when needing to know the location for placement. The solution was proposed by Walder and others in 2003 (42) and provides a significant improvement in the time necessary to find a suitable placement location by posing some restrictions on the supported fitting strategies.
The basic idea behind this approach is the consideration that, if the fitting criterion does not depend on features of the specific core to be placed other than its size and shape, then those fitting choices can be performed in advance. This means that, upon placement location request, the allocation manager only has to check the correct entry in a table that stores the answer which was computed before the core arrival.

In practice, a set of non-overlapping rectangles, similar to that of KNER, is kept in the hash table; the table is indexed by values of height and width for a core and in each cell, a linked list of all rectangles that can accommodate that shape is stored. The strength of this method lies in the fact that the “best” location (according to the chosen fitting strategy) for each cell does not depend on the core, as long as it is small enough to fit in there; this means that each list inside a cell can be autonomously maintained in such a way to provide rectangles in order of goodness (the simplest one being from smallest to biggest).

The computations to order these lists can be performed during the reconfiguration process of a previous core, which will normally take longer and, when a new core requires a placement location, the algorithm can simply check the hash table for a rectangle of the needed shape and return the found value. If, for a given core, no entry is found in the table, then there is no feasible location for its placement, and the core is rejected.

The way the 2D-Hash algorithm works implies that it has a constant time complexity, $\Theta(1)$, which is an extremely good result, as other heuristic solutions have a linear complexity. The space complexity of the method is linear, as one entry is stored in the table for each rectangle plus one pointer for each of those rectangles; this is correct under the assumption that the
number of rows times the number of columns is significantly smaller than that of the cores to be placed, otherwise, the memory requirement becomes proportional to the number of rows times the number of columns.

Precomputation of the requested location allows the 2D-Hashing method to be extremely fast, with a constant time complexity; for what concerns quality of placement the method is comparable to KNER, as it is basically an enhanced version of it in terms of running time and it keeps information on the free area in the same way. The main drawback of this solution lies in the fact that it cannot be applied to focused fitting strategies, such as routing costs minimization. The reason for this is that those strategies require runtime computations that also depend on the specific core to be placed which is not known a priori. On the other hand, when the placement decision does not depend on features of a core other than height and width, for example when applying Best Fit, 2D-Hashing represents a valid solution, extremely fast and with relatively good placement quality.

3.7 Global Comparison

In this section, the literature techniques previously presented are compared to give an overall view of the possibilities for 2D reconfiguration management; Table I shows a brief summary of the features of the methods that were previously described: KAMER, CUR, KNER and 2D-HASHING.

The main criteria that were considered in the global comparison are placement quality, which tends to be higher for methods that exploit complete information and lower for heuristics, time
TABLE I

<table>
<thead>
<tr>
<th>Method</th>
<th>Placement Quality</th>
<th>Time Compl.</th>
<th>Space Compl.</th>
<th>Applicable Fit Strats</th>
</tr>
</thead>
<tbody>
<tr>
<td>KAMER</td>
<td>Complete</td>
<td>$\vartheta(n^2)$</td>
<td>$\vartheta(n^2)$</td>
<td>General, Focused</td>
</tr>
<tr>
<td>CUR</td>
<td>Complete</td>
<td>$\vartheta(n \log n)$</td>
<td>$\vartheta(n)$</td>
<td>FF, BL, Focused</td>
</tr>
<tr>
<td>KNER</td>
<td>Heuristic</td>
<td>$\vartheta(n)$</td>
<td>$\vartheta(n)$</td>
<td>General, Focused</td>
</tr>
<tr>
<td>HASH</td>
<td>Heuristic</td>
<td>$\vartheta(1)$</td>
<td>$\vartheta(n)$</td>
<td>General only</td>
</tr>
</tbody>
</table>

As can be seen from the data shown in the table no method clearly dominates the others.

As an example, both KAMER and CUR offer top placement quality, due to their larger search space, when looking for a placement solution but are also slower than their heuristic counterparts. CUR has a lower complexity than KAMER but is also less versatile in terms of applicable fitting strategies: for example it cannot exploit the simple but effective Best Fit.

On the other hand, heuristic methods have a lower placement quality than those offered by KAMER and CUR but also tend to be faster. 2D-HASH has the lowest complexity among all the solutions, but still is not strictly better than KNER for what concerns fitting strategies; 2D-HASH only works with a general strategy, while the other method can support all kinds of fitting strategy.
Memory requirements tend to be linear in the number of placed cores for all the methods, except KAMER, which requires an amount of memory which is quadratical in the number of placed cores.

It is interesting to note that the only combined solution that actually takes into account routing costs minimization is the CUR method. Inter-core routing can be an relevant factor in the performance of a reconfigurable systems; more in general, the possibility of supporting fitting strategies of the focused type can be important for a core allocation management solution.
CHAPTER 4

PROPOSED APPROACH

The system creation and management problems play a relevant role in the field of self reconfigurable architectures; for this reason, we propose a workflow to provide support to those phases. The sections that follow provide a description of both the global workflow and of the aspects that are more specifically part of this thesis. In particular, Section 4.1 illustrates the approach that was taken in the definition of our complete workflow for the creation of a customized self reconfigurable architecture and the phases that are part of the flow. In Section 4.3 and 4.2, instead, we describe the parts of the flow that fall more directly in the scope of this thesis; Section 4.2 describes the policies that were defined to achieve automatic or semi-automatic definition of Area Constraints for Cores, while Section 4.3 illustrates the choices that were made in developing our solutions for online Core Allocation Management.

4.1 The Global Workflow

As previously introduced, we propose a novel workflow to aid the designer in the creation and management of self partially and dynamically reconfigurable systems by automating a significant part of the necessary work. In particular, the scope of this thesis covers mainly the core allocation management aspect. This section provides a general description of the global workflow, while the rest of the chapter will be devoted to the description of the parts of the
flow that are more directly involved in this thesis: Core Allocation Management and Area
Constraints Definition.

We consider the input application as composed of a set of functionalities; each functionality
has to be implemented on the target reconfigurable device. In such a context we will refer to
the physical implementation of each functionality as a core; each core is described by a partial
reconfiguration bitstream. The self reconfigurable architecture we are targeting is defined by a
static region and a reconfigurable one, as proposed in Figure 9.

Figure 9. Overview of the target self reconfigurable architecture
The reconfigurable region can be seen as composed by several reconfigurable areas and the reconfiguration of those areas, with the needed cores, is managed using a general purpose processor embedded in the static region. Figure 10 shows the general structure of the workflow.

The flow starts from basic information provided by the designer, such as target application and device, the chosen communication infrastructure and reconfiguration model and other preferences, for example number of shapes allowed per Core; this information is validated and then given as input to the Solution Identification phase that represents the core of our approach. This stage of the flow performs extensive simulation of the scenarios generated by different initial choices and, in the end, uses its feedback to perform the final architectural choices. Once this phase gives its response, the obtained information is shown to the user that can evaluate the goodness of the proposed solution and decide whether to approve it and proceed with the
flow or to feed the system with additional constraints or hints to improve the results. Then, the support for the reconfigurable system is generated, in the form of constraints definitions, a runtime allocation manager and a relocation solution. In the final step, this information is re-combined with that provided by the user at the beginning (i.e., target application and device and reconfiguration model), and used to generate the complete self, partially and dynamically reconfigurable architecture, in the form of static part bitstream, partial bitstreams, blank bitstreams for deallocation and processor code for allocation management.

4.1.1 Input Management and Validation Phase

The input data provided by the user includes the specification of the target application, the information on the device that is going to be used to implement the system, the chosen reconfiguration model (1D or 2D) and the desired communication infrastructure. In addition to that, the user can specify other options, which include the desired granularity level for reconfigurable area partitioning, the number of different shapes that will be allowed per Core and the tightness of the area constraints that will be generated. Target application is specified as a set of cores, which represent a functionality that will be implemented with reconfigurable logic; the level of detail at which the application is specified can vary greatly, depending on the user’s needs: it can range from simply selecting a set of Cores from a predefined library to fully defining a whole application, composed of customized Cores. The designer can also impose customized constraints on each of the Cores; in particular the user can provide predefined area constraints and deadlines for execution. The information on the device mainly include its size, in rows and columns, and how blocks of different kind, for example logic blocks and RAM
blocks, are distributed. Again, if desired, the user can impose a specific partitioning of the reconfigurable device, overriding its generation by the workflow. Finally, the communication infrastructure definition includes the type of infrastructure, for example bus or point to point, the position of attachment points for cores and the link capacities. This phase is extremely important since it has the objective to check the input information, under which the design has to be defined, provided by the user. A typical case is a scenario where the user request for a 2D reconfiguration approach but he/she chose a reconfigurable device which does not provide this capability.

4.1.2 Solution Identification Phase

This phase is the part of the flow that performs the critical choices for the creation of the final architecture. It does so by exploiting the basic information given by the designer plus a large volume of data obtained by extensive simulation of the scenarios that arise when considering different choices. First of all, the system tries to obtain a reasonable solution for area constraints definition and reconfigurable device partitioning, based on the distribution of cores given by the user, on the structure of communication channels, on the chosen model and on the device structure and size. In addition to that, the possibility of allowing more than one implementation per Core is also considered; in particular, large cores will have, in general, more different implementations than small ones, to improve their chances at being successfully placed. Different placement policies for this first definition are then experimented, to choose the most suitable one for this specific scenario. Once both steps are complete, the most suitable relocation solution is added and the effectiveness of the resulting system is evaluated. This
process is repeated many times, each one using the feedback from the previous ones, until it reaches a good compromise between all the important factors for the final reconfigurable system. The most promising solution is then proposed to the user that can accept it, examine other found solutions or completely restart the process by giving some additional constraints or hints to the system.

This phase of the flow is the largest and most important one and includes, as sub-parts, the aspects that will be described more in detail in the second part of this chapter and that are more directly tied with this thesis. The first of these sub-parts is Area Constraints Definition; the identification of a solution starts from a first heuristic definition of contraints and proceeds by modifying the previous choices at each iteration by exploiting the simulation feedback. The way this is done is explained in detail in Section 4.2. The second sub-part, Placement Policy Selection, makes use of the versatile Core Allocation Management solution that we developed for this thesis and which is described in detail in Section 4.3.

4.1.3 System Generation Phase

When a satisfying solution is obtained at the end of the simulation steps, the flow effectively generates the components of the new reconfigurable system: the area constraints definition together with the partitioning of the device, an online core allocation manager based on the chosen policy and a bitstream relocation solution. The area constraints definition is represented by an UCF file, which contains information on the shape of every Core (of every RFU in the case of multiple implementations per Core); the allocation manager is provided in the form of processor code that will run inside the system, maintaining the empty space and performing
the online decisions necessary to place new cores on the device. Finally, the suitable relocation filter, described in VHDL, is added to the static part of the architecture and will be used to obtain the configuration bitstreams for arbitrary locations at runtime. Those parts are ready to be used for the final system.

4.1.4 Bitstreams Generation Phase

The final part of the flow includes the generation of the actual implementation of the architecture, mainly in the form of bitstreams and processor code. One of those files is the bitstream which defines the static part of the reconfigurable architecture, which includes communication infrastructure, processor, internal memory and relocation filter. This phase also generates all the partial bitstreams that define the various cores of the application implemented using the area constraints that have been chosen for them and in a generic location. Again, the bitstreams used to configure a specific location will be obtained at runtime starting from these generic files; for what concerns the possibility of allowing different implementations per Core, each Core will be associated to one generic bitstream per each shape that it will be able to assume. In other words, the total number of bitstreams that will be stored in the internal memory is equal to the number of Cores times the average number of different shapes that they can assume. For this reason, we can arbitrarily change at runtime the location of a Core but, once the system is online, we will only allow a limited number of physically different implementations to each of them to avoid an explosion in the amount of memory that would be necessary otherwise. In addition to the partial configuration bitstreams, a blank module for each allowed shape has to be generated as it is necessary to erase partial configurations. The last output of the flow is the
processor code for the allocation manager that will perform at runtime the decisions needed to
place new cores on the reconfigurable device.

4.2 Area Constraints Definition

The previous section described the proposed workflow as a whole; this section illustrates
one of the sub-parts of the flow that falls directly in the scope of this thesis: Area Constraints
Definition.

To keep high versatility, we choose to support both totally automated area constraints defi-
nition and designer-assisted definition; if desired, the user can choose to specify the area con-
straints for some or all of the given Cores. Another option that we choose to leave to the
designer is related to the number of shapes per Core: the user can specify the maximum num-
ber of different shapes that system should consider for each Core in the application. Finally,
the user is also allowed to participate in this step by specifying how tight the constraints that
will be automatically defined should be; this way, the system can be guided towards solutions
that try to save as much area as possible or towards solutions with less tight constraints.

As briefly introduced before, during the Solution Identification phase, the system keeps
altering the current Area Constraints Definitions at each iteration, exploiting the feedback that
is given by the simulator; clearly, constraints that were specified by the user are excluded from
this alteration. In the first iteration of the process, for Cores with no user defined constraints,
the system defines a first simple heuristic implementation which tries to achieve a shape that is
as square-like as possible. The slot size and number for the final reconfigurable system, again,
if not specified by the user, is chosen based on the initial constraints, both those given by
the user and those automatically defined by the system. During the iterations of the process, additional shapes are considered for those cores that in simulation resulted difficult to be placed; depending on the preferences that were specified by the user, those additional shapes can be used either instead of the previous ones or in addition to them, allowing a multiple shapes placement scenario. At the end of the process, the constraints that gave the best overall results in simulation are kept for the final solution. A more detailed explanation of how this was implemented is presented in the next chapter.

4.3 Core Allocation Management

This section gives a description of the choices that were made in the definition of the Core Allocation Manager solution that was developed for this thesis and is used by the Solution Identification phase of the complete workflow. As previously explained, the most important concern of the proposed workflow is to allow efficient usage of the FPGA area when exploiting self partial and dynamical reconfiguration. In particular, this goal requires the creation of an efficient allocation manager to perform placement choices at runtime. The goodness of the obtained results can be measured according to several metrics, which include:

- Core Rejection Rate: the percentage of cores for which placement fails;
- Application Completion time: the time that is needed for the user-defined application to complete, can also be compared to time taken with infinite resources or with different amounts of resources (reconfigurable area);
- Fragmentation: the fraction of the total area that is unusable because too scattered, this can be considered a measure for potential CRR;
• Management Overhead: the computational and memory overhead introduced by the allocation management;

• Routing Efficiency: the wiring costs defined by placement choices.

Those metrics are often in contrast one with the other, and it is important to find a good compromise between them. One first concern lies then in the choice of different shaping policies, to define reasonable area constraints for cores to be configured on the target device. These shapes must be defined offline, as they require the logic modules to be actually synthesized, but an observation of the online behaviour of the placement system with various shaping policies can help devising a better one for subsequent implementations.

Another important direction for development is the management of empty space inside the reconfigurable fabric, fundamental to further allow core placement decisions. As introduced in the previous chapters, the information on the empty space can be either maintained in a complete way or be heuristically pruned. The first approach requires more memory and more time to search for the best placement of a core at runtime but guarantees that, if a feasible location exists then it will be found. The second approach does not give this guarantee but, on the other hand, requires significantly less memory and can give a critical speedup in the placement algorithm running time, which must be as fast as possible to avoid hindering the performances of the dynamically reconfigurable system.

The last issue concerns the definition, evaluation and application of different fitting strategies to obtain good results, especially in terms of performance; a fitting strategy is the policy used to choose where to place a core among the possibly many acceptable solutions provided by
the empty space manager. As described in the previous chapters, there are several criteria, but the main subdivision is between policies that perform a choice without directly trying to minimize any particular objective function but with simple considerations on height, width or position and policies that choose among the available locations the one that directly maximizes or minimizes some specific objective function. Some examples of general fitting strategies are First Fit or Best Fit; on the other hand, focused strategies can, for example, try to directly minimize routing costs or fragmentation.

In the previous chapter, various possibilities for combined core allocation management have been explored; the most relevant literature solutions for empty space management have been analyzed and compared to obtain information on their relative advantages and disadvantages and a large number of possibilities for a fitting strategy have been considered.

The definition of our novel combined solution for core allocation management was driven by the comparison of the different approaches and by previous analysis on the specific problem of 2D Self Partial and Dynamical Reconfiguration towards a specific direction. The chosen approach for the newly proposed solution is to exploit the low complexity of an heuristic empty space management technique, inspired by the previous KNER solution, but, at the same time, keep reasonable quality and high versatility in our solution, being able to integrate, if needed, various fitting strategies.

The aim is to obtain a solution that is significantly faster than those that exploit complete information on the empty space and still obtains reasonably good results according to different metrics or, equivalently, keeping a good overall placement quality.
4.3.1 Proposed Solution for Core Allocation Management

The proposed solution to handle core allocation in a 2D self, partially and dynamically reconfigurable system is composed of a heuristic, KNER-like (33), empty space manager plus a multi-purpose fitter.

The choice of a heuristic allocation manager was driven by the fact our target for this work is self reconfiguration; this means that the solution is going to be run on the reconfigurable device hardware, impacting on its performance. Thus we chose to limit this negative impact by using a low-complexity algorithm.

Again, it is important to note that our solution also has the goal of keeping high versatility for what concerns fitting policies strategies. In particular, the main structure of the algorithm is completely independent from the fitting strategy. Different criteria, both general and focused, can be integrated in our allocation manager with some small changes; in the following we will keep the description of our solution as general as possible, explicitly stating which parts have to be changed to integrate a specific strategy.

An additional improvement that we include in the proposed solution is support for multiple shapes; by removing the restriction that the shape of each core must be fixed, the chance of actually being able to place it increases. The price for this improvement lies in the greater amount of configuration bitstreams that must be stored inside the reconfigurable system: for example if we allow an average of two shapes per core, the number of configuration bitstreams doubles too. However, we can counterbalance this drawback by exploiting the relocation technique; even if the amount of bitstreams is two-three times larger than that in a single shape scenario,
relocation reduces the number of originally needed bitstreams by tenths of times, yielding still a significant overall reduction in memory usage with respect to a non-relocation and single-shape scenario.
CHAPTER 5

IMPLEMENTATION

This chapter presents the implementation of the parts of the flow that are directly involved in this thesis. In particular, two components of the Solution Identification phase: Area Constraints Definition and Core Allocation Management are described in detail.

5.1 Area Constraints Definition

The main goal of the proposed workflow is to automate the creation process of complex self-reconfigurable architectures; however, we also aim to keep high flexibility, giving the designer the possibility of directly interact with the flow to guide or constraint the search for a suitable solution. As the following section shows, the Area Constraints Definition phase is a good example of this.

5.1.1 User Preferences Management

The flow supports both completely autonomous definition of constraints for the given set of cores and partial intervention from the designer; on the extreme, the designer can even specify all the constraints, if desired. The way in which this is managed is the following: when the application is defined by the user, as one of the inputs, she can, for each Core, either specify a shape or not. In addition to that, the maximum number of system defined shapes per Core can be specified, either as a global parameter or for each individual Core; another parameter that can be set is the tightness of the system generated constraints, this option represents the desired
balance between area and performance (and also risk of unfeasibility of the implementation, in the extreme).

In is important to note that, to verify whether the implementation of a Core with a given constraint is feasible, it is needed to actually synthesize it. As synthesis is an expensive process, we want to keep a low risk of failure for it and, thus, the default value for tightness will leave some margin to reasonably ensure feasibility, this at the price of some wasted area. The default value is empirically estimated to guarantee a near-certain feasibility. If the user is willing to improve the area usage, by increasing the risk of unfeasibility, and thus the wasted time needed to repeat the synthesis process until feasibility is reached, she can alter the tightness factor towards area.

5.1.2 Constraints Automatic Generation and Alteration

During the Solution Identification phase, Area Constraints are iteratively altered, using feedback from the simulation phase. The constraints given by the user are never changed, as we consider them to be strict requirements; however, if they are unfeasible, the flow will alert the user and ask for them to be changed (also providing a suggested value). In the first iteration, one shape per each Core is needed; in the case of user-defined constraints, that shape is used. Otherwise, a first heuristic implementation is defined with a criterion based on achieving a square-like shape that keeps relatively low area usage but, at the same time, near-certain feasibility. The formulae that are used to achieve this are:

\[ H = \lceil \sqrt{S \times (1 + m)} \rceil \]
\[ W = \left\lfloor \frac{S \times (1 + m)}{H} \right\rfloor \]

Where \( H \) and \( W \) are height and width of the constraint in slices, \( S \) is the number of slices of the Core and \( m \) is the margin that is left to keep a low risk of unfeasibility (an example value for this can be 0.15, which means 15% leftover area). The way \( H \) and \( W \) are used in the formulae is a mere convention and they can be arbitrarily swapped. This first set of square-like constraints is then used to define the slot size and number for the reconfigurable system, unless already specified by the user; the formulae used to compute those values are the following:

\[
\text{Rows} = \left\lfloor \frac{Vg \times V\text{Slices}}{\text{avgH}} \right\rfloor
\]

\[
\text{Cols} = \left\lfloor \frac{Hg \times H\text{Slices}}{\text{avgW}} \right\rfloor
\]

\[
\text{SlotH} = \left\lfloor \frac{V\text{slices}}{\text{Rows}} \right\rfloor
\]

\[
\text{SlotW} = \left\lfloor \frac{H\text{Slices}}{\text{Cols}} \right\rfloor
\]

Where \( \text{Rows} \) and \( \text{Cols} \) are the number of vertical and horizontal slots that will be created, \( V\text{Slices} \) and \( H\text{Slices} \) are the height and width of the FPGA in slices, \( \text{avgH} \) and \( \text{avgW} \) are the average height and width of the Cores in slices (which are available at this point, either because given by the user or computed using the previous formulae); \( Vg \) and \( Hg \) are user defined integer parameters that express Vertical and Horizontal granularity (the higher the values the smaller
slots will tend to be, for example if $V_g$ is 1 and $H_g$ is 3 the average core will span 1 row and 3 columns) and, finally, $SlotH$ and $SlotW$ are the height and width of a single slot in slices.

In the subsequent iterations of the Solution Identification phase, the constraints are altered. Other shapes are computed for cores that encountered significant difficulty of placement in the simulation; this can happen in two ways: if the user allowed more than one implementation for that core, new shape options are added to that core, if only one implementation is allowed the existing one is altered (if it was not defined by the user). The alteration of the shapes is performed by either increasing the height of the core by 1 slot at a time and adapting the width or the opposite; if more implementations can be kept, both routes can be taken at the same time. At the end of the simulation, the set of area constraints that gave the best overall results is kept and passed to the subsequent phase.

5.2 Core Allocation Management

This section describes the developed solution for online core allocation management. It is built upon a KNER-like (33) efficient empty space manager that keeps a low amount of information on the device state to reduce memory usage and search time. The reason for this is because the allocation manager is going to be executed on the reconfigurable device hardware, adding an overhead to its runtime, thus we chose to limit this negative impact by using a low-complexity solution. The empty space manager is also designed to be flexible and to support different scenarios and various kinds of customized placement policies. In particular, the scenarios we consider include: dynamic schedule scenario, blind schedule scenario and variants; placement policies, instead can be either general or focused.
5.2.1 Supported Scenarios

In a dynamic schedule scenario Core allocation requests arrive at an unpredictable time and with an ASAP and ALAP time to be satisfied; the core allocation manager must then try to satisfy all the requests respecting the timing constraints. In this scenario, Core allocation failures are a possibility and will be measured by computing the Core Rejection Rate at the end of each simulation. Since the final system should not have such failures, a CRR greater than 0% implies that another iteration has to be done to improve the results; if, after the possibilities for improvement are exhausted, the CRR is still nonzero, then the only solution to the problem is to prompt the user to increase the reconfigurable device size, since the given resources are not enough to meet the requirements.

In a blind schedule scenario, Cores are assumed to not have dependencies between them; the manager has a set of Cores to be placed for execution and the goal of finding a location for each one of them as fast as possible. In this scenario, failures are not a possibility: if a Core is not successfully placed at time $t$ it is put in a queue and retried later. Eventually, all the cores are placed and finish execution and the application is completed. The quality measure for this scenario, then, is the total time required to complete the application with the actual resources versus an infinite resoures scenario. Variants are also allowed, for example a blind schedule scenario with variable arrival times for Cores, or deadlines for some of the Cores, or both. When including deadlines, again the CRR becomes a useful measure along with completion time. The scenario that is considered can be inferred by the system by examining the application specification that is given by the user.
5.2.2 Cores Representation

Our allocation manager makes use of several data structures to represent the state of the system and handle the placement and deallocation of functionalities. The following paragraphs provide a description of those structures.

For what concerns timing, Cores are defined by Arrival time, which is the time instant when they are received by the manager and a set of Latencies, one per each shape allowed. In the blind schedule scenario, the Arrival times can be either all set to zero, which means the whole application is available from start or can be differentiated; moreover, an optional deadline can be specified for each Core. In the dynamic schedule scenario, an ASAP and an ALAP time for each Core are used, instead; ASAP represents the first time instant from when the Core can be placed on the reconfigurable device, while ALAP is used to determine whether it is too late to place the Core (placement failure) or not.

The physical features that define a Core are its Height and Width, respectively in rows and columns and defined by the chosen vertical/horizontal reconfiguration granularity for the target device. When multiple shapes are allowed, as in the case of the MSLP version of our solution (Multiple Shapes Linear Placer), a set of Heights and Widths is used instead. In the case of a placement policy focused on routing costs minimization, we also need the information on the set of Cores that need to communicate with the current one, which will be used to compute the value for the distance metric to be minimized; this is necessary in the RALP version of our solution (Routing Aware Linear Placer). Those two special versions will be of particular
relevance in the next chapter, which illustrates the results achieved by them along with other, more general, results.

A queue structure is used to host arrived Cores and subsequently navigate through them to manage allocation and deallocation. An additional queue structure is used to keep Cores that were not successfully placed but whose deadline, if any, (or ALAP in the case of dynamic schedule) has still not expired; this queue is used to retry their placement at subsequent time steps, when the device may potentially be less saturated. Other features that are set by the allocation manager for each Core are the StartTime, when the Core started computation, the EndTime, when the Core will finish computation, and the Rectangle where the Core is placed, which represents its location on the device.

5.2.3 Reconfigurable Device State Representation

The reconfigurable device state is described by a Binary Tree: each internal node represents an occupied Rectangle in which a Core has been placed, while each leaf represents a free Rectangle. When looking for a suitable location to place a Core, only the leaves have to be traversed; the first leaf of the tree is accessed through a firstLeaf pointer which is maintained up-to-date during evolution of the system.

Navigation in the tree takes place by means of a leftChild and rightChild pointer for each node, plus a nextLeaf pointer for each leaf, used to quickly traverse them. A previousLeaf function is defined to be used in the bookkeeping of the tree after merge and split operations, which are defined in the next section. An additional pointer is not used for this but, instead, we provide a function that computes the previous leaf by exploring the tree; this is convenient
because that pointer is rarely needed for bookkeeping and omitting it significantly reduces the size of the tree.

As a side note, separate partitions of the reconfigurable area can be represented by several Binary Trees, each one of them with the root located in a different cartesian position. This extends the generality of our approach to systems where the reconfigurable area is composed of physically disjoint parts, like a multi-fpga scenario and can also account to the case of non-homogeneity. A non homogeneous device can be subdivided in homogeneous subparts, each one of them can then be considered as a different FPGA and managed accordingly.

5.2.4 Rectangles, Split operation, Merge operation

The Binary Tree that represents the device state is made up by Rectangles; each rectangle is defined by a (x,y) position (that of its lower left corner in the device), its height and its width. When a Core is placed inside a Rectangle, conventionally at the bottom left of it, the latter is split and up to two new leaves are instantiated as its children: the left child represents remaining free space over the Core, the right child represents remaining free space to the right of the Core; then, bookkeeping must be done to adjust pointers to nextLeaf. Conversely, when a Core is removed from the device (from a Rectangle) a merge has to be done. The merge operation is more complex: in fact, in case of merge three different situations may occur:

- both the children are leaves: in this case we must resize the rectangle, remove both those leaves and adjust nextLeaf pointers of the other leaves in the tree where necessary.
• leftChild is a leaf, rightChild is not: in this situation, we have to merge the rectangle with its left child; this operation is actually implemented by resizing the leftChild. This case is the simplest one, as no bookkeeping of tree pointers is needed.

• leftChild is not a leaf: this is the most complex case; its handling involves instantiating a new internal node between the Rectangle and its leftChild and creating a new leaf with the shape and coordinates of the removed Core, as rightChild of the newly added node. After that, nextLeaf pointers in the tree have to be adjusted for neighboring nodes; this is the case that actually requires the previousLeaf function.

5.2.5 The Algorithms

In this section the algorithms for placement management based on the previous definitions are presented; as previously stated the proposed solutions are designed to manage the placement of a Core in an amount of time which is linear in the number of currently placed Cores.

The basic idea behind our placement management approach is to maintain and manipulate a Binary Tree that represents the state of the reconfigurable device and, in particular, provides access to a list of empty rectangles, which are its leaves; those leaves are processed to find the best one for placement in terms of routing cost. A general Data Flow Chart for the algorithm is shown in Figure 11.

As can be seen from the picture, the first operation that occurs is the loading of new Cores in the queue. After this, the algorithms check for previously placed Cores that have completed their execution; if any completed Core is found, deallocation takes place: the FPGA state is
Figure 11. General Data Flow Chart for the Proposed Solution
updated by merging, adding or reshaping involved Rectangles and the Tree is ready to host new Cores.

After this phase, the algorithms examine the first core in either one of the queues, in a dynamic schedule scenario it is also checked whether the ASAP time was reached before considering it for placement. If none of the queues holds a Core that can be considered for placement the algorithms loop back to core loading and deallocation. Otherwise, the extracted Core is processed for placement. The way the solution works implies that, for each iteration of the algorithm, at most one Core is deallocated and at most one Core is allocated. Performing deallocation prior to allocation is more convenient that the reverse, as it permits to immediately give a placement location to a new Core in cases where device area is nearly filled but the new Core can fit in the space left free by the deallocated Core; if we chose the reverse order, that new Core would have been placed in the subsequent iteration.

When a Core is processed for placement, the first leaf is extracted from the tree, in constant time, and examined to see if the Core can fit it; if so, the chosen criterion is used to compute a goodness measure. For example, if we apply a routing path minimization strategy, the distance metric is used to compute the sum of distances between the Rectangle and all the placed Cores that interact with the candidate one. After this computation, if the candidate leaf is better than those previously examined (for the Routing Aware example, if it has a lower routing paths length), then the current best is updated and the algorithm proceeds to the next leaf.

When the last leaf examination is finished, either a placement solution has been found or not. In the first case, the Core is placed in the chosen location and necessary tree bookkeeping
is performed: the chosen Rectangle is split into up to two new leaves and the nextLeaf pointers in the tree are updated to permit subsequent navigation. In the second case, the core is not successfully placed (i.e. no large enough location has been found). When a Core is not successfully placed, different cases can occur. In a blind schedule scenario with no deadline for that Core or with deadline not expired, it is always inserted in the additional queue; in a blind schedule scenario with expired deadline the core is rejected, increasing the Core Rejection Rate (CRR). Similarly, in a dynamic schedule scenario with ALAP not expired the Core is inserted in the additional queue; if the ALAP time has expired, instead, the Core is, again, rejected, increasing the CRR.

A more precise description of how the proposed solutions work is provided in form of pseudocode. In particular, we present the code for two versions, one for the blind schedule scenario (with variable arrivals and no deadlines, Algorithm 1) and one for the dynamic schedule scenario, Algorithm 2. Both solutions use the same procedure to find the best placement location, illustrated in Algorithm 3.

It is important to note that, in the dynamic schedule scenario, Re-Examination of the Cores that were not successfully placed via the additional queue, does not necessarily have a positive impact on global CRR. The basic idea to do not immediately reject cores but to insert them in the second queue with an ASAP time equal to their ALAP; in this way, they can be re-checked for placement later and, hopefully, some area will be freed meanwhile. The drawback lies in the fact that giving two placement chances to a core could help avoid rejecting it but, at the
Algorithm 1: Empty Space Manager for Blind Schedule Scenario

repeat
  if (a Core C arrived) then
    add C to Queue1;
  end
  if (a placed Core C finished) then
    deallocate C and do tree bookkeeping;
  end
  if (QueueA not empty) then
    C ← first Core in Queue1;
  else if (QueueB not empty) then
    C ← first Core in Queue2;
  else
    continue;
  end
  findPlacement (C);
  if (returned value ≠ failure) then
    allocate C and do tree bookkeeping;
  else
    add C to Queue2;
  end
until (no more Cores in the application) ;
**Algorithm 2**: Empty Space Manager for Dynamic Schedule Scenario

```plaintext
repeat
  repeat
    if (a Core C arrived) then
      add C to Queue1;
    end
    if (a placed Core C finished) then
      deallocate C and do tree bookkeeping;
    end
  until ((a Core C in Queue1 reaches ASAP) OR (a Core C in Queue2 reaches ALAP));
  findPlacement (C);
  if (returned value ≠ failure) then
    allocate C and do tree bookkeeping;
  else if (current time ≥ ALAP (C)) then
    increase CRR;
  else
    add C to Queue2;
  end
until (no more Cores in the application):
```
Algorithm 3: Procedure findPlacement(Core C)

1. \( L \leftarrow \) first leaf from tree;
2. repeat
3. \( \text{if } C \text{ fits in } L \text{ then} \)
4. \( \quad \text{if goodness } (C,L) > \text{currentBest then} \)
5. \( \quad \quad \text{currentBest } \leftarrow \text{goodness } (C,L); \)
6. \( \quad \text{end} \)
7. \( \text{end} \)
8. \( L \leftarrow \) next leaf from tree;
9. until no more leaves ;
10. if suitable location \( L \) was found then
11. \( \quad \text{return } L; \)
12. else
13. \( \quad \text{return } \text{failure}; \)
14. end

same time, could prevent the placement of one or more subsequent cores, removing the benefit of the improvement.

In either case of successful and unsuccessful placement, at the end the algorithms loop back to the first step, checking whether new cores have been received and a new deallocation/allocation cycle can start.

As can be inferred from the pseudocodes, the algorithms succeed in completely managing the placement of a core in an amount of time which is linear in the number of currently placed cores.
CHAPTER 6

EXPERIMENTAL RESULTS

This chapter presents the experimental results obtained by the proposed placement solutions. The results that were gathered on the performance of the solutions are organized in three main experiments. The first scenario considers a dynamic schedule setting, with a routing aware fitting strategy, and compares the results of our solution with those of closely related literature solutions on a randomly generated benchmark application. With this experiment, we directly gather results on the CRR, routing costs effectiveness and management overhead of our solution and indirectly obtain results on fragmentation, which can be inferred from the CRR. This version of our Core Allocation Management Solution and the experiment that is outlined in the following are also described in (44). In the second experiment, the completion time of a benchmark application consisting of real world cores when placed by our method on a reconfigurable device of limited size are compared with the times obtained on larger FPGAs and also versus an infinite resources scenario. With this experiment, direct results are obtained for what concerns application completion time; some insight on CRR and fragmentation can also be obtained by looking at the ratio between real-world scenarios and infinite resources scenario (which, by definition, has no fragmentation and thus zero CRR). Finally, the third scenario provides insight on the effectiveness of allowing multiple shapes for cores, comparing the result obtained by our solution against that of the KNER (33) approach, which is similar
but does not exploit multiple shapes. With this final experiment, we directly measure CRR, from which we can obtain information on fragmentation, and management overhead.

6.1 Comparison with similar literature solutions

Table II shows the compared results obtained by the proposed solution integrated with a routing cost minimization fitting policy (in the second row, RALP - Routing Aware Linear Placer) against the two that are most similar to it: CUR (40), which shares the fitting strategy based on minimizing routing paths but is not heuristic, and KNER (33) which follows the same philosophy as RALP in empty space management but does not apply any focused fitting strategy.

The benchmark used to perform this first evaluation is a set of 100 randomly generated Cores, with size varying from roughly 5% of the reconfigurable device area to approximately 25% of it; the required interconnections were also randomly generated along with the Cores and the routing cost obtained by each solution was subsequently computed with manhattan distance metric. The Core Rejection Rate reflects the percentage of all the Cores that were not successfully placed due to suitable area not found by the allocation manager. The benchmark was run several times with different random sets and the results presented in the table reflect the average of the obtained results.

The table shows the considered algorithms in decreasing complexity order. CUR, being the most complex, has a running time that is about three times that of the other two; on the other hand RALP, the proposed algorithm, being more complex than KNER only by a constant factor, obtained a running time which is only slightly more than the fastest solution.
The difference in running time between RALP and CUR should also significantly increase when considering a schedule with many more cores, given their asymptotical difference.

For what concerns routing paths cost, the best results were obtained by CUR, which is not surprising as this solution was explicitly designed with the goal of minimizing routing paths; RALP placement, also routing focused but simpler, caused slightly more expensive interconnections but still three times better than those of an unfocused solution like KNER. It is interesting to note that the additional computation necessary to focus on minimizing routing paths is rewarded by a total interconnection cost which is three to four times less for this benchmark and that the difference should increase with bigger examples with more interconnections; this result suggests that a routing aware fitting strategy is actually useful to improve the performance of a reconfigurable system.

Finally, the three Core Rejection Rates show how, in general, the focus on routing paths minimization indirectly gives a negative impact on CRR. This is due to the fact that placing

### TABLE II

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Running Time (per Core)</th>
<th>Routing Cost (average)</th>
<th>CRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUR</td>
<td>Complete, Routing Focused</td>
<td>156 ms (1.56 ms)</td>
<td>67 (2.7)</td>
<td>5%</td>
</tr>
<tr>
<td>RALP</td>
<td>Heuristic, Routing Focused</td>
<td>51 ms (0.506 ms)</td>
<td>94 (3.6)</td>
<td>4.7%</td>
</tr>
<tr>
<td>KNER</td>
<td>Heuristic, Unfocused (FF)</td>
<td>39 ms (0.386 ms)</td>
<td>285 (11.4)</td>
<td>4.2%</td>
</tr>
</tbody>
</table>
cores to minimize their distance from interacting ones does nothing to guarantee a good fragmentation state of the device; on the other hand, simple unfocused strategies like BF, or even FF, tend to naturally cluster cores to leave free the more contiguous space possible, with a positive impact on CRR. For this reason the CRR of KNER is better than that of the other two solutions.

However, it should also be noted that the benchmark considers a high core density situation, where most of the placement failures are due to lack of reconfigurable resources more than to bad placement choices. This was observed by performing additional runs of the algorithm with a slightly modified benchmark, or a slightly bigger FPGA. The 0% CRR for RALP (and approximately also for the other two which have a similar average CRR) was reached when either increasing FPGA size by 10% or average time between the arrival of two Cores by about 20%. With lower core density or a slightly bigger device, all the three CRRs rapidly go to zero, which is what would be expected for a placement solution. When considering this, the slightly worse CRR of RALP with respect to KAMER can easily become irrelevant in a real system, while efficiency in running time and routing costs still remains relevant.

6.2 Completion times when varying device size

This section presents a comparison of application completion time with different amount of resources and different Core densities. The scenario that is considered here is a blind schedule one, with variable arrival times for cores. The measure that is computed is the number of time instants that pass from the arrival of the first Core to the completion of the one that was placed last; each core has an execution time, expressed in time instants and labeled latency.
This experiment is important to measure the performance of a system using our proposed solution when no prior information on the schedule that will be requested is available; this is relevant in a scenario where functionalities are highly independent one from the other and the main concern for the designer is to achieve an high throughput of the whole application, with no particular constraint on the completion order of tasks.

The Cores that are used for this experiment are real world ones, taken from OpenCores.org and whose area constraints have been defined automatically, using the method described in the previous Chapter. In particular, the cores that have been used as benchmark include common applications such as 3DES, AES, JPEG, Cordic and a floating point unit. The benchmark applications have been built by combining multiple instances of those cores, with variable number of cycles required, reflecting different usages and with shapes defined using our automated solution.

The time that passes between the arrival of two cores varies randomly between zero and two time instants while the global density of the application is defined by the average latency; given constant inter-arrival time, a higher latency implies a more dense application; the average latencies that we considered have the values of 10, 20 and 30 time instants. On the other hand, application complexity can be represented using different number of cores; in particular, we considered three scenarios, one with 20, one with 30 and one with 50 cores. The placement policy that is applied in those scenarios is a simple First Fit.

The available resources are represented by the device size: the larger the device, the more resources are available. The extreme case is the infinite resources scenario, which basically
TABLE III
COMPLETION TIMES WHEN VARYING DEVICE SIZE

<table>
<thead>
<tr>
<th>Application Size</th>
<th>Average Latency</th>
<th>4x8 Device (ratio vs inf)</th>
<th>4x10 Device (ratio vs inf)</th>
<th>4x14 Device (ratio vs inf)</th>
<th>Infinite Resources taken as (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 Cores</td>
<td>10</td>
<td>74 (2,1)</td>
<td>51 (1,5)</td>
<td>47 (1,3)</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>106 (2,1)</td>
<td>97 (1,9)</td>
<td>89 (1,6)</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>167 (2,7)</td>
<td>160 (2,6)</td>
<td>75 (1,5)</td>
<td>61</td>
</tr>
<tr>
<td>30 Cores</td>
<td>10</td>
<td>108 (3,3)</td>
<td>85 (2,6)</td>
<td>53 (1,6)</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>144 (2,8)</td>
<td>112 (2,2)</td>
<td>109 (1,8)</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>198 (3,4)</td>
<td>164 (2,8)</td>
<td>90 (1,8)</td>
<td>58</td>
</tr>
<tr>
<td>50 Cores</td>
<td>10</td>
<td>138 (2,9)</td>
<td>120 (2,6)</td>
<td>110 (2,3)</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>225 (3,4)</td>
<td>187 (2,8)</td>
<td>152 (2,3)</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>332 (4,0)</td>
<td>281 (3,4)</td>
<td>214 (2,6)</td>
<td>83</td>
</tr>
</tbody>
</table>

assumes an infinite device and can be useful as a term of comparison for the real scenarios. The device sizes that were considered are, in slots, of 4 rows by respectively 8, 10 and 14 columns. The slot size, defined again using our automated approach based on the distribution of core sizes in the application is of 45 slice (height) by 20 slice (width). To give an example, with those definitions, a JPEG core can fit in a 3 columns by 2 rows rectangle, while an AES can fit in 1 row and 2 columns.

Table III shows the results obtained in the outlined scenarios by our proposed solution for Core Allocation Management, in its version tailored to a blind schedule case. Since the infinite resources value is basically independent on the device, the ratio among that and the other,
real-world, cases can also be used to gain insight on the quality obtained also by our Area Constraints Definition method.

The immediate consideration that can be made on the results is that the real-world scenario times are in general quite larger than those of the infinite resources one; in fact those values are, on average, more than two times larger. This is reasonable, since the amount of parallelization that can be achieved with a finite, amount of resources is limited while in the infinite resources case, all the available cores can immediately be configured and run. However, in some cases, such as the 20 Cores with average latency 10 and 30, the result obtained with a finite device (4x14) is quite close to the ideal value; this means that, for that specific application, the given amount of resources can be considered enough to achieve an extremely high level of parallelism.

If we restrict ourselves to the real-world scenarios, it is also interesting to note that, in many cases, an increase in device size does not imply a significant improvement in completion time or, equivalently, in the level of parallelism. Taking for example the 30 Cores and average latency 20 case, passing from a 4x10 to a 4x14 device does yields only a 5% improvement in completion time at the cost of a 40% increase in available area. Similar considerations can be done for the 20 Cores, average latency 30 scenario. Moreover, it can also be observed how, in general, an increase in the application density, represented by the average latency, has a significant negative impact on the ratio between completion time with finite and infinite resources; this seems also to be true for what concerns application complexity, which is reflected by the number of Cores.

As a final consideration, it is interesting to note that the proposed solution can be also used to define a suitable size for the device that is going to implement the reconfigurable
system; this can be done by performing a similar experiment with the desired application, or a benchmark that is representative of a family of desired applications, and selecting the configuration that gives an acceptable tradeoff between cost of the reconfigurable device and amount of parallelization that can be achieved.

6.3 Comparison between multiple-shapes and single-shape

To evaluate the benefit of allowing multiple shapes per Core in a Core Allocation Management solution, this third experiment was developed. The experiment compares the results obtained by KNER (33), a literature solution for fast core placement that only allows a single fixed shape for each Core, with those obtained by our solution, here called MSLP - Multiple Shapes Linear Placer, with 3 and 5 shapes per Core. The goal of the experiment is to demonstrate that the cost increase, in term of number of bitstreams to be stored in the system and running time of the algorithm, is adequately rewarded by a significant improvement in placement quality, measured as Core Rejection Rate.

The scenario we are considering here is, similarly to the previous one, a blind schedule one, with no dependances between Cores. However we introduce a difference: in this experiment a deadline is associated to each Core, if the Core is not successfully placed before its expiration it must be rejected; this permits to actually compute a CRR value for the application. Except for this modification, the way the benchmark sets for this scenario are built is quite similar. The used Cores are real world ones, taken from OpenCores.org, with area constraints (and thus shapes) defined using our automated solution; instances of those Cores, with variable parameters such as number of cycles, are combined to create applications of different size and densities.
Application size is the number of Cores that compose the application, here we considered 50 and 100 Cores, while the density is a measure of how many Cores request placement at a time; again, we consider an average inter-arrival time that does not vary between experiments, while an increased average latency is used to obtain higher density scenarios, the average latencies are 10, 20 and 30 time instants. For this experiment we considered two different placement policies, a simple First Fit one and a slightly more complex Best Fit, aiming to minimize leftover area around a newly placed Core.

**TABLE IV**

**SINGLE SHAPE VS MULTIPLE SHAPES USING FIRST FIT**

<table>
<thead>
<tr>
<th>Application Size</th>
<th>Average Latency</th>
<th>KNER (1 Shape)</th>
<th>MSLP (3 Shapes)</th>
<th>MSLP (5 Shapes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runtime</td>
<td>CRR</td>
<td>Runtime</td>
<td>CRR</td>
</tr>
<tr>
<td>50 Cores 10</td>
<td>0.261</td>
<td>2.2%</td>
<td>0.296</td>
<td>0.8%</td>
</tr>
<tr>
<td>50 Cores 20</td>
<td>0.273</td>
<td>3.8%</td>
<td>0.3</td>
<td>2.1%</td>
</tr>
<tr>
<td>50 Cores 30</td>
<td>0.262</td>
<td>6.0%</td>
<td>0.298</td>
<td>3.6%</td>
</tr>
<tr>
<td>100 Cores 10</td>
<td>0.28</td>
<td>2.3%</td>
<td>0.304</td>
<td>0.9%</td>
</tr>
<tr>
<td>100 Cores 20</td>
<td>0.276</td>
<td>3.8%</td>
<td>0.303</td>
<td>2.2%</td>
</tr>
<tr>
<td>100 Cores 30</td>
<td>0.285</td>
<td>6.7%</td>
<td>0.311</td>
<td>3.4%</td>
</tr>
</tbody>
</table>

Table IV and Table V show the results obtained by KNER, that uses a single shape, and our multiple shapes solution, with 3 and 5 shapes per Core (labelled MSLP) on different benchmark sets built using the rules explained in the previous paragraph. In particular the results in
Table IV are obtained using a First Fit placement policy while those of Table V are obtained using a Best Fit policy.

The first consideration that can be done on the results is that increasing the number of shapes considered for placement seems to have a significant effect in reducing the CRR; in most of the cases the CRR is halved and sometimes the improvement is even better. The improvement in CRR was obviously expected, given the fact we are increasing the number of possibilities for placement of each Core, but the observed effect is significantly better than the expectations. Another consideration that can be done is that the improvement obtained passing from a traditional, single shape solution to a 3 shapes one is vastly superior than the one obtained by further increasing the number of shapes to 5. This is reasonable, given the way additional shapes are defined: the first one is square-like, the second and the third are obtained distorting the Core either horizontally or vertically, obtaining a rectangular shape, the last two

<table>
<thead>
<tr>
<th>Application Size</th>
<th>Average Latency</th>
<th>KNER (1 Shape)</th>
<th>MSLP (3 Shapes)</th>
<th>MSLP (5 Shapes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 Cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.295</td>
<td>2.0%</td>
<td>0.322</td>
<td>0.6%</td>
</tr>
<tr>
<td>20</td>
<td>0.292</td>
<td>3.6%</td>
<td>0.319</td>
<td>1.4%</td>
</tr>
<tr>
<td>30</td>
<td>0.298</td>
<td>5.7%</td>
<td>0.32</td>
<td>2.6%</td>
</tr>
<tr>
<td>100 Cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.299</td>
<td>2.2%</td>
<td>0.344</td>
<td>0.6%</td>
</tr>
<tr>
<td>20</td>
<td>0.307</td>
<td>3.5%</td>
<td>0.342</td>
<td>1.5%</td>
</tr>
<tr>
<td>30</td>
<td>0.31</td>
<td>6.6%</td>
<td>0.342</td>
<td>3.0%</td>
</tr>
</tbody>
</table>

Table V

SINGLE SHAPE VS MULTIPLE SHAPES USING BEST FIT
are obtained by further distorting the Core but, for Cores of small or average size, this can yield to shapes that are either too thin or too flat to have a significant chance in finding a new possibility for placement. On the other hand, for large Cores, the last two shapes that are found can actually be relevant; for this reason, the small improvement obtained when passing from three to five shapes can be associated to the largest Cores, which however represent a small percentage of the total.

For what concerns the costs of exploiting multiple shapes, the tables give information on the runtime of the different solutions. As would be expected, a higher number of shapes means a longer time in looking for a placement solution; the single shape solution is faster than the 3 shapes one that is faster than the 5 shapes one. However, the running time is only about 10%-15% longer when passing from one to three shapes and grows similarly when passing from three to five; given the huge reduction in CRR this small increase is probably highly rewarded by an improvement in the overall performance of the system. The other cost that emerges when allowing multiple shapes per each Core in a self reconfigurable system is the increase of the number of configuration bitstreams that must be stored and, thus, of internal memory that must be used; this increase can be trivially computed, as each additional shape requires an additional bitstream. Thus, the three shapes solution requires 3 times more internal memory to store configuration bitstreams while the five shapes requires 5 times more. This can be a high price but, depending on the specific situation, it can be worth using more memory to achieve a lower CRR, especially when considering that the other solution available to achieve such drastic reduction is to increase the device size, which is normally even more expensive. In
addition to that, it must be noted that the number of configuration bitstreams that need to be stored inside the system is reduced in our approach to self reconfiguration by tenths of times by exploiting the bitstreams relocation technique; even if their number is multiplied by three to exploit multiple shapes, the amount of memory they use could still be far from relevant.

Figure 12. Improvement in CRR with FF and BF

If we compare the two tables, we can also observe how, passing from First Fit to Best Fit, the gain in exploiting multiple shapes increases even more. Figure 12 permits to better notice this phenomenon. Passing from a First Fit to a Best Fit policy yields a small improvement to the single shape solution but, when exploiting multiple shapes, gives a huge decrease in CRR. This can be explained by considering the cases in which the Best Fit policy gives a good solution; the best case scenario for the policy is to find a location that has the exact same shape of the
Core that is going to be placed or, at least, is really close in size to that of the Core. Then, the possibility of trying multiple shapes for a Core that is going to be placed can greatly help finding a location that fits well leaving as few leftover area as possible, which means reducing fragmentation; a lower amount of fragmentation means an increased likelihood of finding an available location for subsequent Cores and, in the end, lower CRR. On the other hand, the use of multiple shapes combined with a First Fit policy, has a more limitate impact in reducing the CRR because it does not exploit this fragmentation reduction.
CHAPTER 7

CONCLUSIONS

The main goal of the presented work was to obtain a lightweight but versatile solution for core allocation management that can quickly provide the location for placement of a new core solution without sacrificing too much in terms of placement quality, in the form of Core Rejection Rate or application completion time.

In particular this solution was developed to be used inside a larger project, the complete workflow for automation of the creation and management processes for self reconfigurable architectures that is also briefly described in this thesis. The proposed Core Allocation Manager, however, is also valuable as a standalone solution to perform the online decisions on the placement of Cores in any user-defined reconfigurable platform and, therefore, has a more general value and is not limited to that specific scenario. Similar considerations can be made for the Area Constraints Definition solution that was also given; again, the solution was created to be used inside our automated workflow but with a general value as a method to quickly obtain sets of shapes for the Cores of a given application in different scenarios.

In the previous chapter a set of experiments to obtain results on the quality of our solutions was described. In particular, the first experiment gave compared results against those of literature solutions for a version of our core allocation manager that, as placement policy, has the goal of achieving low total interconnection costs between configured cores. Another set of results that were presented is related to Area Constraints Definition and Placement quality
measured in scenarios that differ in the amount of available resources (reconfigurable area); the metric that was used here is total application completion time and ratio versus infinite resources scenario. Finally, the third experiment gives insight on the effectiveness of exploiting multiple shapes per Core, all automatically defined with our method, to increase the chance of successful placement of Cores using out core allocation manager without increasing the device area and with minimal impact on the performance.

As the results show, the initial goal was met; the proposed solutions represent a good compromise between complexity and quality of the results; moreover, the core allocation manager is highly versatile, supporting various kinds of fitting strategies and, in the focused version, combines the runtime advantage of the heuristic approach with the interconnection cost minimization advantage offered by routing aware focus. Furthermore the core allocation algorithms are simple enough, both computationally and memory wise, to be run on the internal processor of reconfigurable systems; this is important and one of our main concerns in the first place because the scenario on this the work is based is self reconfiguration and, in that scenario, all the reconfiguration management choices and computations must be done inside the system.

Future work is directed towards the complete integration of the proposed solutions in the automated workflow that was briefly presented in this thesis. This integration will allow to exploit different solutions for Placement, Area Constraints Definition and Relocation to obtain a completely independent system that exploits 2D reconfiguration in a significantly automated way, greatly reducing the concerns of the designer.
CITED LITERATURE


15. *Steven A. Guccione and Delon Levi*, "JBits: A Java-Based Interface to FPGA", Proc. of
the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technology Conference (MAPLD), 2000.


29. **H. Kalte, M. Porrmann and U. Rckert**, ”System-on-programmable-chip approach enabling online fine-grained 1D placement”, The 18th International Parallel and Distributed Processing Symposium (IPDPS), 2004.


34. **A. ElFarag and H. El-Boghdadi and S. Shaheen**, ”Miss Ratio Improvement For Real-Time Applications Using Fragmentation-Aware Placement”, The 14th Reconfigurable Ar-


40. A. Ahmadiania and C. Bobda and S. P. Fekete and J. Teich and J. v.d. Veen, ”Opti-


44. Massimo Morandi and Marco Novati and Marco Domenico Santambrogio and Donatella Sciuto, "Core allocation and relocation management for a self dynamically reconfigurable architecture”, IEEE Computer Society Annual Symposium on VLSI ’08, 2008.
VITA

Full Name : Massimo Morandi

Place and date of birth : Busto Arsizio (VA), ITALY. October 21st 1984

Nationality : Italian

Address : via Friuli 9, 21012 Cassano Magnago (VA), ITALY

E-mail : morandimassimo@gmail.com

Bachelor’s degree : Politecnico di Milano, 2006, under the advising of Prof Donatella Sciuto.

Publications :

- Fabrizio Ferrandi, Massimo Morandi, Marco Novati, Marco D. Santambrogio, Donatella Sciuto, Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead. International Symposium on System-on-Chip, 2006
• Fabrizio Ferrandi, Simone Corbetta, Massimo Morandi, Marco Novati, Marco D. Santambrogio, Donatella Sciuto, *Two Novel Approaches to Online Partial Bitstream Relocation in a Dynamically Reconfigurable System*. IEEE Computer Society Annual Symposium on VLSI, 2007

• Massimo Morandi, Marco Novati, Marco D. Santambrogio, Donatella Sciuto, *Core allocation and relocation management for a self dynamically reconfigurable architecture*. IEEE Computer Society Annual Symposium on VLSI, 2008