1

Design Methodologies for Reconfigurable NoC–based Embedded Systems

CONTENTS

1.1 Introduction 3
1.2 The Proposed Design Flow 5
1.2.1 High Level Specification 7
1.2.2 Communication Infrastructures Generation 10
1.2.3 Mapping 11
Smart Exhaustive Algorithm 11
GA1ver 13
GA2ver 14
1.2.4 Routing 15
1.2.5 Evaluation and Selection 17
1.2.6 Placement 19
1.2.7 Reconfigurations Minimization 20
1.3 Related Work 24
1.4 Algorithm Performance Analysis 28
1.5 Real-World Case Study 29
1.6 Concluding remarks 33

Dynamic reconfiguration capabilities offered by modern FPGA devices can improve performance, flexibility and reusabilty of embedded systems. On the other hand, due to the increasing complexity of modern on–chip applications, different and heterogeneous modules are now embedded on the same platform, requiring a design–paradigm shift towards a communication–centric approach. In this work we propose a design flow that, relying on the automatic analysis of a set of applications that will run on a given FPGA platform, generates a reconfigurable interconnection infrastructure that maximizes the overall communication performance.

1.1 Introduction

As the logic density of FPGAs steadily grows at Moore's pace, design practices try to keep up with the increasing complexity, and possibly take advantage of it. One of the most natural trends is to raise the level of abstraction relying on already designed and engineered components (generally called Intellectual Properties, IPs), and instantiating and connecting them to team up towards the desired functionality. As the number of such components increases, the communication between them becomes more relevant, and the design of the infrastructure to support it more sensible. Naive approaches such as point-to-point ad hoc channels fail to scale, for at least two reasons: the quadratic growth with the number of endpoints does not affect only obvious resources such as area and routing time (and cleverness) of the synthesis algorithm, but also effort of the designers, who have to devise adaptors between components exposing different interfaces and having different protocols.

Inspiration is then sought where communication scenarios of comparing complexity have already been faced, namely in the field of communication networks. As a wide set of abstractions, tools, concepts and architectures become available to the designers, ecosystems on FPGAs are increasingly seen as Networks of interconnected components, even though on a single Chip (hence Networks–on–Chip, NoCs). The NoC architecture is a micro-network of interconnects integrated on a single chip; in particular, it consists of a set of switches organized in a specified topology (that may be regular, such as a mesh-grid or rings, or irregular) and its communication paradigm is based on packet-switching (similarly to the standard telecommunication networks).

To put more complexity into the picture comes the remarkable and peculiar ability of FPGAs of being dynamically reconfigured; this feature allows to change at runtime the set of applications (consisting of several computing components and of the underlying communication infrastructure connecting them) running on the system. In this scenario, let us consider a typical system in which different applications are executed at different instants of time: profiling such a system, it would be possible to determine its application *working points*, that is the sets of applications and computations that typically cluster together during execution. When the system is in any of these points, the computational cores are executing defined tasks. Now, it is clear that, for each such point, there exists an ideal communication infrastructure, that is the one (be it a Network–on–Chip with some topology and features, or a bus based system) that maximizes the goal functions of interest (the throughput, the inverse of energy consumption, etc.) for that particular working point. One of the things possible with dynamic reconfiguration would be to seek and locate all the working configurations of a system (let's call the set of such configurations $S = s_1, s_2, ..., s_n$), to design the ideal communication networks for

each $s \in S$, and then to reconfigure the infrastructure each time the system switches to a new point.

The only problem would be, of course, the overhead costs (energy and time) of reconfiguring the network. At the other extreme, one could exploit the information, gathered through profiling, to design a *one size fits all* compromise network, which would maximize all the goals on an average of the working points, weighted upon their relative mean execution time. Such a network would be a function of the whole set S, plus the information on the relative execution times ET of the different working points ($ET : S \mapsto [0-1], \sum_i ET(s_i) = 1$). Now, if we had some additional information on how the systems switches between working points (ex: every 100ms it typically runs s_1 for 30ms, then switches to s_3 for 40ms, then to s_5 for 30ms, and then back all over to s_1), it would be possible to think of a "third way": defining a set of communication structures, one for every working point, each suboptimal with respect to the corresponding point, but requiring less time to be reconfigured, and better than the compromise network. Being able to do this requires the ability to design networks lowering the overhead required to switch between them.

Main contribution of this work is an overall design flow for the reconfigurable NoC-based embedded systems implemented onto FPGA devices running a set of applications evolving in time and organized in working points. The design flow supports the designer from the initial specification of the system towards the definition of the partially-reconfigurable implementation to be synthesized by means of the Xilinx Partial Reconfiguration (PR) flow [25]; moreover the various design steps automate the exploration of the network design space in order to optimize the reconfiguration overhead, while keeping the performance as high as possible. As it will be discussed further on, the proposed flow represents a step forward w.r.t. the literature since it supports the designer in all the various design steps and, moreover, the proposed algorithms produce better results and achieve higher performance.

The text is structured as follows: Section 1.2 presents the flow for the generation of the set of reconfigurable NoCs, while Section 1.3 discusses the state of the art in NoC design flows, and Section 1.4 analyzes timing performance of the proposed design flow for reconfigurable NoC–based architectures with respect to the state of the art. Then, Section 1.5 puts the network generator to the test with some benchmarks, and finally, Section 1.6 draws some conclusions.

1.2 The Proposed Design Flow

The proposed approach aims at defining a complete design flow that automatically generates a NoC-based partially-reconfigurable system onto Xilinx

FPGA devices, starting from the specification of the set of applications that have to be deployed on the target platform. Since the applications that run on the target system change over time, goals of the proposed design flow are to find an optimized NoC for each set of applications that can be executed at the same time and to provide the designer with all the information needed in order to physically place all the components on the target reconfigurable device [6]. This information can be used to feed the Xilinx PR flow [25], which automatically generate all the complete and partial bitstreams to dynamically reconfigure at run-time the target FPGA device.

As described by the Xilinx PR flow ([25]), the target reconfigurable architecture can be considered as composed of two different portions: a *static part* and a *reconfigurable part* as shown in Figure 1.1. The reconfigurable part can be further divided into a matrix of *tiles*, each one representing a reconfigurable region of the device. Tiles are assumed to be homogeneous (w.r.t. their size, their shape and the amount of reconfigurable resources within a single tile). The communication channels between the static part and the reconfigurable one and among the reconfigurable regions can be established by using special hardware interconnection components called *proxy logic*, automatically placed by the PR synthesis tool along each edge of each *reconfigurable region*, in order to meet the constraints specified by [25]. The system that will be implemented on such architecture is modeled with a structural specification composed of a set of reconfigurable modules interconnected each other. As shown in Figure 1.1, there is a main controller, placed in the static part, that is devoted to the communications with the outside, the coordination of all the activities of the system and the reconfiguration of the reconfigurable part. Moreover, the system contains a set of reconfigurable modules, each one hosting a set of processing cores (e.g. processors, HW accelerators, memories, ...) together with the network interface with a time-multiplexed fashion; reconfigurable areas may also host the various switches of the NoC infrastructure. It is worth noting that in the presentation we will consider only the reconfigurable part, since it is the focus of the proposed optimization approach.

The design flow is composed by a set of consecutive steps shown in Figure 1.2. At the beginning the designer has to define an high level model based of the system to be deployed on the FPGA device; the model is based on graphs and captures the organization of the system's application in various working points to be loaded in different time instants. Then, the second phase aims at instantiating for each application working point a set of NoC communication infrastructures, and the next two steps at mapping and routing of the application working point on each communication infrastructure. Then, among the generated mapping and routing solutions, for each working point the optimal one is chosen according to a set of metrics. Finally, these solutions are placed on the FPGA device grid and integrated in order to minimize the reconfiguration time to pass from a working point to the next one. As shown in the figure, each step generates a set of intermediate solutions according to the specific analysis that is performed; such solutions are then used to feed

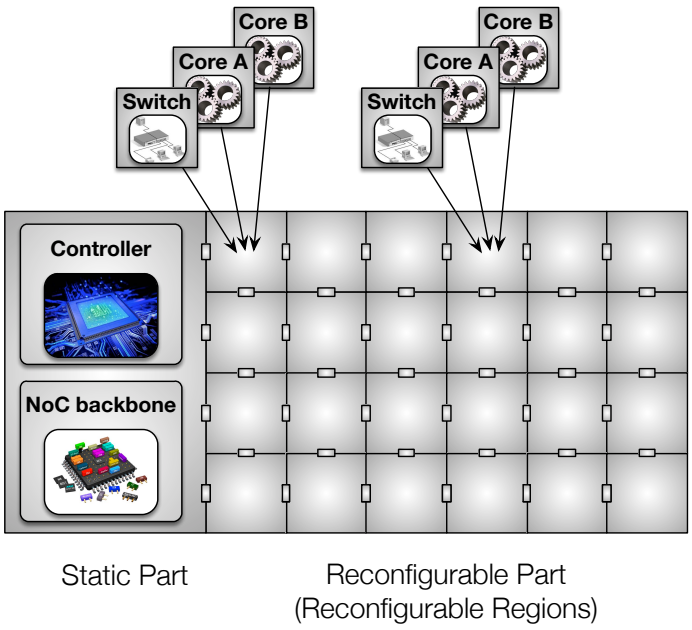

FIGURE 1.1
Target architecture schema with reconfigurable part divided into homogeneous tiles.

the subsequent step. During the various refinements performed by the steps, solutions previously identified may be discarded, and at the end of the flow, the output is a single solution representing the optimized implementation of the reconfigurable system.

The single steps are discussed in detail in the following sections, presenting the overall goals and the algorithms that have been defined to automate the optimization process. Moreover, a running example is used to exemplify the various performed activities. It is worth noting that one of the characteristics of the flow is its high modularity; therefore, if a more efficient implementation of any algorithm will be identified in the future, it can be integrated by simply replacing the current one.

1.2.1 High Level Specification

The first phase of the proposed flow is the definition of a high level model of the system specification in terms of a set of communication graphs (CGs).

The system specification received in input consists in a set of applications that have to be implemented in the FPGA device. Each application is specified in terms of a set of communicating master and slave processing cores. Usually,

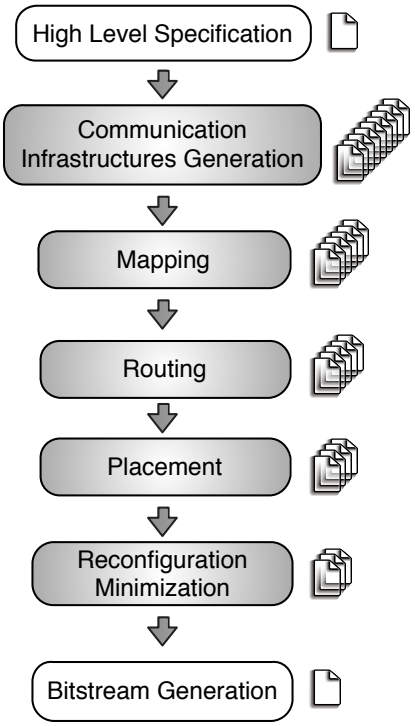

FIGURE 1.2
The proposed Design Flow schema

the applications are organized in a sequence of *working points*. In particular, each working point describes the set of applications loaded at the same time on the device (without exceeding the available resources) and in execution in a specific time period. Then, according to the specified schedule, or based on the input request, the system will evolve at runtime from a working point to another one. Figure 1.3 describes an example of sequence of three working points; each working point contains a set of unconnected communication graphs modeling the contained applications. It is worth noting that, the same application may appear in different working points as well as the same type of core may be required by different applications.

For each application working point a specific communication graph is defined, that models the connections among the cores of each application. Nodes in the graph represent the processing cores while the edges the communications between pairs of master and slave cores. The edges are characterized in terms of various parameters, such as the required communication throughput. For the purpose of this work, two main connection constraints have been

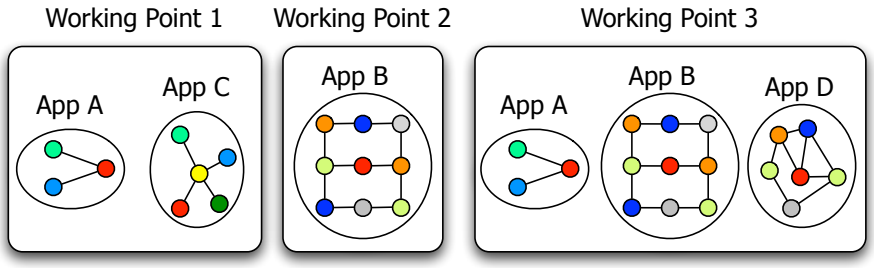

FIGURE 1.3
A sequence of three different working points

considered, namely the **latency value**, that specifies the maximum latency that the communication infrastructure can introduce in the communication between the cores, and the **throughput value**, that defines the throughput that the communication infrastructure has to support between the two connected cores (specified in terms of absolute values or with a *percentage* with respect to the total workload of the network). This last constraint will be used during the *routing phase* to find the path from the source core to the destination core that will ensure the required communication throughput. Moreover, the designer can set the *switch throughput upper bound*; this value represents the maximum throughput that each switch can handle. It will be used to discriminate feasible solutions during the *routing* phase.

This information can either be specified by the designer, or can be automatically obtained with some profiling techniques (as shown in the example in Section 1.5).

Running example

In order to better explain how the proposed framework works, let us consider the scenario in which two applications (A and B) have to be deployed at run–time on a reconfigurable target system in two different working points A and B. In particular, *application A* is an example of an application able to perform generic matrix computation over 3 different MicroBlaze processors (M0, M1 and M2) that work in parallel using 3 different memories (S0, S1 and S2). M0 acts as a *manager* of the other two processors. When M0 starts its execution, it reads data from its private memory (S0) and then spreads this information on two other memories, each one connected to a single MicroBlaze processor, so that each processor can read its information and work on its main memory. In order to perform this type of work, the overall system must ensure a specific throughput for each connection. Let us consider a *fast* communication link (high throughput and low latency) for each connection between a processor and its main memory, and a *medium* one (medium throughput and latency) for the connections between M0 and the two memories S1 and S2.

On the other hand, *Application B* is an example of a simple graphic ac-

celerator application that consists of 3 different MicroBlaze processors (M0, M1 and M2) and 5 memories (S0, S1, S2, S3, S4 and SHM). M0 is the *coordinator* that controls and delegates every task to the other processors M1 and M2. M0, M1 and M2 are connected to a shared memory (SHM) in which common data must be stored, and each communication link to this shared memory requires a *medium–high* throughput and a *medium* latency in order to ensure a good data communication. As for application A, M1 and M2 have their own main memory (S1 and S2) in common with the coordinator, and the corresponding communication link requires a *low* throughput due to the random accesses to this particular memory. Moreover, both M1 and M2 are supported with a dedicated memory (S3 and S4, respectively), connected with a *high* throughput and a *low* latency link.

During the first phase of the design flow, the *communication graphs* of application working points A and B are defined as shown in Figure 1.4. Application working point A consists of 3 masters and 3 slaves with 5 communication links, while Application working point B consists of 3 masters and 5 slaves with 9 communication links. The graphs are annotated with latency and throughput requirements.

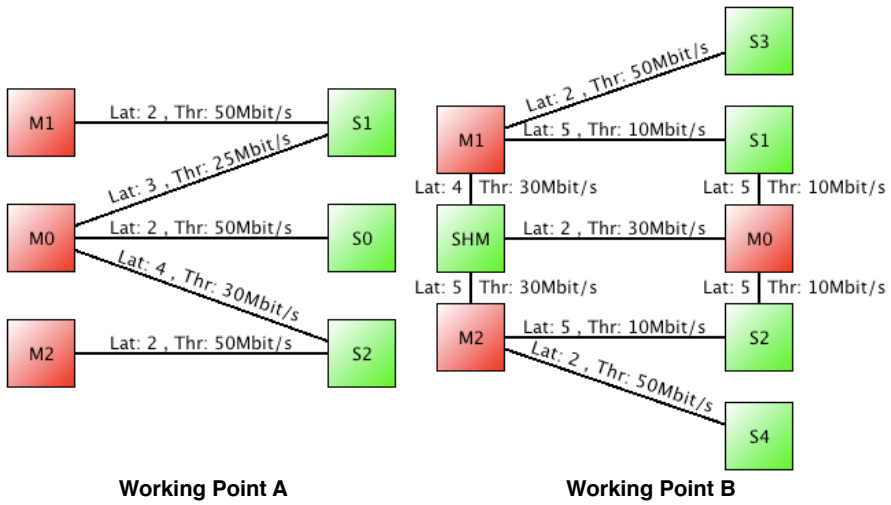

FIGURE 1.4
Communication graphs of Working Points A and B.

1.2.2 Communication Infrastructures Generation

For each application working point described in the specification in input, a set of NoC infrastructures is generated; each of them is derived from a specific template by instantiating the minimum number of switches required by the applications. The current implementation of this phase supports both stan-

dard and custom NoC–based communication infrastructures: *ring*, *star*, *mesh*, *spidergon* and custom (usually a composition of the previous topologies). Note that this phase instantiates only the backbone structure of the provided set of target topologies; this means that, so far, no mapping activity of the cores to the NoC infrastructure is performed.

Running example

Some of the topologies generated in this phase for the running example are reported in Figure 1.5: *(i)* a $2x3$ mesh and *(ii)* a spidergon with diameter[1] 3 for application working point A and *(iii)* a $3x3$ square–mesh and *(iv)* a ring with 8 elements for application working point B.

1.2.3 Mapping

The mapping phase aims at deploying the working points of the system specification received in input on the communication infrastructures generated in the previous step. In particular, the activity is performed for each pair composed by an application working point and an instantiated communication infrastructure; the output is the specification of the position in the communication infrastructure where each core of the working point is mapped on. All pairs for which no feasible mapping solution are identified are automatically discarded and no more considered in the subsequent phases.

Currently, three different algorithms have been developed to find the best feasible mapping solution (or at least a good sub–optimal solution) for a large set of different target reconfigurable systems. In particular, one exhaustive and two genetic algorithms have been developed: *Smart Exhaustive*, *GA1ver* and *GA2ver*, that are discussed in the following sections. In an empirical evaluation we carried out, we have noted that each algorithm outperforms the other ones depending on the characteristics of the considered working point, in particular according to the number of cores. For this reason, the design flow automatically selects the algorithm to be used for each working point according to its size, as discussed in the following.

Smart Exhaustive Algorithm

The **Smart Exhaustive algorithm** can be used only with very small application working points (up to 12 cores), since it evaluates all the feasible mapping solutions of a given (application working point–communication infrastructure) pair, computing the average latency on all the connections between the cores and storing this value for each solution that has been found.

Algorithm 1 shows the pseudo–code of this smart exhaustive algorithm; its

[1]The diameter has been considered as the maximum number of hops between every couple of cores.

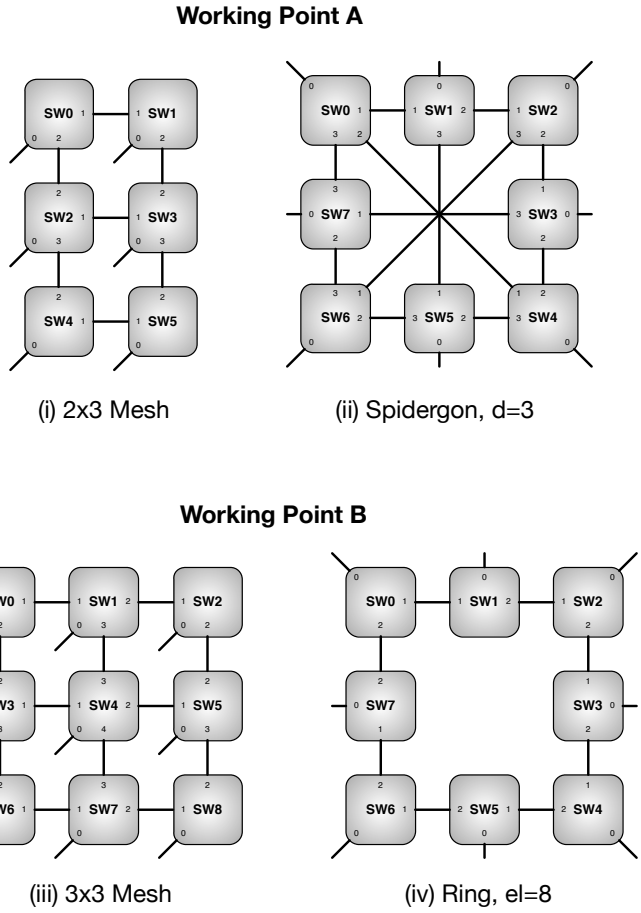

FIGURE 1.5
Topologies generated for Working Points A and B.

inputs are the number of working point's cores (*elements*) and the number of the infrastructure switches on which the elements have to be mapped (*slots*).

This algorithm includes a set of rules and policies that allow the exclusion of all the unfeasible solutions in an early stage of the process and to reduce the computational time required for the completion of the task. All these checks are performed within the *SmartSelection()* function (Lines 5 and 9 of Algorithm 1). In particular, once a single core is mapped on a single switch port of the communication infrastructure, the rules adopted for the selection of the next core to be mapped are the following ones:

- *first rule*: if some cores have already been mapped on the selected communication infrastructure, then the subsequent core to be mapped has to be selected among the cores that are directly connected to (at least) one

Algorithm 1: SmartExhaustive (*elements, slots*)

```
1  Create a temporary array solution: temp
2  Create a list of temporary array solutions to be evaluated: evaluation_list
3  Create a list of final array solutions: solutions_list
4  for i ← 1 to elements do temp[i] ← −1;
5  elem ← SmartSelection();
6  AddConfigurationsWith(elem, temp, slots, evaluation_list, solutions_list);
7  while (evaluation_list is not empty) do
8      temp ← evaluation_list.pop();
9      elem ← SmartSelection();
10     AddConfigurationsWith(elem, temp, slots, evaluation_list, solutions_list);
11 end
12 if (solutions_list is empty) then Exit();
13 else
14     ComputeFitness(solutions_list);
15     SortByFitness(solutions_list);
16 end
17 return solutions_list;
```

of the mapped cores. This rule makes it possible to firstly complete all the dependencies among the mapped and the unmapped cores before starting with the mapping of unconnected cores;

- *second rule*: if there are more than one core that is directly connected to (at least) one of the mapped cores (as specified by the *first rule*), then the core with the highest number of connections with the already mapped cores is selected as the subsequent core that has to be mapped;
- *third rule*: if no cores have already been mapped or if there are no cores that are directly connected to (at least) one of the mapped cores, then the subsequent core to be mapped is selected among the cores that have the highest number of connections with the lowest latency constraint (since these are the most strict constraints).

GA1ver

The **GA1ver** algorithm shown in Algorithm 2 is a custom single–objective genetic algorithm. This algorithm is suitable for small or medium application working points (12 - 20 cores).

In **GA1ver**, each individual represents a possible mapping solution among the computational cores and the switches; i.e. the individual chromosome encodes for each computational core the switch on which it is mapped onto. Then, to evolve the population custom crossover and mutation operators have been defined. The crossover operator takes two individuals and combines them in the following way: all the computational cores that are mapped in the same position in both the individuals, will be mapped in the same position also

in the offspring, while all the other computational cores will be randomly mapped in the other available locations. The mutation operator modifies a single individual by randomly swapping the locations of two different computational cores. The *GA1ver* has been developed in order to minimize the average latency on all the connections between the couple of computational cores that have to communicate in the target reconfigurable system.

It is important to note that, in *GA1ver*, all the individuals represent feasible solutions (mapping solutions in which there are no constraints violations, Line 7 of Algorithm 2). Obviously, both crossover (Lines from 15 to 29 of Algorithm 2) and mutation (Lines from 30 to 35 of Algorithm 2) have to be redefined accordingly to the new definition of individual, thus both of them are applied as many times as required in order to produce a feasible solution (since they are intrinsically random, each time they are applied it is possible to obtain a different output).

GA2ver

In order to further speed–up the mapping task, a different version of the genetic algorithm has been developed that does not check, during the iterations, if a solution is feasible or not (*fast_version* flag on Lines 7, 25 and 34 of Algorithm 2). Once the execution of the algorithm is completed, thus the set of mapping solutions has been successfully obtained, a filter function discriminates whether a solution is feasible or not considering the communication graph constraints (Line 40 of Algorithm 2). Thanks to this optimization, it is possible to decrease the total execution time of the mapping algorithm with respect to *GA1ver*. This algorithm can be effectively applied on the application working points that present a medium-large size (this algorithm has to be preferred for working points consisting of more than 16 computational cores).

Running example

Each application working point of the running example have been mapped on the communication infrastructures generated in the previous phase. Figure 1.6 shows the mapping solutions (the ones related to the topologies presented in Figure 1.5) that have been obtained with the *GA2ver* algorithm in less than 0.01 ms.

Considering the solutions *(ii)* in Figure 1.6, it is possible to see that this spidergon topology has unused switches (*sw0* and *sw7*). This is because the spidergon topology is a regular topology that must ensure the minimum number of hops between each couple of processing elements; thus, without these two unused switches the resulting topology is no more a spidergon. A topology that after the mapping phase has unused switches is called *over–dimensioned solution*.

Another interesting consideration is that every latency constraint is not only satisfied but also minimized, when possible. An example can be seen considering the solution *(i)* of Figure 1.6, in which the latency constraint

Algorithm 3: GeneticMapping (*slots, fast_version*)

```
1  Create a list of final array solutions: solutions_list;
2  Generation of the initial population as follows:
3  repeat
4      Create a temporary solution: temp
5      for i ← 1 to slots do temp[i] ← i;
6      for i ← 1 to slots do Swap(temp, i, RandomSelection(1, slots));
7      if (temp is feasible) ∨ (fast_version is true) then solutions_list.push(temp);
8  until (solutions_list.size() < slots ∗ POPULATION) ;
9  ComputeFitness();
10 SortByFitness();
11 repeat
12     for i ← slots ∗ SELECTION to slots ∗ POPULATION do
13         a ← RandomSelection(1, slots ∗ SELECTION);
14         b ← RandomSelection(1, slots ∗ SELECTION);
15         Crossover operation:
16           for j ← 1 to slots do
17               solutions_list[i][j] ← solutions_list[a][j];
18               fixed[j] ← (solutions_list[a][j] == solutions_list[a][j]) ? true : false;
19           end
20           for j ← 1 to slots do
21               if (fixed[j] is false) then
22                   tmp ← RandomSelection(1, slots);
23                   if (fixed[rnd] is false) then
24                       Swap(solutions_list[i], j, rnd);
25                       if (fast_version is false) ∧ (solutions_list[i] is not feasible) then
26                           Swap(solutions_list[i], j, rnd);
27                   end
28               end
29           end
30         Mutation operation:
31           a ← RandomSelection(1, slots ∗ SELECTION);
32           b ← RandomSelection(1, slots ∗ SELECTION);
33           Swap(solutions_list[i], a, b);
34           if (fast_version is false) ∧ (solutions_list[i][j] is not feasible) then
35             Swap(solutions_list[i], a, b);
36     end
37     ComputeFitness();
38     SortByFitness();
39 until (number of required ITERATIONS is reached) ;
40 if (fast_version is true) then CleanSolutions(solutions_list);
41 return solutions_list;
```

between *m0* and *s2* were 4 (as pictured in Figure 1.4) and the actual hops between these two processing elements are only 2: *sw2* and *sw4*.

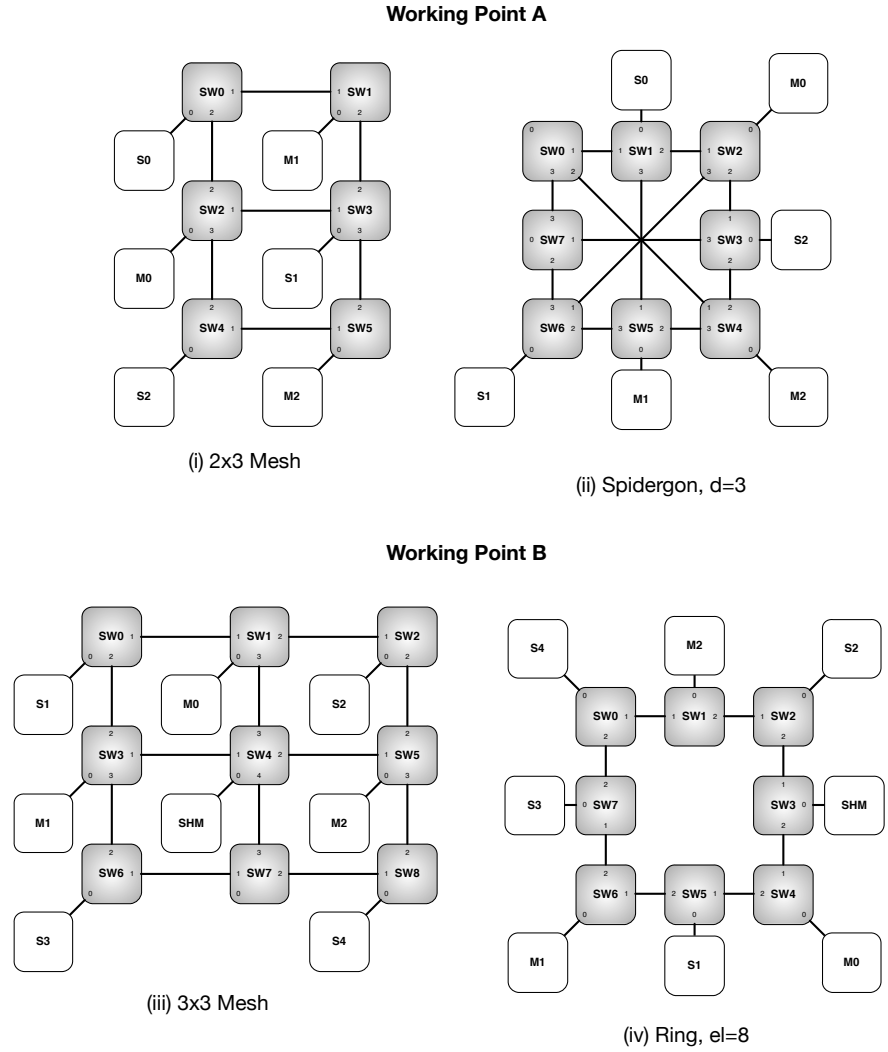

FIGURE 1.6
Mapping solutions for Working Points A and B.

1.2.4 Routing

This phase performs for each identified mapping solution the search of all the minimal routing paths between each couple of cores (master–slave), generating the best routing solution based on the throughput constraints and on the switch throughput upper bound value. The computational requirements of this phase are quite small with respect to the rest of the flow and we never noticed scalability issues for the routing phase in all the synthetic and real-

world case studies we have analyzed, so it has not been necessary to employ a complex and optimized routing technique to accomplish this task (even though any custom routing algorithm can be easily substituted, if needed, to the one proposed in our approach). In particular, the proposed algorithm performs a recursive iteration over the mapping solutions provided by the previous phase, executing for each one of them the following steps, until a valid solution is found:

1. compute all the possible (and minimal) routing paths from each source to its corresponding target;
2. store the different throughput values on each link;
3. select the routing paths that minimize the average throughput on the links;
4. verify that the solution is valid.

In particular, a routing solution can be considered a valid solution only when these two constraints are satisfied:

- all the throughput constraints specified in the *high level specification* phase are satisfied;
- all the switches of the current communication infrastructure have to handle no more than the *switch throughput upper bound* value provided in input by the designer during the *high level specification*.

If there is no valid routing solution, the overall mapping solution is discarded from the next phases. Moreover the routing paths ensure also the latency constraints in order to exclude all the solutions that have two neighbor elements that, for instance, exploit a longer routing path to communicate.

Running example

Table 1.2.4 presents an example of routing solution (obtained in less than 0.002 ms) that refers to the *(iii)* $3x3$ mesh solution of Figure 1.6.

Considering all communication paths in Table 1.2.4, it is possible to create the list of switches involved in the communication, giving for each entry the exact amount of throughput needed, as can be seen in Table 1.2. Each throughput value is under the upper bound of 110.

1.2.5 Evaluation and Selection

During this step, for each application working point the various mapping and routing solutions generated during the previous steps, are evaluated. In particular, different cost factors are computed in order to select the best–fitting communication infrastructure for the specific application working point considering the objective functions specified by the designer. The current implementation of the framework considers the following objective functions, which

SRC − TRG	Throughput	Paths Found	Paths list (sw ids)
m2 – s2	5	1	**5, 2**
m2 – shm	30	1	**5, 4**
m2 – s4	50	1	**5, 8**
m1 – shm	30	1	**3, 4**
m1 – s1	10	1	**3, 0**
m1 – s3	50	1	**3, 6**
m0 – s2	10	1	**1, 2**
m0 – shm	30	1	**1, 4**
m0 – s1	10	1	**1, 0**

TABLE 1.1
Routing solution of the *(iii)* $3x3$ square–mesh mapping solution of application working point B.

Switch	*sw0*	*sw1*	*sw2*	*sw3*	*sw4*	*sw5*	*sw6*	*sw7*	*sw8*
Throughput needed	20	50	20	90	90	90	50	0	50

TABLE 1.2
Throughput needed by each switch considering the routing solution for the *(iii)* $3x3$ square–mesh topology and a switch throughput upper bound value of 110.

are among the most important metrics for NoC design [3]: *area minimization* (that also impacts on the *power minimization*); *throughput variance minimization*; *maximum throughput minimization*; *average latency minimization.* In addition, it is also possible to associate weights to these functions, in order to give more importance to a subset of them (in particular, it is possible to employ a linear combination of the previously presented objectives). According to the objective functions values obtained from each infrastructure, only one communication infrastructure for each input application working point is selected.

Running example

After the *mapping* and *routing* phase, the *evaluation and selection* phase iterates over the four mapped and routed solutions of Figure 1.6, to select the best–fitting communication infrastructure with respect to the provided objective functions. The results obtained considering the area, throughput variance, maximum throughput and average latency objective functions are presented in Table 1.3. In particular, the table shows that the selected communication infrastructures, considering the *total* normalized values, are the $2x3$ mesh for application working point A and the ring for application working point B, respectively; the **bold** highlighted values in these tables represent that the

corresponding metrics is minimized for that particular communication infrastructure.

App. Working Point	**Topology**	Area (slices)	Thr. Variance	Max Thr.	Avg Lat.	**Total** (norm.)
A	***2x3 mesh***	**4145**	**422.22**	105	2	**317,12**
A	*spidergon w/ d=3*	5073	1192.19	105	2	400
B	*3x3 mesh*	5733	1009.88	**90**	**2**	371,97
B	***ring w/ el=8***	**5083**	**425**	100	2.44	**330,75**

TABLE 1.3
Reasoning results for application working points A and B.

1.2.6 Placement

After the selection of the best communication infrastructure for each application working point, the subsequent phase consists in the placement of the communication infrastructures on the target physical device. Since the target architecture is based on a grid of tiles, the placement phase aims at finding the location of each element of the application working point (a switch or a master/slave core along with its network interface) in a single tile of the architecture in such a way that all the elements can be correctly connected among them.

The whole *placement* essentially consists of two stages: (i) *the initialization stage* is performed through the partitioning of the target device and the generation of the placement order vector, and (ii) *the second stage* is the actual execution of the algorithm in order to find valid placement solutions. In particular, the first part of the initialization stage is related to the *partitioning* of the target reconfigurable device in an *homogeneous grid* of reconfigurable regions. The number of reconfigurable regions in which it will be possible to split the target device will be evaluated according to the following two values: (i) the maximum size of the input application working points, evaluated as total number of elements (cores and switches), and (ii) the size of the component with the highest area usage, in order to maintain the homogeneity of the reconfiguration approach (it has to be possible to place each element of the system in each reconfigurable region).

The second part of the initialization stage deals with the generation of the *placement order vector* (POV), whose elements represent either cores (master or slave) or switches. Furthermore, each element also contains (i) the information on all its links towards the other elements of the system, (ii) the number of links that still have to be inserted in the POV and (iii) a link to the *father element*, that is the first element inserted in the *placement order vector* directly connected to the selected element. In order to build a good POV, the

first element that has to be inserted in the vector is the most constrained one, in other words the element with the highest number of connections towards the others. Starting from this element, it is possible to fill the *POV* by inserting the elements directly connected to the already inserted one. Among all these elements, the following element to be inserted will be the one with the highest number of connections towards the others, and so on. In this way it is possible to place in the first positions of the *placement order vector* the most constrained elements (the ones with a very large number of connections with other elements); this choice will lead, during the actual placement process, to drastically reduce the number of valid solutions in an early stage of the process, thus reducing the execution time of the whole *placement* phase.

The *Connection Matrix* (*CM*) represents the partitioned reconfigurable device and is used to support the placement algorithm; all the cells of the *CM* are initialized with a value that represents the maximum number of connections that the selected cell is able to support in the target reconfigurable architecture. This number depends on the maximum length that each connection (in the *communication layer*) can assume; for instance, if the maximum length is set to 1, only adjacent cells can communicate, while if it is set to 2, it is possible to skip (at maximum) one cell on each communication channel. The designer can manually modify this parameter to fine tune the final solution towards a more spread system (if the parameter is set to a high value, more than 3 or 4) or a more condensed one (if the parameter is set to a low value, such as 1 or 2). Two extra sets of rows and columns (the number of elements of these sets depends on the maximum length that each connection can assume) have been added on each side of the *CM* and each cell of these rows and columns has been initialized with a negative value (-1), that is the value associated with a *busy* cell. Thus, it is possible to evaluate all the possible placement solutions on the generated *CM* by means of a *depth first algorithm*, shown in Algorithm 4.

In order to optimize the proposed algorithm, the framework is able to exploit the *specular* and *symmetrical* properties of a rectangular grid. For instance, the algorithm is not directly applied to the whole matrix of cells, since it is sufficient to start from a subset of that matrix (that is the portion of the matrix comprised between a diagonal and a vertical or horizontal axis of symmetry) and to use the obtained partial set of solutions to obtain the whole set of solutions by means of geometrical transformations. In this way, the placement algorithm can compute all the solutions up to 8 times faster.

1.2.7 Reconfigurations Minimization

At this point, the framework has to select for each working point a placement solution among the ones generated by the previous step. In particular the choice is performed with the aim of minimizing the reconfigurations needed to pass from one application working point to the subsequent one (even though sometimes the working points sequence is not known a priori). Indeed, each

Algorithm 4: Placement (*POV, CM*)

```
1  Select the first element, A, from the POV ;
2  Remove A from the POV ;
3  repeat
4      Select a reconfigurable region C;
5      if ((connections of C ) ≥ (connections of A )) then
6          newCM ← CM;
7          place (A in C) in newCM;
8          (connections of C ) in newCM ← −1 ;
9          if (POV is empty) then
10             add newCM to the Solutions Array;
11         else
12             repeat
13                 Select a reconfigurable region C;
14                 if (D is connected to C ) then
15                     (connections of C) in newCM ← ((connections of C) in newCM) −1 ;
16                 end
17             until (no more reconfigurable regions) ;
18             Placement (POV, newCM)
19         end
20     end
21 until (no more reconfigurable regions) ;
```

reconfiguration process changes the content of a single reconfigurable region, holding either a computational core (master or slave) along with its network interface or a network switch. Minimizing the number of reconfigurations implies the reduction of the reconfiguration timing overhead, since this overhead is directly proportional to the number of regions that have to be reconfigured.

The main goal of this phase is the maximization of the number of components (computational cores and switches) that are physically placed in the same reconfigurable region for more than one input application working point. This means that the average number of reconfigurable regions that need to be reconfigured when switching application working point will be minimized. This search can be performed in an exhaustive way by considering only a subset of all the placement solutions. In particular, it is possible to start the computation from the placement solutions that present the lowest average physical distance among the connected components (considering the physical distance among the reconfigurable regions holding connected components). In this way the search is performed starting from the best placement solutions and it is concluded when the time slot reserved for this phase (usually some tens of seconds) is elapsed.

Running example

Finally, the *placement* and the *reconfigurations minimization* phases make it possible to perform a physical placement of the best mapping solutions and to find different placements (one for each application working point) that

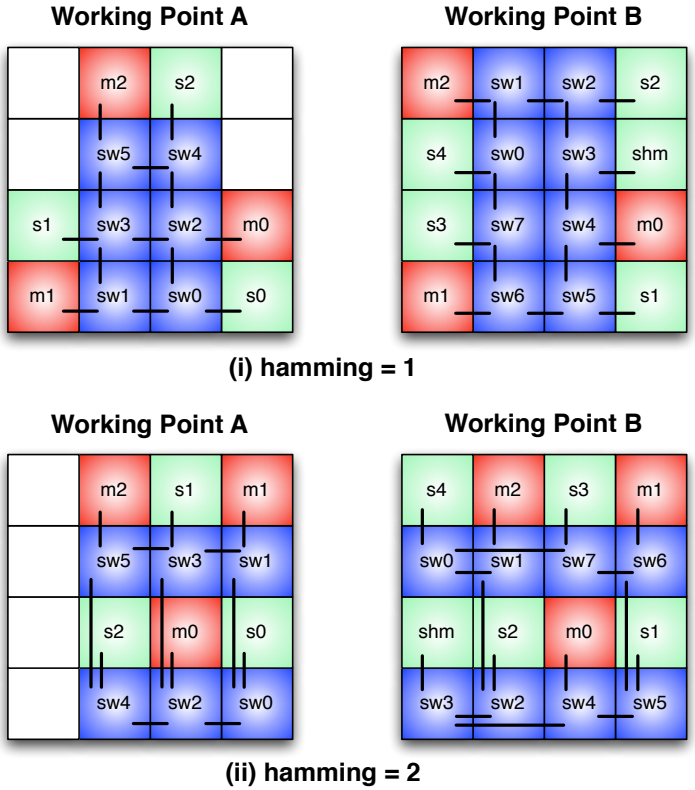

FIGURE 1.7
Two placement solutions for Working Points A and B on a $4x4$ grid with hamming distance set to 1 *(i)* and hamming distance set to 2 *(ii)*.

minimizes the number of reconfigurations needed to pass from one application working point to another one. The solutions proposed in Figure 1.7 (obtained in less than 0.15 s) minimize the reconfiguration time to pass from application working point A to B.

Considering the mapping solutions *(i)* and *(iv)* of Figure 1.6, the corresponding placement solutions are reported in Figure 1.7. Two different placement solutions have been proposed, both found on a $4x4$ grid: *(i)* considering the maximum hamming distance equal to 1 and *(ii)* considering the same distance equal to 2. As shown in Figure 1.7, the total reconfiguration cost (RC) needed to switch from application working point A to B decreases with the increment of the hamming distance. These placement solutions have been selected by the *reconfigurations minimization* phase among all the valid solutions in few tens of seconds. Figure 1.8 shows the reconfigurations minimization considering the hamming distance equal to 1. There can be different

reconfiguration policies that can be exploited to optimize the reconfiguration cost. As shown in Figure 1.8, three different cases can be identified.

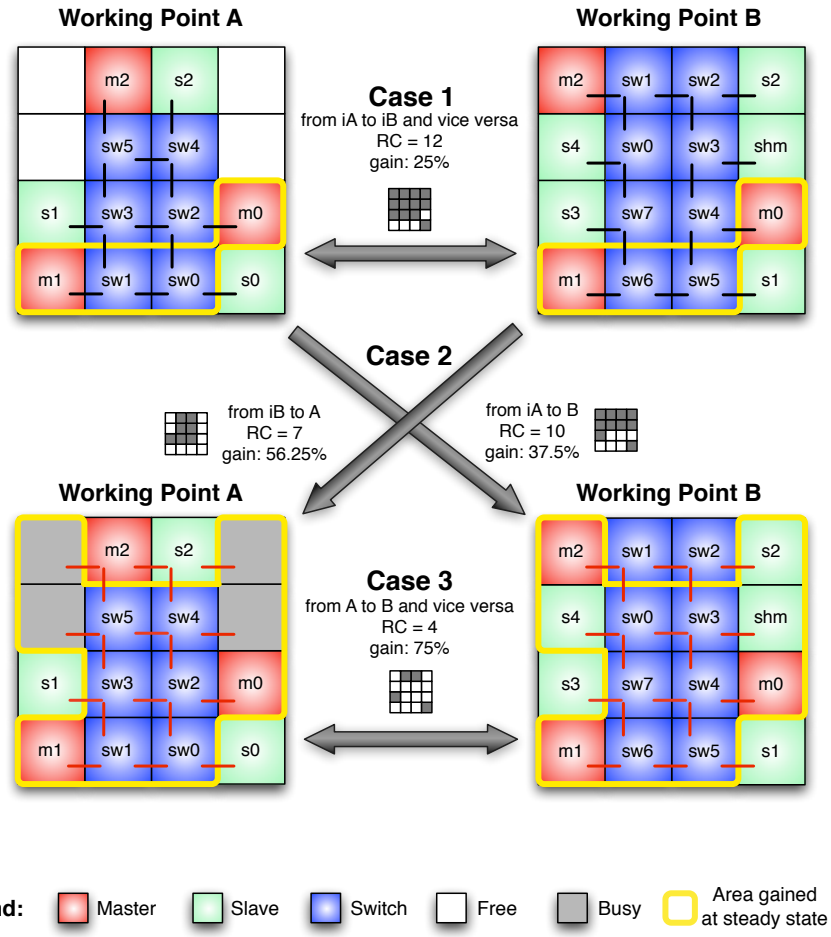

FIGURE 1.8
Reconfiguration cost introduced switching from Working Point A to B and vice versa, considering three possible cases.

Case 1: the first case deals with a reconfiguration that does not consider any kind of optimization. In particular in Case 1 of Figure 1.8, switching from one application working point to the next one, refers to the working point structure found by the *placement phase* (iA, the initial configuration for application working point A and iB, the initial configuration for application working point B), without keeping into consideration the possibility of having floating communication links or unused tiles. This type of policy produces a reconfiguration cost (RC) of 12, which means that at least 12 tiles of the grid

must be reconfigured, corresponding to a 25% of reconfiguration time speed up.
Case 2: the second case considers a possible initial situation in which, for instance, the system starts the execution of application working point A and then switches to application working point B considering two significant optimizations. In Case 2 of Figure 1.8, application working point B implements a communication infrastructure that is a merge of both the infrastructures of application working points A and B, which means that there can be unused or floating communication links and also unused tiles. These two optimizations (A for application working point A and B for application working point B) bring to a reconfiguration cost (RC) of 10 and 7, for the switch from A to B and B to A, respectively. The reconfiguration time gained is 37.5% and 56.25%, respectively, when compared to a complete reconfiguration.
Case 3: the last case considers the situation in which the system starts one application working point and then switches to the other one as described in Case 2, performing all the previously described reconfigurations. After that, the system enters in the so called *steady state*, represented by Case 3 of Figure 1.8. In this scenario, it is possible to keep both unused tiles and floating communication links and the reconfiguration cost (RC) is 4, with a significant gain of 75%. Finally, Figures 1.9 (a) and (b) show the physical implementation on a Virtex 5 XC5VLX110T device of the two placement solutions presented in Figure 1.7 with the hamming parameter set to 1.

1.3 Related Work

In the past few years a large body of work has addressed the study of the NoC paradigm and the definition of suitable design flows to reduce design time and effort, even though most of them focuses on the optimization of the NoC architecture for a single application. For instance, [8] proposes a set of specification and generation steps of the *μspider* NoC design flow. The overall flow consists in the definition of hardware settings such as: *data word size*, *communication mode*, *topology* and *virtual channels*. On the other hand, [5] describes a *monitoring–aware NoC design flow*, in order to improve the debugging and performance analysis of automatically generated NoC infrastructure. Their main target architectures are ASIC–like designs and so the application is known at design time.

In [17] the authors show that with the integration of the mapping and the physical planning phases, the resulting NoC design can be improved with respect to several aspects. The authors target the NoC design for complex System–on–Chips (SoCs) with *heterogeneous* processor/memory cores, improving Quality–of–Service (QoS) for the application. Their work presents an integrated approach for mapping cores onto NoC topologies and NoC physical

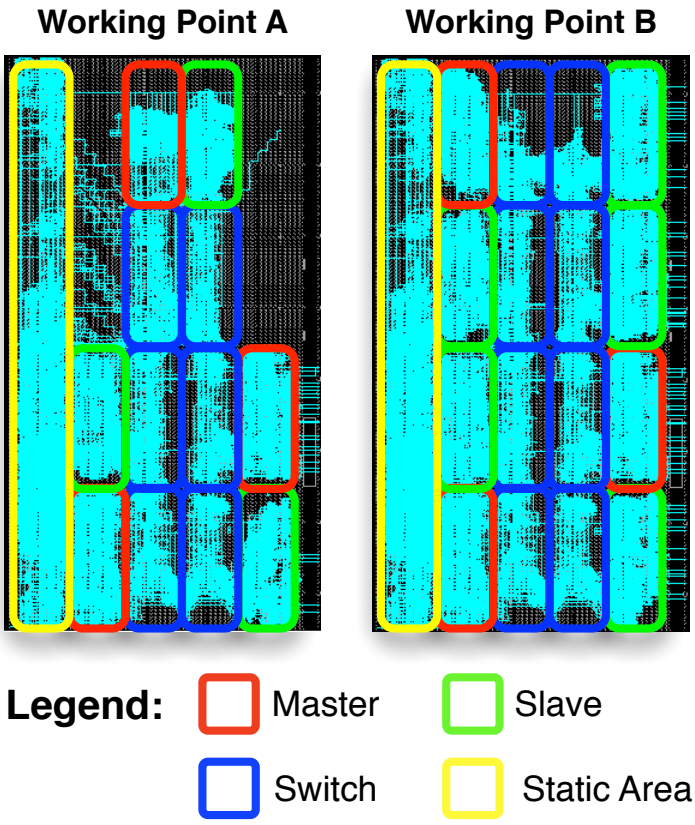

FIGURE 1.9
Implementation of Working Point A and of Working Point B on a Virtex 5 device with the hamming parameter set to 1

planning. [10] proposes a mapping algorithm for energy-aware placement of the cores considering the communication volume between cores and needed bandwidth. The presented branch–and–bound algorithm can automatically map the computational cores onto a generic regular NoC architecture such that the total communication energy is minimized. At the same time, the performance of the mapped system is guaranteed to satisfy the specified constraints through bandwidth reservation. The same authors propose in [11] an improvement to their algorithm constructing a deadlock–free deterministic routing. They show how the routing flexibility can be exploited to expand the solution space and improve the solution quality. Finally, [2] surveys the main challenges in application-specific NoC design and outlines an application-specific NoC design flow and methodology.

X-pipes is an example of a complete flow for designing NoCs and it is composed of different phases. In the first phase, the user specifies the objectives

and constraints that should be satisfied by the designed NoC. In the second phase, the NoC architecture that optimizes the user objectives and satisfies the design constraints is automatically synthesized. The last phase of the flow consists in the automatic generation of the synthetizable HDL code of switches, network interfaces and links for the designed topology.

All the previously presented works, address the problem of the automatic generation of efficient NoCs (such as monitoring–aware or layout–aware NoCs) that are specifically developed for a particular application (or set of applications) but that are essentially static; in fact, the NoC reconfiguration is not taken into account. On the other hand, several works have addressed the definition of reconfigurable NoC architectures (e.g. [12] and [19]), while much less effort has been spent in the definition of design flows for the development of reconfigurable and adaptable NoCs, characterized by the possibility of dynamically modifying some architectural parameters at run–time. [9] presents a set of tools that make it possible to control changes in a NoC, showing some architectural additions and describing a library to change at run-time some parameters of the NoC, but without proposing a complete design flow. The work proposed by [15] is a framework for the generation of Multi–Processor Systems-on-Chip (MPSoCs) based on NoCs. This framework is an extension of the Xilinx EDK tool-chain to support the automatic generation of NoC–based MPSoCs, even if no design space exploration is performed and all the proposed examples are based on a NoC that only consists of a single switch (no algorithms have been proposed to build a different NoC topology and to map cores on it). Finally, the work proposed by [13] addresses the automatic generation of a run–time reconfigurable NoC–based MPSoC architecture. The authors present a complete design flow that is able to generate, starting from the high-level specification, the VHDL code of the desired reconfigurable system. In this approach, both the cores and the NoC topology (thus also the connections among cores and switches) cannot be dynamically changed at run–time, since they have to be fixed at design–time and implemented as static components of the final design.

A reconfigurable NoC architecture, called ReNoC, has been presented in [21]. It enables the network topology to be configured for the application running on the SoC by using topology switches. However, the customization of the NoC structure and topology is limited to two main modifications that can be performed at run-time: router bypass and links insertion. Similarly, in [16] a reconfigurable NoC architecture has been proposed on which regular and application specific topologies can be implemented according to the application running by means of a configurable communication layer. The concept of *router bypassing* has been exploited also in [24], where the authors have proposed a Dynamic Bypass Circuit and a north-last weave routing algorithm to realize a dynamically reconfigurable NoC. Another flexible network design has been proposed in [1]: it is particularly suitable for building networks with irregular topologies, characterized by low latency and high throughput. In [7], authors have proposed a reconfigurable NoC by clustering of cores. They have

	Rec. Support	**Cores Rec.**	**NoC topology Rec.**	**Standard & Custom Topologies**	**Map Phase**	**Route Phase**	**Place Phase**	**From HLS to bit**
[8]	No	No	No	Yes	Yes	Yes	No	No
[5]	No	No	No	No	Yes	Yes	No	No
[10]	No	No	No	Yes	Yes	Yes	No	No
[12]	Yes	Yes	No	No	No	Yes	No	No
[9]	Yes	No	Yes	Yes	No	No	No	No
[15]	Yes	No	No	No	No	No	No	Yes
[13]	No	No	No	No	No	No	No	Yes
[22]	Partially	No	No	No	Yes	Yes	No	No
[4]	Yes	No	No	No	Yes	Yes	Yes	No
[14]	Yes	No	No	Yes	No	No	No	No
[20]	Yes	No	No	No	Yes	Yes	No	No
[21]	Partially	No	Partially	Yes	Yes	Yes	No	No
[16]	Partially	No	Yes	Yes	Yes	Yes	No	No
[24]	Partially	No	No	No	Yes	Yes	No	No
[1]	No	No	Yes	Yes	Yes	Yes	No	No
[7]	Yes	No	No	No	Yes	Yes	Yes	Yes
[17]	No	No	No	Yes	Yes	Yes	Yes	No
[2]	No	No	No	Yes	Yes	Yes	Yes	No
[19]	Yes	Yes	Yes	Yes	No	Yes	No	No
P.F.	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

TABLE 1.4
Comparisons with state–of–the–art solutions.

connected many cores per router which may lead to increase in the complexity and power consumption of the routers. Only one application is taken at a time for mapping onto the reconfigured NoC. Thus, differently from our approach, the mapping may not be suitable (or, at least, optimized) for other applications. However, in all these approaches, the reconfiguration of cores is not taken into account, thus all the cores belonging to all the applications that will be executed at run-time on the system have to be configured statically on the device. In fact, the adaptation of the underlying system to the currently running application mainly consists in the modification of some parameters of the communication infrastructure (see also the works proposed in [22, 4, 14, 20]), while the number, kind and position of computing cores cannot be dynamically modified, thus severely limiting the flexibility of this kind of approaches.

Table 1.4 presents a comparison between the proposed flow (named P.F.) and the solutions found in literature. The terms of comparison have been the capability of the various approaches to support the following features: 1) run-time reconfiguration, 2) cores run-time reconfiguration, 3) NoC topology run-time reconfiguration, 4) standard and custom topologies, 5) mapping phase, 6)

routing phase, 7) physical placement phase, 8) automatic generation, starting from the high-level specification, a complete set of bitstreams that are able to configure on the target device the desired reconfigurable system. Among the considered approaches, it is possible to state that the most interesting ones, i.e. the ones supporting the highest number of features, are the works presented by [17], [2] and [19].

The works proposed by [17] and by [2] support standard/custom topologies and mapping, routing and placement phases, even though they are essentially static, since reconfiguration is not taken into account at all. On the other side, from the reconfiguration point of view, the only approaches that consider the reconfiguration are [19] and the proposed work. [19] on the contrary of the proposed approach, it does not include mapping and placement algorithms and it is not able to automatically generate, starting from the high-level description of the system, the desired physical implementation.

1.4 Algorithm Performance Analysis

Table 1.5 presents the timing performance of the proposed design flow for reconfigurable NoC–based architectures. In particular, the proposed design flow has been used to generate a reconfigurable system for different applications, varying the number of elements in order to show how the flow execution time grows. Five applications have been selected: *i)* application A with 6 cores, *ii)* application B with 9 cores, *iii)* application C with 16 cores, *iv)* application D with 32 cores and *v)* application E with 64 cores. It is important to note that all the timing results presented in Table 1.5 are perfectly compatible with the ones required for a design flow, since they are in the order of tens or hundreds of seconds at maximum. Moreover, different state–of–the–art algorithms have been analyzed in order to show in particular the timing performance comparison of the mapping phase. Table 1.6 presents the execution time[2] of different mapping algorithms plus the routing process, varying the number of core elements. The listed algorithms, even if considering different constraints, perform an optimized mapping of the cores on a 2D mesh communication infrastructure, the most common NoC topology that can be found in literature. The timing results of the proposed algorithms have been evaluated by running them on the same input applications used by the state-of-the-art approaches (when these applications were explicitly shown) and considering the average value over 100 executions. Even though it is not possible to compare the timing performance of the different approaches in a completely fair way, since they are considering different constraints and optimization objec-

[2]For the exhaustive and the genetic algorithms, the execution times have been taken on a MacBook Pro 2.16 GHz with 2 GB of RAM.

TABLE 1.5
Design flow timing performance.

	App. A (6 cores)	**App. B (9 cores)**	**App. C (16 cores)**	**App. D (32 cores)**	**App. E (64 cores)**
HLS	<1 s	<1 s	<1 s	<1 s	<1 s
CIs Generation	<1 s	<1 s	<1 s	<1 s	<1 s
Mapping (exhaustive)	<1 s	<1 s	NA	NA	NA
Mapping (*GA1ver*)	<1 s	<1 s	<1 s	151.3 s	1780.4
Mapping (*GA2ver*)	<1 s	<1 s	8.1 s	17.3 s	71.s
Routing	<1 s	<1 s	<1 s	<1 s	<1 s
Reasoning	<1 s	<1 s	<1 s	<1 s	<1 s
Placement & Rec. Minimization	<1 s	32.2 s	42.2 s	47.3 s	120.5 s

TABLE 1.6
Timing comparison among different mapping algorithms.

Algorithm	2x2 mesh (sec)	3x3 mesh (sec)	4x4 mesh (sec)	5x5 mesh (sec)	6x6 mesh (sec)	10x10 mesh (sec)
PLBMR [23]	0.3	1.5	5.5	16.4	**44.9**	NA
GA [23]	0.9	6.6	65.6	346.6	1370.4	NA
BnB [23]	**0.234**	**1.345**	6.768	18.564	55.678	NA
SA [10]	NA	NA	25.55	181.67	hours	prohibitive
BnB [10]	NA	NA	NA	**6.52**	NA	minutes
EPAM-OE[11]	NA	NA	**0.31**	NA	NA	minutes
Exhaustive	**0.0002**	**0.011**	NA	NA	NA	NA
GA2ver	NA	NA	**0.003**	**1.584**	**3.312**	10s of sec
Gain	1170	122.27	103.33	4.11	13.55	NA

tives, the proposed approach, thanks to the automatic run–time selection of the employed algorithm, has been proved to be very fast, without reducing the quality of the final mapping solution, and very scalable on real-world applications, since it is able to rapidly perform the mapping of the cores, on both very small ($2x2$ and $3x3$) and very large (up to $10x10$) meshes.

1.5 Real-World Case Study

To demonstrate the completeness of the proposed approach, we applied our methodology to a real-world case study consisting of two different applications running on an embedded system provided with a Xilinx Virtex IV FPGA:

- *Mathematics Computation (MC)*, which basically involves a set of matrices multiplications;
- *Edge Detection (ED)*, which is fundamental in the image processing and in the computer vision fields and aims at identifying points in a digital image at which the image brightness has discontinuities.

The target system is a mobile robot that has to continuously perform the application *MC* in order to update its coordinates and to reconstruct a geometrical map of the environment, while the application *ED* is required only sporadically (in an completely unpredictable way) in order to elaborate the images acquired by the camera as a reaction to external inputs, such as a particular noise or a sharp change in the temperature. Thus, it is possible to identify two different working scenarios:

- *WS1*, consisting of an *MC* application;
- *WS2*, consisting of an *ED* application and an *MC* application (working on matrices that are smaller w.r.t. the ones employed in *WS1*).

Since the typical timing overhead for a complete reconfiguration of a Virtex IV FPGA device ranges roughly from 576 ms (for a XC4VLX25) to 1472 ms (for a XC4VLX60), while the time needed to perform the edge detection is in the order of hundreds milliseconds, a complete reconfiguration of the device in order to switch from *WS1* to *WS2* is clearly unacceptable (the time needed to switch to the new configuration and then to switch back to the old one when the computation is completed, which is around 2944 ms on a XC4VLX60, would be greater than the time needed to compute the data).

In this scenario, we are comparing the proposed solution with a static approach, where all the cores needed by the two applications are already present on the FPGA device, even though some of them (such as most of the memories) are shared between them (thus, they cannot run in parallel). It is important to note that this constraint considerably limits the possibilities of the proposed approach (as well as the quality of the solution it is able to generate), but it is necessary in order to perform a fair comparison with a static solution. In particular, the resources required on the FPGA device by the two working scenarios are the following:

- *WS1*: up to 16 *MC Cores* working at around 33 MHz, and 12 memories (to be split into 3 sets of memories, 2 for the input and 1 for the output values) with a maximum working frequency of 100 MHz;

- *WS2*: 2 *ED Cores* working at around 25 MHz, and 4 memories (1 input and 1 output memory for each *ED Core*) with a maximum working frequency of 100 MHz, plus 8 *MC Cores* working at around 33 MHz, and 8 memories (to be split into 3 sets of memories, 2 for the input and 1 for the output values).

The complete system is then composed by 18 computing cores (16 *MC Cores* and 2 *ED Cores*) and 12 memories (some of them are shared between the two applications). Thus, a system with 64 slots has been designed in order to hold all the components of the system and of the underlying NoC. In particular, the NoC switches connecting all the components have been set with a working frequency of 200 MHz and have been provided with a 5-slot buffer for each one of the four output directions, while the NoC timeout has been set to 450 *ns*.

Table 1.5 shows a comparison between an optimized static solution and the one automatically generated by the proposed approach. All the results have been obtained by means of modeling the system with SystemC and TLM [18]. It is important to note that the static architecture has to implement a trade-off between the two applications, while the proposed solution is able to optimize most of the metrics (such as increasing the average throughput and decreasing the average number of timed out sessions, dropped flits and hops), especially for what concerns the working scenario *WS2* (with an increment of more than 35% on the average throughput). The main cost of this optimization is the reconfiguration overhead introduced when switching from one application to the other. The approach presented in this paper makes it possi-

	WS1 (Static)	**WS1 (Static)**	**WS2 (Proposed)**	**WS2 (Proposed)**
Average Throughput (MB/s)	11,50	2,73	11,98	3,70
Transmitted Packets (per μs)	285	166	296	223
Transmitted Flits (per μs)	768	396	798	529
Average Timed Out Sessions (per μs)	1,8	0,8	1,5	0,2
Average Dropped Flits (per μs)	2,5	1,0	2,2	0,3
Average Number of Hops(per packet)	3,26	3,74	3,13	2,69
Average Number of Hops (per flit)	3,26	3,68	3,13	2,72

TABLE 1.7
Comparison between the static and the proposed solution.

ble to reduce this reconfiguration overhead to only 18 slots over 64 (only some of the connections among the NoC switches have to be modified, since all the computational cores are already present on the device). Obviously, the timing overhead strictly depend on the particular target device. Table 1.5 shows a comparison among the timing overhead w.r.t. to different FPGA target device. In order to make a fair comparison between the static approach and

FPGA Device	**Size (slices)**	**Number of slots**	**Size of each slot (slices)**	**Rec. overhead per slot** (ms)	**Total rec. overhead** (ms)
XCV4VLX15	6144	64	96	5	90
XCV4VLX25	10752	64	168	9	162
XCV4VLX40	18432	64	288	16	288
XCV4VLX60	26624	64	416	23	414

TABLE 1.8
Reconfiguration overhead on different target FPGA devices.

the partially reconfigurable one, we can compare the time needed to compute the application *ED* in WS2 by the static solution with the time needed to configure the device, compute the *ED* with the proposed solution and then configure the FPGA in order to restore the previous configuration of the device. Figure 1.10 shows this comparison, w.r.t. to the different target FPGA devices. As can be easily seen in Figure 1.10, the proposed solution performs

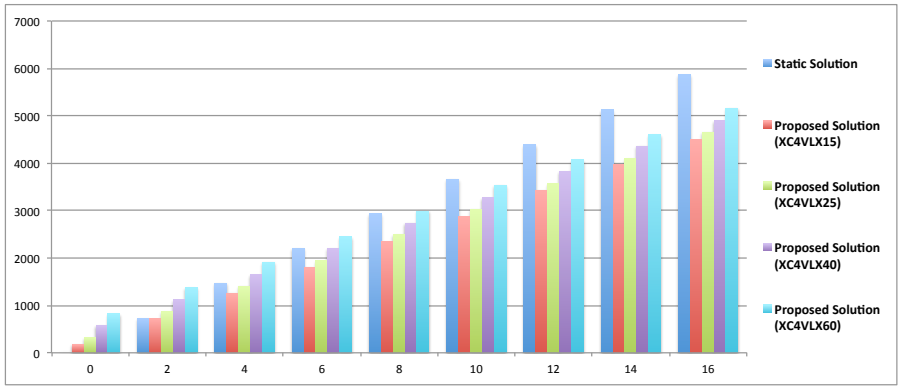

FIGURE 1.10
Comparison between the static solution and the proposed one (on different target devices) w.r.t. the amount of data (in MB) to be processed (horizontal axis) and the time (in ms) required to perform the computation (vertical axis).

better than the static one starting from around 2 MB (1920 $\times$ 1080 pixels,

which means Full HD resolution), since the time required to configure a Virtex IV XC4VLX15 device and perform the computation with the proposed solution is smaller than the time required to elaborate the image with the static solution (and much smaller than performing a complete reconfiguration). Furthermore, the benefits of the proposed approach increase with the growing of the amount of data to be processed (which is exactly the trend that all the embedded systems are facing nowadays). It is also important to note that, when the application *MC* is running in both the working scenarios, the proposed solution always outperforms the static one, since it has a higher throughput and a lower number of timed out sessions, dropped flits and average number of hops.

1.6 Concluding remarks

In this paper we presented a design flow that automatically defines a set of networks optimizing communication performances for a system with time-varying requirements. To achieve this goal, the flow is fed with the description of the communication requirements of the applications that will run on the system, in the form of *Communication Graphs*, and produces as output a set of networks, in which communication performances for the single *working point* and switching reconfiguration overheads are traded to maximize overall throughput. The flow is composed by several algorithms for the mapping, the routing and the physical placement of the developed reconfigurable system. In particular, the proposed mapping algorithms have been proved to be up to more than three order of magnitude faster than the other state-of-the-art algorithms. In the examples considered, the set of networks generated by the flow improved the throughput by 35% with respect to the optimal static NoC.

Bibliography

[1] T. A Bartic, J-Y Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Topology adaptive network-on-chip design and implementation. *Computers and Digital Techniques, IEE Proceedings* -, 152(4):467–472, July 2005.

[2] L. Benini. Application Specific NoC Design. In *Proc. of Intl. Conf. on Design, Automation and Test in Europe (DATE)*, pages 1–5, 2006.

[3] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. Cost considerations in network on chip. *Integration, the VLSI Journal*, 38(1):19–42, 2004.

[4] Chia-Hsin Owen Chen, Sunghyun Park, Tushar Krishna, Suvinay Subramanian, Anantha P. Chandrakasan, and Li-Shiuan Peh. Smart: A single-cycle reconfigurable noc for soc applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 338–343, March 2013.

[5] C. Ciordas, A. Hansson, K. Goossens, and T. Basten. A Monitoring-Aware Network-on-Chip Design Flow. In *Proc. of EUROMICRO Conf. on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 97–106, 2006.

[6] D. Cozzi, C. Farè, A. Meroni, V. Rana, M.D. Santambrogio, and D. Sciuto. Reconfigurable NoC design flow for multiple applications run-time mapping on FPGA devices. In *Proc. of ACM Great Lakes Symp. on VLSI (GLSVLSI)*, pages 421–424, 2009.

[7] Hui Ding, Huaxi Gu, Bin Li, and Keming Du. Configuring algorithm for reconfigurable network-on-chip architecture. In *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, pages 222–225, April 2012.

[8] S. Evain, J.-P. Diguet, and D. Houzet. A generic CAD tool for efficient NoC design. In *Proc. of Intl. Symp. on Intelligent Signal Processing and Communication Systems (ISPACS)*, pages 728–733, 2004.

[9] A. Hansson and K. Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. In *Proc. of Intl. Symp. on Networks-on-Chip (NOCS)*, pages 233–242, 2007.

[10] J. Hu and R. Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *Proc. on Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 233–239, 2003.

[11] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4), Apr. 2005.

[12] S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda. CuNoC: A Scalable Dynamic NoC for Dynamically Reconfigurable FPGAs. In *Proc. of Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 753–756, 2007.

[13] A. Kumar, A. Hansson, J. Huisken, and H. Corporaal. An FPGA Design Flow for Reconfigurable Network-Based Multi-Processor Systems on Chip. In *Proc. of Intl. Conf. on Design, Automation Test in Europe (DATE)*, pages 1–6, 2007.

[14] Ying-Cherng Lan, Hsiao-An Lin, Shih-Hsin Lo, Yu Hen Hu, and Sao-Jie Chen. A bidirectional noc (binoc) architecture with dynamic self-reconfigurable channel. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(3):427–440, March 2011.

[15] S. Lukovic and L. Fiorin. An Automated Design Flow for NoC-based MPSoCs on FPGA. In *Proc. of IEEE/IFIP Symp. on Rapid System Prototyping (RSP)*, pages 58–64, 2008.

[16] M. Modarressi, A Tavakkol, and H. Sarbazi-Azad. Application-aware topology reconfiguration for on-chip networks. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(11):2010–2022, Nov 2011.

[17] S. Murali, L. Benini, and G. de Micheli. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. In *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC)*, volume 1, pages 27–32, 2005.

[18] Open SystemC Iniative OSCI. *SystemC documentation (Last Check March 2008): http://www.systemc.org*.

[19] T. Pionteck, R. Koch, and C. Albrecht. Applying Partial Reconfiguration to Networks-On-Chips. In *Proc. of Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2006.

[20] J. Soumya, Ashish Sharma, and Santanu Chattopadhyay. A locally reconfigurable network-on-chip architecture and application mapping onto it. In *VLSI Design and Test, 18th International Symposium on*, pages 1–6, July 2014.

[21] Matthias Bo Stuart, Mikkel Bystrup Stensgaard, and Jens Sparsø. The renoc reconfigurable network-on-chip: Architecture, configuration algorithms, and evaluation. *ACM Trans. Embed. Comput. Syst.*, 10(4):45:1–45:26, November 2011.

[22] M. ValadBeigi, F. Safaei, and B. Pourshirazi. An energy-efficient reconfigurable noc architecture with rf-interconnects. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 489–496, Sept 2013.

[23] Z. Wenbiao, Y. Zhang, and Z. Mao. Link-load balance aware mapping and routing for NoC. *WSEAS Transactions on Circuits and Systems*, 6(11):583–591, 2007.

[24] Li-Wei Wu, Wei-Xiang Tang, and Yarsun Hsu. A novel architecture and routing algorithm for dynamic reconfigurable network-on-chip. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 177–182, May 2011.

[25] Xilinx Inc. *Partial Reconfiguration User Guide*. Xilinx Inc., 2010.