POLITECNICO DI MILANO
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

DESIGN METHODOLOGIES FOR DYNAMIC RECONFIGURABLE MULTI-FPGA SYSTEMS

Relatore: Prof. Donatella SCIUTO
Correlatore: Ing. Marco Domenico SANTAMBROGIO

Tesi di Laurea Specialistica di:

Alessandro Panella
Matricola n. 708496

ANNO ACCADEMICO 2008-2009

A mio nonno Felice,
maestro di vita.

When my blood stops
Someone else's will not.
When my head rolls off
Someone else's will turn.
And while I'm alive, I'll make tiny changes to earth.
[...] *You can mark my words, I'll make changes to earth.*

Frightened Rabbit, "Head Rolls Off"

Contents

List of Figures

Ringraziamenti

Prima di tutto ringrazio i miei genitori. Senza il loro aiuto niente di ciò che ha reso possibile il raggiungimento di questo traguardo sarebbe stato possibile. Non solo: il loro aiuto è stato condizione necessaria per cogliere ogni opportunità che mi si è presentata durante il corso degli studi, prima di tutte la possibilità di intraprendere un'esperienza di studio a Chicago che mi ha cambiato la vita. Grazie.

Un enorme grazie a Marco Santambrogio, colui che mi ha seguito e consigliato in questo lavoro di tesi che, insieme ad altre esperienze, ha contribuito a rafforzare una grande amicizia. Un sentito ringraziamento al relatore della tesi, prof. Donatella Sciuto, per l'interesse e il tempo dedicatimi.

Ringrazio di cuore ("che cuore") Massimo detto Panno, come amico e come fratello, insostituibile per svariati motivi. Auguro a lui e a Daniela una vita meravigliosa.

Un grande grazie al resto della mia famiglia, in particolare a mia zia Romanella, per l'affetto che sempre dimostra nei miei confronti.

Un grazie romantico va a Gaëlle, per essere semplicemente meravigliosa, e meravigliosa semplicemente. Ciò che ci lega è più forte della distanza.

Grazie a tutti i miei amici di San Lorenzo e dintorni. Non posso nominarvi tutti perchè siete troppi e mi dimenticherei qualcuno. Siete una risorsa insostituibile, mi avete dato "la ricarica" tutti i weekend, cene su cene su feste su feste su sbaitate su sbirrate su cocavanate! Aggiungo che mi mancate molto, i soci son soci.

Grazie agli amici del Politecnico e dintorni. Anche qui siete troppi e nominarvi tutti è impossibile. Grazie per evere condiviso con me esperienze di studio e divertimento. Una menzione particolare per Matteo e Malex, con i quali ho vissuto grandi momenti.

Grazie al mio rommate di Chicago, Alessandro detto Gugo, che ha intrapreso la mia stessa avventura, e con il quale si è creata profonda amicizia e grande intesa.

Grazie a tutti gli amici in quel di Chicago, italiani e americani, che rendono questa città ancora più splendida. Ringrazio in modo particolare Simone e Silvano, per tutti i momenti di studio e di svago insieme.

Questa tesi è dedicata a mio nonno Felice, scomparso recentemente. Credo di avere ereditato e imparato da lui ciò che maggiormente mi definisce come persona. È stato e sarà sempre per me un modello di vita: persona saggia, equilibrata, leale, benvoluta da tutti. Sono sicuro che vedermi al termine di questo cammino, e all'inizio di un altro ancora più lungo, l'avrebbe riempito di orgoglio.

Un affettuoso pensiero va a mia nonna Bambina, e a tutti i ricordi che mi legano a lei, che è stata un grande esempio di dedizione alla vita e all'amore materno.

Con amore,

Alessandro

Sommario

L'utilizzo di Field Programmable Gate Array (FPGA) è al giorno d'oggi un approccio largamente diffuso nello sviluppo di sistemi VLSI, sia nell'ambiente industriale che in quello della ricerca accademica. La capacità di essere riprogrammati un numero indefinito di volte dopo la produzione, unitamente all'esistenza di strumenti di sviluppo che automatizzano il processo implementativo, rende questo tipo di circuiti integrati molto flessibili ed economici. Negli ultimi anni, l'evoluzione delle architetture FPGA ha reso possibile un ulteriore incremento nel grado di flessibilità d'uso di tali chip. Questa innovazione è rappresentata dalla possibilità di riconfigurare alcune parti della FPGA durante la fase di esecuzione, mantenendo le altre parti attive, in modo tale che l'intero sistema non smetta mai completamente di operare. Questa tecnica è chiamata *riconfigurazione dinamica parziale*, e si affianca alla più comune *riconfigurabilità statica*, che viene effettuata in fase di compilazione.

Il potere computazionale delle FPGA può essere incrementato attraverso la creazione di *cluster* di chip, in modo da aumentare l'area disponibile e sfruttare la possibilità di computazioni parallele. Queste architetture distribuite sono denominate *sistemi multi-FPGA* (multi-FPGA systems), e vengono estensivamente utilizzate nell'emulazione della logica di circuiti custom e applicazioni di supercomputing. Nonostante diverse soluzioni per lo sviluppo di sistemi multi-FPGA sono state proposte in letteratura, la possiblità di incrementare ulteriormente il potenziale di queste piattaforme attraverso la riconfigurabilità dinamica non è

praticamente mai stata esplorata.

L'obiettivo di questo lavoro di tesi è lo sviluppo di un flusso per il progetto per sistemi multi-FPGA che abbia capacità aggiuntive rispetto a quelli esistenti. Due sono i contributi principali allo stato dell'arte. Il primo è uno sfruttamento reale della gerarchia di progetto durante il flusso di sviluppo. La ragione alla base di questa innovazione è che tale gerarchia è considerata una fonte di informazione molto importante, che può facilitare il processo di sviluppo e migliorare la qualità delle soluzioni fornite. Il secondo contributo è l'estensione del flusso di progetto oltre i limiti imposti dalle risorse fisiche disponibili: quando un'applicazione necessita di area maggiore rispetto a quella disponibile, il riuso di alcuni dei suoi componenti sfruttando la riconfigurabilità dinamica permette di ridurre l'area necessaria alla sua implementazione. Ciò causa inevitabilmente alcuni ritardi in fase di esecuzione, quindi è necessaria una soluzione per minimizzare questi tempi extra.

Il flusso di sviluppo per sistemi multi-FPGA proposto in questa tesi parte dall'estrazione automatica della struttura e gerarchia del progetto design dalla sua descrizione in VHDL. Le informazioni sono raccolte in una struttura dati appositamente creata, così che esse possano essere usate nelle fasi successive. A questo punto, una fase di layout globale tenta una implementazione statica dell'applicazione su una determinata architettura FPGA. Per quanto riguarda i passi di partizionamento e placement richiesti da questa fase, la tesi propone due soluzioni distinte. La prima proposta consiste in un algoritmo iterativo di tipo simulated annealing che effettua i due passi contemporaneamente. La seconda è un algoritmo di clustering di tipo bottom-up che sfrutta le regolarità presenti nell'applicazione da implementare, che vengono estratte dalla gerarchia di progetto, che viene poi seguito da un algoritmo di placement 1-a-1 sempre basato su simulated annealing.

A causa di vincoli di area e input/output, questo tentativo può fallire. In tal caso, un ulteriore algoritmo esplora l'albero gerarchico per l'identificazione di strutture isomorfe, ossia occorrenze ripetute di uno stesso pattern di blocchi di design.

Tali strutture possono essere riutilizzate nell'applicazione in modo da risparmare area. Siccome il riutilizzo di parti del design richiede del tempo aggiuntivo per la riconfigurazione parziale dei blocchi di interconnessione, l'uso di un modello ILP permette di trovare la soluzione di riutilizzo delle parti che minimizza il tempo di esecuzione, effettuando scelte come la selezione di pattern ricorrenti da riutilizzare.

Di seguito viene descritta l'organizzazione della presente tesi.

Il **capitolo 1** introduce le definizioni e i concetti basilari che costituiscono il contesto di questo lavoro di tesi. Viene descritta la tecnologia FPGA, e vengono presentate architetture e applicazioni per sistemi multi-FPGA. Successivamente, il concetto di riconfigurabilità dinamica viene introdotto, per concludere con le motivazioni e possibili scenari implementativi per sistemi multi-FPGA dinamicamente riconfigurabili.

Il **capitolo 2** costituisce un breve compendio di alcuni lavori relativi ai sistemi multi-FPGA che si trovano in letteratura. Vengono descritti e analizzati diiversi algoritmi di partizionamento e flussi di sviluppo. Due di questi ultimi costituiscono dei flussi completi, mentre altri approcci forniscono solo una soluzione a una o più fasi dello sviluppo. Alla fine del capitolo, le soluzioni discusse vengono confrontate tra loro.

Il **capitolo 3** descrive il flusso di progetto di sistemi multi-FPGA proposto in questa tesi e le prime due fasi che lo compongono. Dapprima, vengono spiegati i principi del progetto gerarchico e la fase di estrazione del progetto. Successivamente, viene descritta la fase di layout globale e vengono proposti due approcci per questo problema.

Il **capitolo 4** è dedicato alla descrizione della nuova metodologia per il riutilizzo di blocchi del progetto proposta da questo lavoro. Dapprima vengono introdotte le motivazioni e definizioni basilari, e il problema affrontato viene definito formalmente. Dopo una descrizione dei possibili scenari architetturali, si intro-

duce la metodologia proposta, separandola in due fasi: il riconoscimento delle strutture isomorfe e l'effettuazione delle scelte di riuso.

Il **capitolo 5** spiega come le soluzioni esposte nei due precedenti capitoli sono state implementate. L'esposizione è strutturata seguendo il flusso metodologico: dapprima viene descritto il framework per l'estrazione del design, seguito dalla implementazione della fase di layout globale. Successivamente sono presentati gli algoritmi per il riuso di parti dell'applicazione.

Il **capitolo 6** descrive l'attività di verifica sperimentale che è stata svolta e fornisce i risultati ottenuti. Dapprima, sono presentati i risultati relativi all'estrazione del design. Successivamente, i due approcci al layout globale vengono confrontati, per poi fornire i risultati relativi alla fase di riuso dei blocchi. Infine, il funzionamento del flusso di sviluppo proposto viene esemplificato tramite un caso di studio.

Il **capitolo 7** conclude la tesi e traccia la strada per future espansioni e miglioramenti del presente lavoro.

Summary

The use of Field Programmable Gate Arrays (FPGAs) is nowadays widely spread both in VLSI industry and academic research. The capability of being reprogrammed an indefinite number of times after having been produced, together with the existence of design frameworks that automate the implementation process, makes such type of integrated circuits very flexible and cheap. In recent years, the evolution of FPGA architectures has made it possible to further increase the degree of flexibility in the use of such chips. This innovation is represented by the possibility of having parts of the FPGA reconfigured at run-time, while others are still running, so that the execution of the whole system is never stopped. Such technique is called *partial dynamic reconfigurability*, as opposed to the standard *static reconfigurability*, which is applied at compile time.

The computational power of FPGAs can be increased by the creation of clusters of chips, in order to augment the available silicon area and exploiting parallel computation. Such distributed architectures are referred to as *multi-FPGA systems*, and are largely used in logic emulation of custom circuits and supercomputing applications. Although several solutions to cope with the design of multi-FPGA systems have been proposed in literature, the possibility of further increase the potential of these platforms through dynamic reconfigurability has never been explored.

The aim of the present thesis work is to develop a multi-FPGA design flow with expanded capabilities with respect to the existing ones. There are two main

contributions to the current state of the art. The first one is the strong exploitation of the design hierarchy throughout the design flow. The rationale behind this is that the design hierarchy is considered a very useful source of important information, that can ease the design process and improve the quality of the solutions. The second one is the extension of the multi-FPGA design workflow beyond the limit imposed by the available physical resources: when an application does not statically fit in a given multi-FPGA architecture, the reuse of hardware components through dynamic reconfigurability is exploited to reduce the area necessary to implement the application. This inevitably introduces some extra time delays in the execution, therefore a solution to minimize such time is provided.

The proposed multi-FPGA design workflow starts from the automatic extraction of the design structure and hierarchy out of the VHDL description of the application. The retrieved information is captured by a specifically designed data structure, in order to be used by the subsequent phases. Then, a global layout phase attempts a static implementation of the application on a given multi-FPGA architecture. Two solutions are proposed and evaluated for the partitioning and placement tasks. The first one is a simulated annealing algorithm which simultaneously performs the two tasks. The second one is a bottom-up clustering algorithm that exploits the regularities in the application which are retrieved from the design hierarchy, followed by an annealing-based 1-to-1 placement.

Due to area and I/O pin constraints, this attempt may fail. In this case, an algorithm explores the hierarchical tree for the identification of isomorphic structures, that are repeated occurrences of the same pattern of design blocks, which can be therefore reused in the application in order to save area. Since the reuse of portions of the application requires some extra time for dynamically reconfiguring the interconnections between blocks, an ILP model is used to find the blocks reuse solution which minimizes the execution time of the application, by operating choices such as the selection of the recurrent patterns which may be reused.

In the following, the organization of the present document is described.

Chapter 1 introduces the basic definitions and concepts which constitute the context of this thesis work. The FPGA technology is described, and multi-FPGA architectures and applications are presented. Then, the dynamic reconfigurability of FPGAs is described, before providing the rationale and possible scenarios for multi-FPGA dynamic reconfigurable systems.

Chapter 2 constitutes a brief survey of some of the works related to multi-FPGA systems that can be found in literature. Different partitioning techniques are described, and multi-FPGA flows are analyzed. Two of them constitute complete flows, while other approaches only provide the solution to one or more phases of the design. In the end of the chapter, a comparison of the discussed solutions is carried out.

Chapter 3 describes the multi-FPGA design workflow proposed in this thesis and the first two phases that compose it. First, the hierarchical design principles and the design extraction phase are explained. Then, the global layout phase is described and two approaches are proposed, the first one based on simulated annealing and the second one based on bottom-up clustering.

Chapter 4 is dedicated to the description of the novel design blocks reuse methodology proposed in this thesis work. Definitions and motivations are provided, and the addressed problem is formally defined. After the description of the architectural scenarios, the methodology for coping with the stated problem is introduced by splitting it in two blocks: finding the isomorphic structures and carrying out the reuse choices.

Chapter 5 explains how the solutions proposed in the two previous chapters have been implemented. The description is structured according to the methodological workflow. Therefore, first the design extraction framework is described,

followed by the implementation of the global layout tasks. Then, the blocks reuse algorithms are presented.

Chapter 6 describes the experimental verification activity that has been carried out and provides the obtained results. Evaluations and comparisons are provided as well. First, the results provided by the design extraction are presented. Then, the results for the two proposed global layout approaches are shown and compared. Subsequently, the results relative to the blocks reuse algorithms are provided. In the end of the chapter, a case study provides an example of use of the design flow in a practical scenario.

Chapter 7 concludes the work and provides ideas for future extensions and improvements.

Chapter 1

Context and problem definition

This preliminary chapter is an introduction to the work presented in this thesis. Since the addressed problem can be considered as a part of a complex context, this introduction is divided in several sections, each one dealing with a different but related sub-field. First, Section 1.1 presents the device that constitutes the physical ground for every solution taken into account successively, which is the Field Programmable Gate Array (FPGA). Then, Section 1.2 introduces multi-FPGA systems, in which more than one such devices are used together. FPGAs represent the most common technology to realize reconfigurable hardware systems. Since the notion of reconfigurability is central to this work, it is introduced in Section 1.3, along with an explanation of different types of reconfigurability using Xilinx FPGAs. Particular attention is paid to dynamic reconfigurability, which is the field where this thesis project takes place. In Section 1.4, the problem addressed by this work, which deals with dynamic reconfiguration of multi-FPGA systems, is stated. The chapter ends with a brief description of the organization of the present document.

1.1 Field Programmable Gate Arrays

The last two decades of VLSI design have seen a continue growth in the use of programmable devices. Among them, Field Programmable Gate Arrays (FPGAs) have shown to be the most promising technology for a large variety of applications. In particular, FPGAs are usually preferable to Programmable Logic Devices (PLDs) because they can implement multi-level logic. With respect to Mask Programmable Gate Arrays (MPGAs), FPGAs do not have to be custom fabricated, thus significantly lowering costs and times for low-volume circuit productions [1].

The peculiarity of FPGAs rises from the fact that they are reprogrammed by the user, a potentially infinite number of times. On the contrary, MPGAs have to be programmed - once and for all - at the end of the production process, still in-factory. FPGAs can be seen as occupying the rightmost place on an ideal classification which goes from full-custom technology to full-reprogrammable technology, presented in Figure 1.1. Obviously, the aforementioned advantages come at the expenses of speed and area occupation. As a matter of fact, the speed and the integration density of the realized circuits decreas going from left to right in our graphical taxonomy. However, only very few applications are worth the efforts of creating a specific custom solution. This derives from a simple observation: custom designs have extremely high NRE (nonrecurrent engineering) costs. This implies that a custom implementation is worth only for very large volumes of production.

The basic structure of a FPGA chip is conceptually simple: it is constituted by a matrix of reconfigurable blocks and a routing architecture that permits arbitrarily point-to-point communication. Moreover, it has a set of configurable I/O blocks along its perimeter. This basic structure is showed in Figure 1.2. FPGA technology was introduced in 1984 by Xilinx, Inc., which is currently the main producer and vendor. The structure of Xilinx FPGAs is briefly exposed in the following, taking as example the Xilinx Virtex FPGA family [2]. The smallest

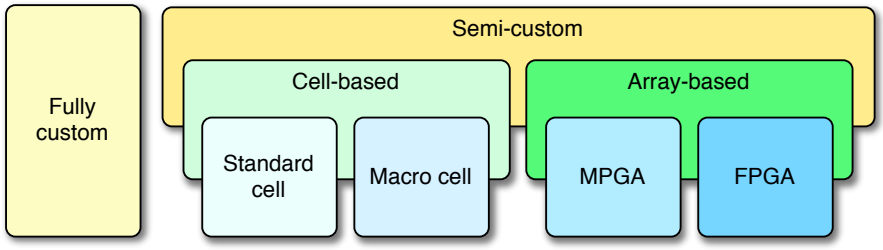


Figure 1.1: Integrated Circuit technology classification.

building unit is constituted by the Reconfigurable Logic Block (CLB). In Xilinx FPGAs, CLBs are constituted by two identical *slices*, each containing two *Logic Cells* (LC). A *Look-Up Table* (LUT), along with a multiplexer and a flip-flip, is contained in each LC. Depending on the model of the FPGA, LUTs can have four inputs or more. LUTs allow arbitrary combinatorial functions of their inputs to be created. The scheme of the Xilinx Virtex CLB is showed in Figure 1.3. Besides this common basic structure, different FPGAs can have different features: to give an example, Xilinx Virtex-2 Pro FPGAs embed an hard IBM PowerPC 405 general purpose processor.

The chip is programmed by changing the function carried out by reconfigurable blocks. The programming process makes use of a *bitstream*, which is a sequence of bit containing the information about how to configure the chip. Obviously, the user does not need to manually program *each* CLB: several design tools exist, which take as input a description of the system to be implemented at different levels of abstraction - from low-level circuit netlists to high-level HDL descriptions -, synthesize it and produce the configuration bitstreams. Xilinx ISE (Integrated Software Environment) is an example of such tools [3].

FPGAs are used for a large variety of applications. VLSI industries exploit them for realizing prototypes of custom ASIC circuits, as they provide a solid, cheap, and reliable instrument for testing and emulating circuits. Besides being used as a step in the design flow, they can constitute a high-performance hardware

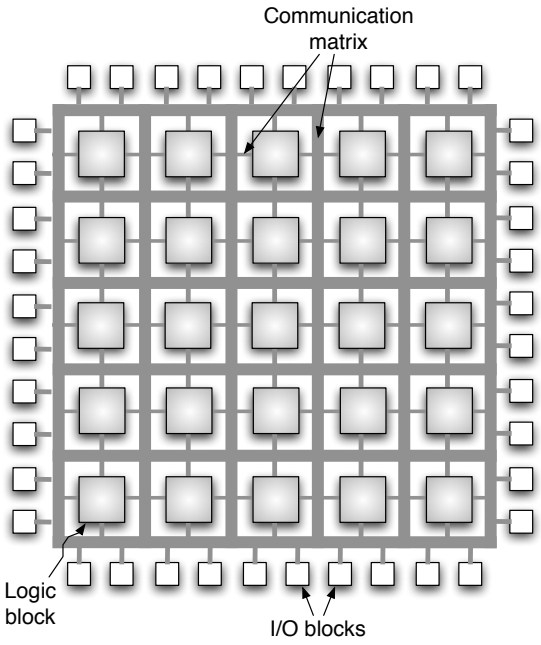


Figure 1.2: Basic structure of a FPGA.

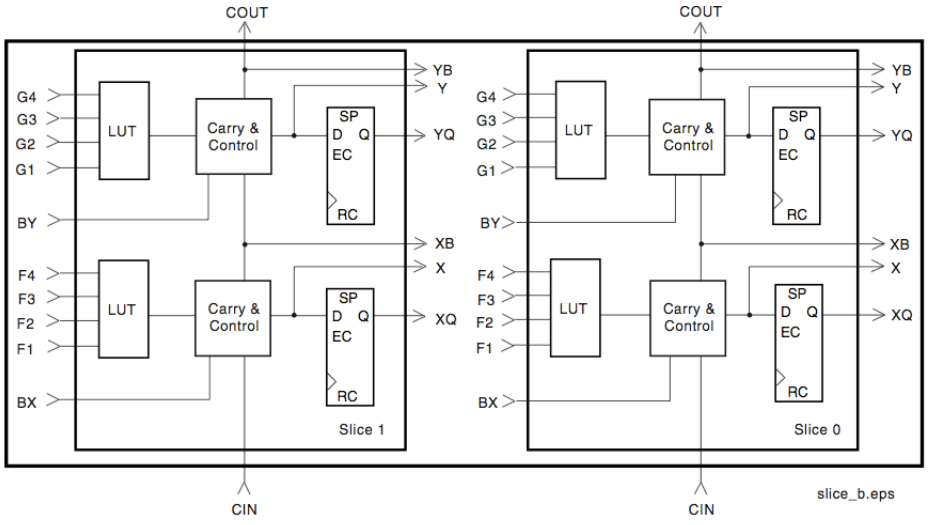


Figure 1.3: Xilinx Virtex Configurable Logic Block schematic, from [2].

platform by themselves: this is the case of big FPGAs clusters for large parallel computations. Another use of FPGAs is targeted to the final user, who can implement on his own a real hardware system for his needs. Besides all this variety of uses, a relatively new field in which FPGAs are suitable to be adopted is embedded systems, where they can be used to implement application specific hardware components.

It is possible to distinguish two main classes of FPGA usage: solutions where one or more FPGAs are used as application specific coprocessors, to which a general purpose CPU demands some particular tasks, and solutions in which the FPGAs themselves constitute complete System-on-Chips. A further distinction is between FPGA systems which implement an entire application and ones in which FPGAs performs only some of the tasks, while the others are carried out using software that runs on a general purpose processor. In this latter case we speak of *Hardware/Software codesign*, that frequently represents the best methodology to obtain good cost-quality trade-offs.

1.2 Multi-FPGA systems

The incessantly growing demand of computational power has been traditionally addressed by increasing the operating frequency of processors and by exploiting parallelism at the instruction level. This has led to the creation of very powerful stand-alone computing units, mainly general purpose processors, which have been able to offer satisfiable performance for almost all applications. However, these methods have been being abandoned by the semiconductor community, mainly due to their design complexity and effectiveness-cost trade-offs [4]. A new approach, inherited by supercomputing applications, is becoming the dominating design paradigm. The key feature is to exploit a coarser-grain level of parallelism, by splitting applications in a number of concurrent tasks to be assigned to several computational units. These can be either general purpose CPUs, usually situated

on the same chip, or application specific hardware components. FPGAs represent a suitable means to implement this latter solutions: such an ensembles of chips is called *Multi-FPGA System* (MFS). The main advantage is the same as the single FPGA case: a specific custom solution is usually not worth, and reprogrammable logic provides sufficient computational speed with low design costs. Several research works have investigated the potential of multi-FPGA systems, and tens of specific multi-FPGA architectures have been designed in the last 15 years; a brief review of some solutions can be found in [1]. In order not to generate ambiguities in the remainder of this work, the physical cluster of FPGAs will be referred to as the *architecture*, while the conjunction of the architecture and the application which is implemented on it will be indicated as a multi-FPGA *system*.

1.2.1 Multi-FPGA architectural topologies

One of the main features which distinguishes different kinds of multi-FPGA architectures is the topology, meant as the way the different FPGAs are connected one each other. The first considered distinction is between *hard-wired* and *programmable* connections.

As the name suggests, hard-wired connections are fixed wires that link the I/O pins of two FPGAs, and are usually implemented as physical tracks on a multi-FPGA board. Several topologies adopts hard-wired connections. In a *complete-graph* topology, each FPGA is connected to each other. Despite this topology offers a direct connection between any pair of FPGAs (i.e. without using intermediate devices), it has the following simple but crucial drawback: as the number of chips increases, the width of each connection decreases (since the number of available pins on each FPGA is finite and fixed). Moreover, a complete graph topology is generally not planar, which means that some of the interconnecting tracks are incident one each other, imposing remarkable limits to the circuit practical realization. It is clear that it is impossible to implement architectures with a

high number of chips completely interconnected. The most widely adopted hard-wired topology is the *mesh*. In a mesh, the chips are disposed on a grid, and are connected in a nearest-neighbor pattern. We further distinguish between *4-way* meshes, in which the wires connect only horizontal and vertical neighbors, and *8-way* meshes, where each FPGA is connected also to its diagonal neighbors. When the FPGAs on two opposite boundaries of the grid are connected in a circular fashion, the topology is named *torus*. A mesh is usually specified through the dimensions of the grid: in a n x m mesh, $n*m$ FPGAs are disposed in a n x m matrix. When one dimension is equal to one, we speak about *linear arrays*. An example of 4-way mesh is depicted in Figure 1.4(a). The advantage of the mesh topology is the inherent expandability of the architecture, due to the use of local connections. As a matter of fact, adding an FPGA to an existing architecture means creating some local connections without any other constraint. The disadvantages are due to the fact that there is no a fixed-length path between every pair of FPGAs. This causes different delays in signal transmission and the need to use some area to implement communication logic in intermediate chips.

Programmable connections consist of wires connected to reprogrammable components. Such components can be programmed in order to implement a particular connection among the incident wires. The most used topology using programmable connections is the *crossbar*. In this topology, chips are divided in two classes on the basis of their functionality. *Logic bearing* FPGAs contain the logic functions and perform computations (the lower ones in Figure 1.4(b)), while *routing* chips provide the connections between logic chips (the upper ones in Figure 1.4(b)). The idea is that communication between any pair of logic FPGAs requires exactly one extra routing hop, such that communication delays are all equal. When only one chip is used to provide the interconnections, the crossbar is said to be *total*. When several chips are used, the topology is named *partial* crossbar. Due to the cost of producing a big routing chip, partial crossbar is usually preferred. The routing chips can be either standard FPGAs or cheaper reprogrammable devices,

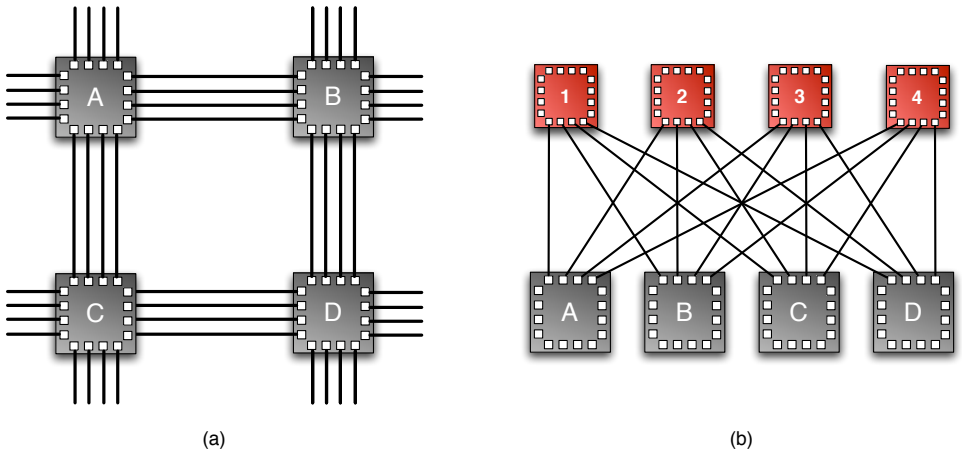


Figure 1.4: Mesh (a) and Crossbar (b) multi-FPGA topologies, adapted from [5].

such as FPIDs (Field Programmable Interconnection Devices), otherwise called FPICs (Field Programmable Interconnections Chips). Crossbar architectures have the drawback that they are not expandable, since the connections are implemented over a global communication infrastructure. A crossbar topology variation topology attempts at solving this problem: in *hierarchical crossbar* (Figure 1.5) crossbars are organized in levels so that the architecture is easily expandable. However, this last solution implies varying - even if easily computable - delays.

Specific topologies used in real multi-FPGA architectures are often the result of variations and combinations of these two basic approaches. In [6], Khalid proposes three types of such *hybrid* architectures. These modifications of the basic partial crossbar aim at taking advantage of the locality of inter-FPGA connections. The *Hybrid Torus Partial-Crossbar* (HTP) architecture consists of a set of FPGAs which are connected both through hard-wires in a 4-way torus topology and through a partial crossbar (Figure 1.6(a)). Some of the FPGA pins are assigned to hard-wired connections, while others are used mapped on programmable connections. In the *Hybrid Complete-Graph Partial-Crossbar* (HCGP) the FPGAs are connected through a partial crossbar and by means of a complete-graph hard-

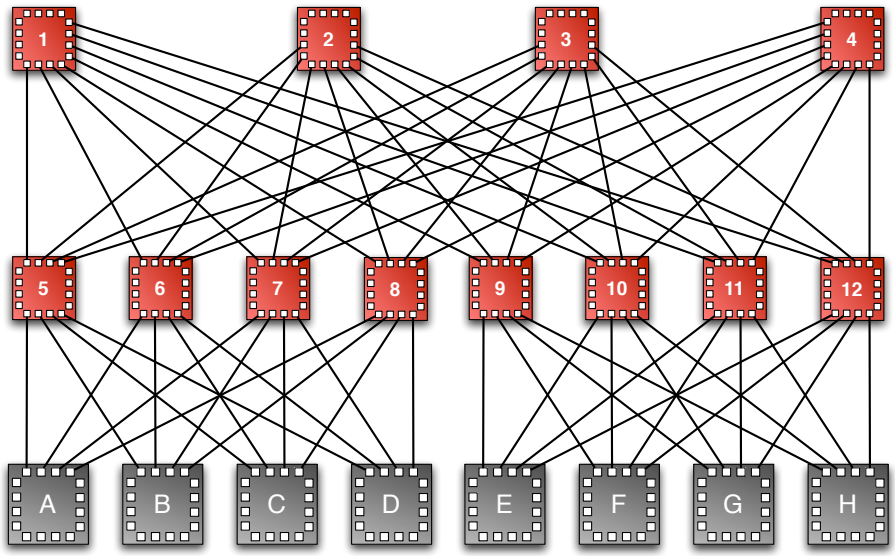


Figure 1.5: Hierarchical crossbar multi-FPGA topology, adapted from [5].

wired interconnection structure (Figure 1.6(b)). In the *Hardwired-Clusters Partial Crossbar* the FPGAs are divided in clusters. Within each cluster, FPGAs are connected through a complete-graph topology. Clusters in turn connected one each other using partial-crossbar programmable connections (Figure 1.6(c)).

The topologies presented so far have a common trait: they are all implemented over *dedicated* wires, in the sense that once a physical (hard or reprogrammable) wire is assigned to a particular logic net, its use is reserved only to that net. Even if few approaches take it into account, another possibility is to used *shared* wires. In this case, more than one net can be mapped on a single physical wire. *Virtual Wires* [7] represents an interesting methodology for sharing physical wires: multiple logical I/O ports are mapped on a single physical FPGA I/O pin. The usage of such pin is controlled on a time-sharing basis: a schedule which works at the highest possible FPGA clock frequency determines when a physical pin is actually assigned to a given logical I/O port. Shared wires can be utilized also through address mapping: a simple example is a bus topology. In this case, each

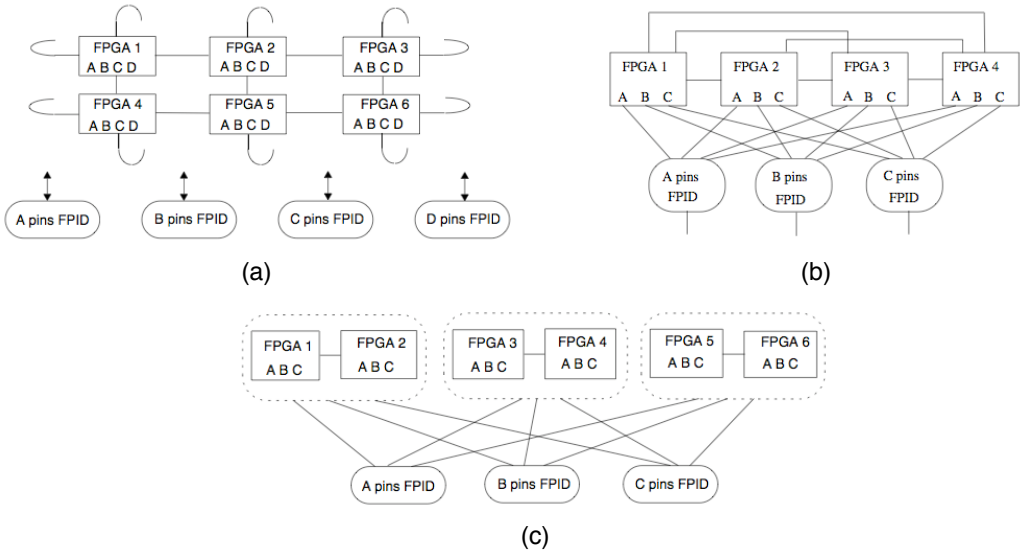


Figure 1.6: Multi-FPGA hybrid topologies: HTP (a), HCGP (b), and HWCP (c). From [6].

FPGA connected to a common bus and is provided with a communication infrastructure, able to handle buffers and all the features required by the particular bus. Every FPGA is assigned to an address range, and the I/O ports of the logic modules contained in each FPGA are associated with a particular address value. Another similar solution is the use of point-to-point shared connections: wires of fixed width connect pairs of FPGAs, using a communication protocol which allows the different I/O ports to use the unique communication channel. Although being undeniably slower than dedicated connections, these methodologies offer a high flexibility, in particular allowing modules to be dynamically added to and removed from the system. An example of a multi-FPGA bus topology is found in [8] and will be discussed in Section 2.2.

1.2.2 MFS applications

In this section some interesting applications of MFS's are briefly reported. It has been already mentioned that, in supercomputers, clusters of application specific circuits perform better than general purpose CPUs, since they are fitted on the specific computations that must be executed. However, the high design costs usually forbid the development of custom ASICs. For this reason FPGAs constitute a good trade-off between costs and performance.

Another field in which MFS's s have been extensively adopted is logic emulation in the testing of custom circuits. The circuit to be tested is implemented on a multi-FPGA architecture and is debugged and validated through many different runs which must cover all the possible faults[1]. Reprogrammable hardware provides a middle-ground between software simulation and prototyping, mitigating the drawbacks of both approaches. With respect to software simulation, it is faster by some orders of magnitude. Like simulation, emulation allows to isolate and easily correct possible bugs. Since the realization of hardware prototypes is really expensive, emulation offers a cheaper approach. However, emulation is not suitable for delay testing, since the timing of the MFS is different from the final circuit's one. Therefore, a common approach is to adopt logic emulation for fault testing, and successively prototyping for timing debugging.

MFS's have been recently used to implement neural networks: the reprogrammability of such architectures offers a suitable means to implement learning processes. Evolvable systems like neural networks can exploit the capability of FPGAs of having some circuit functionalities modified over time without interrupting the overall execution.

[1] Notice that we are referring to design testing and not circuit production testing.

1.3 Reconfigurable computing using FPGAs

So far FPGA technology and multi-FPGA systems have been presented. In this section, the focus shifts from the architectural to the methodological domain: the notion of *reconfigurability* applied to FPGAs is introduced and investigated, both in the single- and multi-FPGA cases.

The notion of a *reconfigurable computer* has been introduced in the 1960s, when Herald Estrin conceptualized a machine composed by a standard processor and an array of reconfigurable hardware elements [9]. The idea was to have these elements configured for the execution of a particular task, and then *re*configured when a new task has to be executed. It is possible to define reconfiguration as the process of altering the location or the functionality of a system element, as a response to faults, changes in the environment or explicit application needs.

FPGA technology is nowadays the most suitable way to implement hardware reconfiguration. As a matter of fact, their programmability can be naturally viewed as the *physical* counterpart of the *operational* definition of reconfiguration given above. Therefore, from this point on we will intend reconfiguration as a process which involves one or more FPGAs, that constitute the architecture which is reconfigured. There exist several types of reconfiguration; in the following, a classification of reconfiguration methods is provided, as described in [10].

- *Who controls the reconfiguration and where the reconfigurator is located.* When the reconfiguration controller and the reconfigurator are inside the boundaries of the FPGA, the reconfiguration is said to be *internal* (Figure 1.7(a)). On the contrary, when the reconfiguration is handled from a PC or any processor outside the FPGA, the reconfiguration is *external* (Figure 1.7(b)). Even if the distinction is clear in the case of a single FPGA, more attention has to be paid to the multi-FPGA case. In this situation, if one or more FPGA chips contain the reconfiguration units, the reconfiguration is internal for that chips, external for the other ones and internal if we consider

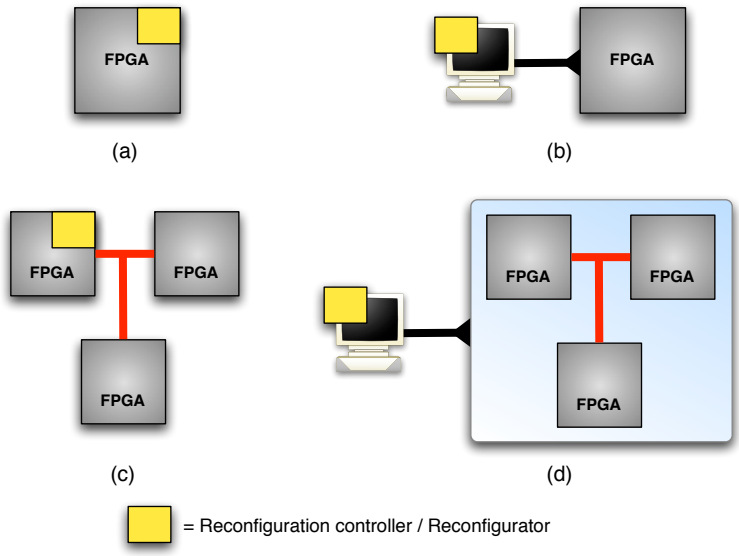


Figure 1.7: Internal vs. external reconfiguration.

the whole MFS (Figure 1.7(c)). The reconfiguration is truly external when it is handled by a unit outside the MFS (Figure 1.7(d)).

- *When the configurations are generated*. Three scenarios are possible: in the first case, all the possible configuration of the whole system at a given moment are generated at design time (*static way*). The second method is *run-time placement of previously created modules*: the modules which take part in the reconfiguration process are generated at design time, but the decision about when and where to insert them into the system is taken at run time. The *fully dynamic* case is intended as the synthesis of modules at run-time. Although this would be the most flexible solution, it is unfeasible because of the time needed for the modules' synthesis.

- *Which is the granularity of the reconfiguration*. The first distinction is between *complete* and *partial* reconfiguration. In the first case, the FPGA is entirely reconfigured (Figure 1.8(a)). In the second case, only a portion of

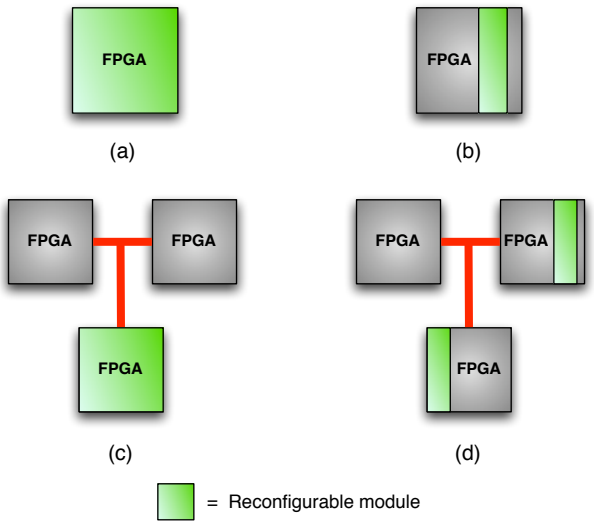


Figure 1.8: Total vs. Partial reconfiguration.

the chip is take part in a reconfiguration (Figure 1.8(b)). This latter case can be further subdivided: on Xilinx FPGA, two partial reconfiguration methodologies are possible [11]. *Small-bit manipulation* is the reconfiguration of very small portions of the system (single CLBs). On the other hand, *module-based reconfiguration* involves hardware blocks, called modules. Even this case needs a disambiguation in a multi-FPGA scenario. Reconfiguration can be total for a single chip but partial if considering the whole MFS (Figure 1.8(c)). In case it is partial for a single FPGA it is necessary partial for the entire MFS, too (Figure 1.8(d)).

From what has been said so far, it is possible to see that FPGAs are really flexible devices, suitable for implementing a large variety of systems and applications. In recent years, a further degree of flexibility has come to light: the possibility of having a portion of the chip reconfigured, while the other part does not interrupt its execution. The implementation of this new scenario is achievable on a single FPGA chip only if the reconfiguration is partial. Therefore, we speak about *partial*

dynamic (or *run-time*) *reconfiguration*. There are many possible applications for this new reconfiguration paradigm. To cite some, it is suitable for implementing applications which require more space than the area available on a FPGA, even if there are some functional units which are not active at the same time. These units can occupy the same place on the chip at different instants of time, through partial dynamic reconfiguration. Another possible application is in presence of changing requirements: suppose to have a small device which is asked to carry out many different tasks in a short time (this could be the case, for instance, of a modern mobile phone). On the basis of what is asked at a given moment of time, the hardware can be reconfigured to accomplish a specific task in a small amount of time.

1.3.1 DRESD and the Earendil flow

DRESD (Dynamic Reconfigurability in Embedded System Design) is an active research project in the Electronic and Information Department at Politecnico di Milano. Its global aim is to provide a complete and flexible design flow for dynamic reconfigurable systems, paying particular attention towards embedded applications. It is composed by several subprojects and branches. The general workflow currently under development is called *Earendil*, which is depicted in Figure 1.9.

Earendil is a complete specification-to-bitstreams HW/SW codesign workflow which possibly uses third-party software. The target architecture, whichever the physical device is (e.g. a single particular FPGA, a cluster of FPGAs, etc.), will be composed by a *static* part and a *dynamic* part. The static pars contains some modules which are not reconfigured, such as a hard or soft general purpose processor and the reconfiguration controller. The dynamic part is the reconfigurable area, in which blocks can be added and removed. The Earendil workflow is composed of three main phases. The first one is called High Level Reconfiguration

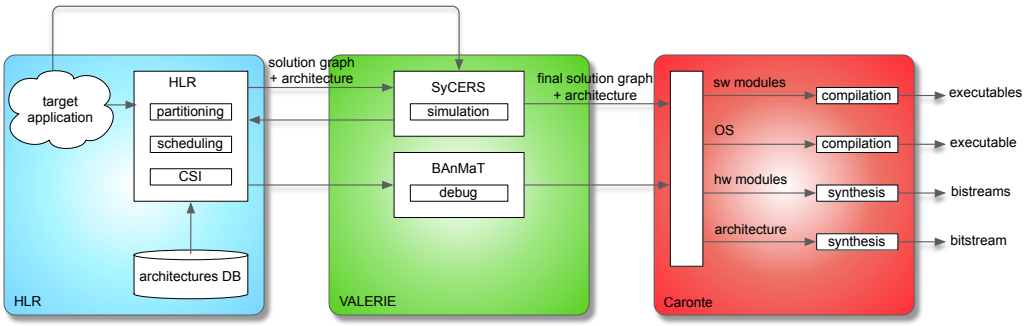


Figure 1.9: DRESD project: the Earendil workflow.

(HLR): during this phase the input specification is partitioned into modules (both hardware and software ones) and scheduled. The result is passed to a Validation (VAL) phase, which has the aim of validating the proposed solution through a simulation. The output of this phase is taken as input by the Low Level Reconfiguration (LLR) step, which has the goal of synthesizing the architecture and the hardware reconfigurable modules, along with software components and possibly adding an Operating System support.

1.3.2 Multi-FPGA systems and dynamic reconfiguration

In Sections 1.2 and 1.3 MFS's and dynamic reconfigurability have been treated as two separated topics. In this section the two concepts are merged, by motivating our interest in such solutions and outlining some possible scenarios of dynamic reconfigurable MFS's.

1.3.2.1 Rationale

The reason why dynamic reconfiguration is by itself an interesting and promising field lays essentially in two arguments. First, it represents a method to cope with the large demand of computational resources on small devices: a temporal scheduling and partitioning of the application to be implemented, possibly along

with HW/SW codesign techniques, causes the *virtual* area to be more than the *actual* one. Second, it embodies a solution to evolving and mutable specifications: a dynamic reconfigurable chip is able to react to even frequent changes in the application needs.

The use of multi-FPGA apparently addresses the first of these issues. Nevertheless, the available area, even if large, could not be anyhow enough to fit big applications. For this reason, dynamic reconfiguration applied to MFS provides even more computational area. This fact can be considered also from another point of view: even if the application to be implemented would fit in a reasonable number of FPGAs, by exploiting dynamic reconfiguration it would be possible to optimize the number of chips, and therefore produce cheaper systems. The second point, regarding changes in requirements, continue to have the same validity in the MFS case as in the single FPGA one.

1.3.2.2 Possible scenarios

In Section 1.3 it has been stated that, in order to be performed at run-time, reconfiguration must be *partial* in the single-FPGA case. On the other hand, several scenarios can be outlined in the multi-FPGA case, because "partial" has multiple meanings. We do not pretend to provide an exhaustive taxonomy of dynamic reconfigurable MFS's; the intent is just to provide some guidelines for classifying these systems and consequently develop methodologies to cope with different situations.

First, it is possible to identify two basic kinds of reconfiguration in MFS's: one is when the reconfiguration is applied to *logic* chips, while the second is when it is applied to *routing* chips. Clearly, this latter solution can be considered only for MFS topologies with programmable interconnections (Section 1.2.1). As a matter of fact, it is possible in such architectures to dynamically reconfigure the routing channels among the components of the systems, thus permitting the reuse of a single module by different components. This possibility will be further explored

in Chapter 5. Clearly, both kinds of reconfiguration can be simultaneously applied on the same MFS; although this latter option seems complex, it is clear that it provides a very high flexibility of the system, which becomes entirely modifiable.

Moreover, it is possible to make distinctions on the basis of the granularity and the location of the reconfiguration controller, as explained in the following:

- The simplest scenario is the use of an external CPU which controls the reconfiguration of the multiple FPGAs. The dynamic reconfiguration can be either total or partial with respect to each single FPGA: as a matter of fact, one FPGA could be totally reconfigured while the others are still operating, thus never stopping the execution flow.

- The second scenario taken into account is reconfiguration *internal* with respect to the MFS. In such case, two situations are possible, described in the following.

  - One FPGA acts as a *master* and controls the reconfiguration - either total or partial - of the other chips. The master FPGA manages the reconfiguration by handling bitstreams to reprogram the other chips.

  - The control of the reconfiguration is *distributed* over the multiple FPGAs. This case implies partial reconfiguration of each chip and involves more communication and coordination issues than the previous case.

The borders between these categories are surely not neat, as many hybrid and complex solutions are not captured by this scheme. However, this classification helps in providing an idea on how a dynamic reconfigurable MFS can be implemented in practice.

1.4 Thesis goals

In the previous sections the notion of dynamically reconfigurable multi-FPGA system has been gradually introduced. Since this is a relatively new research branch and therefore lacks of real effective design approaches, we propose in this work new solutions for the development of such systems. It may be evident that dynamic reconfigurability represents a sort of expansion to the capabilities of static MFS's. Hence, we first address the problem of providing a workflow for traditional MFS's, which is asked to offer a suitable basis for an expansion towards dynamic reconfigurability. Once this design flow has been shaped, we introduce novel methodologies that permits the dynamic reconfiguration applied to MFS in several forms.

Chapter 2

Background and related works

This chapter describes the existing approaches to problems that are related to the one considered in this thesis. First, in Section 2.1 the general design flow for MFS's is described. Some previous general approaches to the challenging partitioning problem are presented and some existing MFS design tools are introduced. Section 2.2 describes the few existing methodologies to the specific problem we address in this work, that is the design of dynamic reconfigurable multi-FPGA systems.

2.1 Multi-FPGA Systems design flow

The design and use of multi-FPGA systems has been being widely investigated for more than a decade. For this reason, different MFS CAD tools and architectures can be found in literature. Intuitively, the differences between the design flow of single-FPGA systems and MFS's lays in the fact that the latter requires an additional global layout synthesis phase. A general MFS synthesis flow can be thought as composed of two main macro-steps. First, the design need to be split into several parts, each of one assigned to a particular FPGA: this is the *global layout synthesis*. After that, each part must be implemented onto the assigned

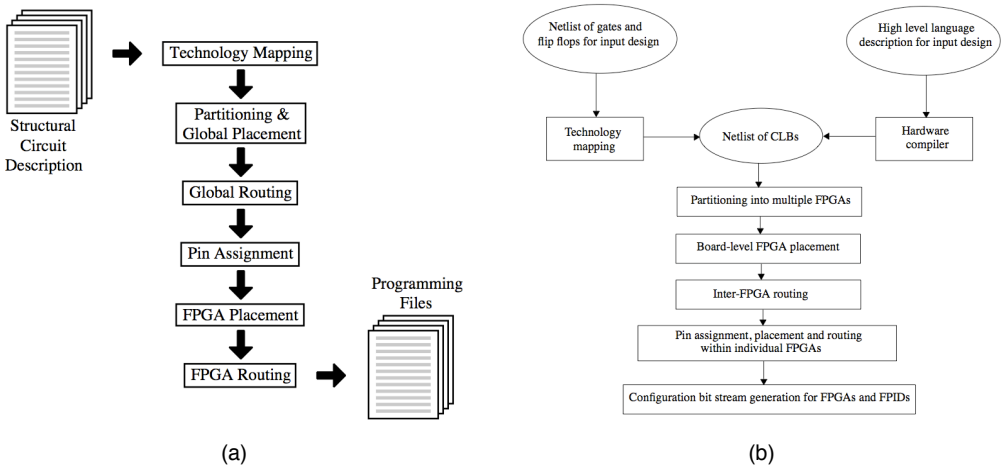


Figure 2.1: Two similar flows for MFS design. (a) is from [5], (b) is from [6].

FPGA; this phase, which we call *local layout synthesis*, is identical to the single FPGA case.

In [5] and [6] the problem of MFS design is faced extensively. Although the two works have different aims, they both propose a similar design flow. Hauck's work [5, 1] has the goal of innovating MFS CAD tools by proposing novel algorithms at different stages of the design flow. Additionally, his work represents a bird's eye view on the state of MFS research at the time it was written, thus providing a solid background in the field. On the other hand, Kahlid [6, 12] developed a MFS synthesis chain in order to provide support to the exploration and comparison of different multi-FPGA architectural topologies. As previously hinted, the two flows share almost all aspects, with the only remarkable difference being that Kahlid's design takes into account the possibility of having high-level languages input specifications. The two flows are depicted in Figure 2.1.

One concern raises from the presence of an early *technology mapping* step. As defined in [13], technology mapping is the transformation of a gate-level netlist using components of the target device's library, usually referred to as cells. In the case of FPGAs, there is only one cell, that is the LUT. This is a non-trivial task,

which strongly depends on the particular hardware adopted, and tools to cope with it are usually provided by chip vendors. As pointed out by Khalid in [6], an alternative is to perform FPGA technology mapping later in the flow, obtaining the advantage of speeding up the process since it can be performed - possibly in parallel - independently for each chip. However, the author claims that such choice forbids to carry out proper timing-driven layout synthesis, because of the lack of accurate implementation information. This issue will be briefly dealt with in Chapter 3.

However, it is obvious that in every MFS design chain a global layout phase must precede chip-by-chip local synthesis. This is accomplished by means of three steps: global *partitioning*, *placement*, and *routing*. Partitioning has the goal of dividing a circuit into the minimum number of parts, each one corresponding to a FPGA, such that the total number of connections between these parts (called *cutsize*) is minimized. Partitioning over multiple FPGAs is subject to dimension and pin limit constraints: each sub-circuit must fit in one FPGA and it must be possible to address the I/O requirements of each part using the pins available on each chip. Placement is the task of assigning each part to a specific FPGA of the architecture as to minimize the *distance* between closely connected sub-circuits. Routing deals with choosing the shortest minimum path for implementing each connection between FPGAs, as to minimize the total *wire length*. Due to fixed topologies and resource limits, all these steps must carefully take into account the nature of the architecture in which the application is being mapped.

The fact that these three tasks are all required does not mean that they have to be performed in three consequent phases. It is not infrequent in literature to find approaches where partitioning and placement are carried out in a single step [14, 15]. Other methodologies, on the other hand, merge placement and routing in a unique phase [16].

We believe that partitioning is the most challenging and crucial task in MFS design, because of its high complexity and the fact that good partitioning solutions

make subsequent steps more effective and easier to be carried out. As a matter of fact, a partitioning with the lowest possible amount of interconnections between sub-circuits makes the constraints which placement and routing have to deal with easier to be met. MFS is only one of the several scenarios in which partitioning is encountered in VLSI CAD: each design flow, even for single-chip systems, include a partitioning step. Unfortunately, partitioning is a NP-complete problem; this means that finding an optimal solution requires exponential time. With the current circuit sizes, this is absolutely unfeasible. Nevertheless, a lot of efforts have been spent in order to develop partitioning heuristics which produce very good results in acceptable run time. Due to its importance in the design flow and in order to provide a solid background, in the next Subsection it is provided a brief review of what we may call *general* VLSI partitioning algorithms, referring to the fact that they are not specifically designed for multi-FPGA applications.

2.1.1 General VLSI circuit partitioning

This section is divided on the basis of the nature of partitioning approaches. First, in Subsection 2.1.1.1 the iterative improvement methods, such as the Kernighan-Lin and the Fiduccia-Mattheyses heuristics are discussed. Subsection 2.1.1.2 describes two randomized iterative methods, namely Simulated Annealing and Genetic Algorithm. Subsection 2.1.1.3 provides an explanation of multilevel partitioning approaches, focusing on the *Metis* partitioning framework. Subsection 2.1.1.4 describes the clustering methodology for partitioning.

2.1.1.1 Iterative improvements methods

The distinguishing feature of the algorithms belonging to this class is that they start from an initial partitioning, and successively improve it through several identical steps. These steps are performed using a metric value which evaluates the quality of a “move” before it is executed. These approaches were introduced by

Kernighan and Lin in 1970 [17]. Fiduccia and Mattheyses gave a major contribution by expanding Kernighan and Lin's work in order to accept as input hypergraph descriptions [18]. Although these algorithms cope with the partitioning of a graph - or hypergraph - in two parts (*bipartitioning*), they can be applied recursively to produce k-way partitionings. Several research works tried to further improve these early works, sometimes providing noticeable enhancements.

The **Kernighan-Lin** (KL) algorithm [17] computes balanced bipartitionings of graphs, aiming at minimizing the cutsize. The algorithm is invoked after the creation of an initial balanced partitioning. It has a two nested loop structure, where the inner loop selects a pair of vertices belonging to different partitions such that its swap gives the largest gain *or the smallest increase* in cutsize. These vertices are then swapped, marked as *locked* (i.e. they are not allowed to move again during that iteration), and the gain for each pair of vertices is updated. This continue until no unlocked pair of vertices exists. After that, the index of the greatest partial sum above the performed swaps (i.e. the first k swaps that give the maximum gain) is computed in the outer loop. If this maximum gain is greater than zero, the correspondent swaps are actually executed, all the nodes return to be unlocked, and a new iteration of the outer loop is performed. If the maximum gain is less or equal than zero, the algorithm ends. In the inner loop the algorithm accepts also moves that temporarily worsen the cutsize, in order to provide the algorithm with the crucial capability of climbing out of local minima.

Under the reasonable assumption that the number of steps of any KL execution is constant (2 to 5 times), the complexity of the algorithm is $O(n^3)$ where n is the number of vertices in the graph. Using suitable data structures and a proper vertex-pair scanning methods, the complexity can be lowered to $O(n^2 \log n)$. Several works propose implementation alternatives that improve the algorithm execution time. In particular, Dutt [19] proposes a method which has a worse case execution time of $O(e \log n)$ where e is the number of edges.

The **Fiduccia-Mattheyses** (FM) hypergraph bipartitioning [18] was proposed

in 1982 and constitutes one of the most influent seminal works for subsequent research. The FM algorithm is similar to KL, and deals with hypergraphs instead of graphs. Each vertex v is associated with a gain, which is the reduction in cutsize obtained by moving v to the other partition. The vertex with the largest gain is selected and, if it satisfy a given *balance constraint*, it is moved to the other partition and tagged as *locked*, until no *unlocked* vertex remains. After each move, the cutsize gains associated with each vertex needs to be updated. Then the gain highest partial sum and its index are computed as in KL, and the vertices are actually moved until that index, while the remaining vertices are not moved. This process is repeated until the gain highest partial sum is negative. The surprising result is that the complexity of the FM algorithm is linear in the size of the network, expressed in term of pin count P, which is the sum of the nets each vertex is connected to. This result is obtained using a *bucket-list* structure for storing the gains of the vertices and fast gain computation expressions.

Several modifications to the FM algorithms have been proposed to cope with different partitioning problems, as k-way hypergraph partitioning [20]. Other approaches aim at performance improvements, usually relying on more effective gain calculation techniques: an example is the work by Dutt and Deng [21], who noticed that one of the drawbacks of the FM algorithm is to compute the gain using local information only.

2.1.1.2 Iterative methods

Iterative methods include those algorithms which are based on random moves and an evaluation function which captures the quality of a given solution. Conversely to iterative improvement methods, these algorithms use a metric function which returns a *global* quality measure of the solution after a random move is performed; on the other hand, iterative improvement algorithms evaluate a *local* quality metric and choose the move to be performed on this basis. Usually, iterative algorithms are suitable for any combinatorial optimization problems, and are particularly fit-

ted to cope with VLSI CAD problems, such as partitioning and placement.

A **genetic algorithm** starts by randomly generating a set of initial solutions called *population*. In each *generation* (i.e. iteration of the algorithm), each single *chromosome* (i.e. possible solution) in the population is evaluated through a *fitness function*. In the following *selection* phase two of the above chromosomes are selected from the population: the individuals having higher fitness values are more likely to be selected. After that, different operators act on the selected individuals in order to generate new individuals called *offsprings*. This genetic operators are *crossover* (applied to a pair of individuals) and *mutation* (applied to a single individual). Different sequences of application of such genetic operators cause different trends in new generations: to give an example, a heavy use of mutation causes the offsprings to be *memory-less*, but also increase the ability to climb out local minima.

The algorithms proposed in [22, 23] cope with multi-objective partitioning problems. In particular, the proposed methodology looks for a good trade-off among cutsize, time delay, power consumption and balance. This trade-off is achieved using a fuzzy logic cost function that takes into account all the mentioned objectives, which is used as the fitness function.

Simulated Annealing is an iterative algorithm which derives from statistical mechanics, and its use in circuit design was proposed in 1983 by Kirkpatrick et al. [24]. The authors noticed that a well known randomized Monte Carlo algorithm for simulating physical microscopic processes (called Metropolis algorithm) was suitable to be used for combinatorial optimization problems. The pseudocode of the algorithm for partitioning problems, as described in [25], is provided in Algorithm 1.

An initial candidate solution is randomly generated, and the algorithm starts at high *temperature* T_0. The *gain* function is crucial in determining the performance of the algorithm. Usually it takes into account both the cutsize and the balance of the partitioning, thus resulting in a *two*-objective optimization. The gain function

Algorithm 1 Pseudocode of Simulated Annealing algorithm. From [25].

```
T = T_0
CurrentGain = CalculateGain()
while t_stop > 0 do
    AcceptMove = FALSE
    for i = 1 to M do
        randomly select vertex V to move from one partition to another
        NewGain = CalculateGain()
        if AcceptGainChange(ΔGain, T) then
            CurrentGain = NewGain
            AcceptMove = TRUE
        else
            return V to original partition
        end if
    end for
    if AcceptMove then
        t_stop = t_s
    else
        t_stop = t_s − 1
    end if
    T = T ∗ α
end while
```

proposed in [25] is $Gain = \frac{cutsize}{|A|*|B|}$, where $|A|$ and $|B|$ are the numbers of vertices in partitions A and B, respectively. It is straightforward to extend this function to k-way partitioning problems. M is the number of *move states* per iteration. When a random vertex is selected for moving from its original partition to another, the move is accepted according to the following rule. If a move will result in an unbalanced partition, it is always rejected. If the balance is preserved and the resulting solution is improved, the move is accepted. Otherwise, the move is accepted with probability $e^{-\frac{\Delta Gain}{T}}$. After each iteration, T is scaled by a *cooling factor* α, $0 < \alpha < 1$. The algorithm stops if there have not been accepted moves after t_s iterations.

2.1.1.3 Multilevel methods

The multilevel partitioning technique was successfully introduced in the middle 1990s, and represents the best known partitioning methodology for *large* graphs or hypergraphs up to date. The main idea behind multilevel partitioning is to split the partitioning process into three main phases. First, the original problem instance is reduced by iteratively clustering and collapsing each cluster into a new vertex. This process is repeated until the size of the instance is suitable for applying an efficient and effective partitioning algorithm. Once an *initial solution* is computed on the smallest graph, it is projected to the upper levels and it is *iteratively refined*. One of the first works to introduce the idea of a partitioning algorithm based on clustering and unclustering was [26]. Alpert [27] and Karypis [28, 29] extensively explored new efficient multilevel solutions simultaneously. The approach described in [28, 29], which is called ***Metis***, is presented. This algorithms computes a k-way partitioning of a graph $G = (V,E)$ in $O(|E|)$ time. As mentioned in the previous paragraph, a multilevel partitioning algorithm is composed of three sequential phases, intuitively depicted in Figure 2.2.

During the **coarsening phase**, a sequence of smaller graphs $G_i = (V_i, E_i)$ is constructed from the original graph $G_0 = (V_0, E_0)$ such that$|V_i| < |V_{i-1}|$. In order for a partitioning of a coarser graph to be good with respect to the original graph, the weight of a vertex must be equal to the sum of weight of the vertices of the original graph that were collapsed to form it. Also, the edges of the new vertex are the unions of the vertices that were collapsed. This rules ensure two important properties: (i) the cutsize of a given partitioning in a coarser graph is equal to the cutsize in the finer graph and (ii) a balanced partitioning of the coarser graph results in a balanced partitioning of the finer graph. This coarsening can be formally defined in terms of *matchings*. A matching is a set of edges, no two of which are incident on the same vertex. The coarser graph is constructed by collapsing couple of matching nodes of the original graph, obtained through different choice policies.

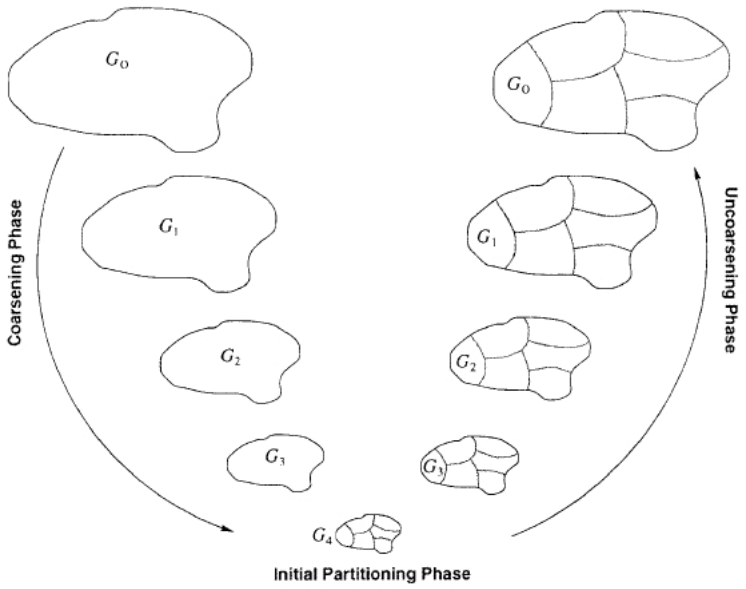


Figure 2.2: Three phases of a multilevel partitioning algorithm. From [28].

The **initial partitioning phase** aims at computing a k-way partitioning of the coarsest graph such that each partition contains roughly $|V_0|/k$ vertex weight of the original graph. The authors describe two ways to produce such initial partitioning. The first is to keep coarsening the graph until it has only k vertices left, but it rises two problems. First, the reduction in graph size becomes very small after some coarsening steps, making it too expensive to continue with the coarsening process. Second, the weight of the obtained vertices are likely to be quite different, making the initial partitioning highly unbalanced. Another way is to recursively use a bisection algorithm, such as FM, which produces good initial partitionings and requires small amount of time.

During the **uncoarsening phase**, the partitioning of the coarsest graph is projected back to the original graph, by going through all the intermediate graphs. it is important to note that, even if the partitioning of a coarser graph is at a local minimum, its projection to the finer graph may not be at a local minimum, since

it has more degree of freedom. Hence, it may be possible to improve the projected partitioning at each level. KL-FM heuristics tends to produce very good results when used as refinement algorithm. The developed refinement approaches are *Greedy Refinement*, which is very efficient but lacks the capability of climbing out of local minima, and *Global Kernighan-Lin Refinement*, which provides hill-climbing features to the greedy approach.

The Metis methodology has been extended by the same authors to deal with hypergraphs, and has been called ***hMetis*** [30, 31]. Despite some effective optimizations have been introduced both in the coarsening and in the refinement phases, it is beyond the aim of this work to describe extensively this hypergraph partitioning approach, since it is methodologically very similar to the already described Metis algorithm.

2.1.1.4 Clustering

Generally speaking, partitioning and clustering are synonyms, and practically it is impossible to rigorously differentiate between clustering and multi-way partitioning [32]. Nevertheless, it is common use to refer to clustering in VLSI by intending a bottom-up method, which starts from single atomic components and iteratively groups them on the basis of some given metrics, as opposed to top-down recursive bipartitioning algorithm or "flat" iterative methods. Clustering is sometimes used as a preprocessing step to reduce the problem size for the subsequent application of others partitioning algorithms: the *coarsening* phase of a multilevel algorithm, seen in section 2.1.1.3, is actually a clustering which eases the subsequent partitioning. Nevertheless, clustering can be used as a partitioning algorithm by itself.

In its simplest form, a clustering algorithm starts by considering all the nodes of the graph which need to be partitioned, and selects a group of them (usually a pair of nodes) which maximizes a given metric function. This group of nodes is then collapsed in a single node, and the algorithm restarts considering the obtained

graph. There are several policies for stopping the algorithm: one example is to fix a maximum dimension of the clusters and stop when it is impossible to form a new cluster which fulfill such size requirement. Otherwise, one can choose to keep on clustering the graph until only one node remains, and then cut the "clustering hierarchy" at any arbitrary level, thus obtaining clusterings at different granularities.

Previous works on clustering can be grouped into approaches that use *local* or *global* information. Intuitively, using metrics which capture global circuit features is better, but it requires much more computation time than using local metrics.

Many different metrics can therefore be adopted in guiding a clustering algorithm: in the following, we briefly report three *closeness* metrics which have been classically adopted for circuit clustering, and are considered in [26]. *Cluster Density* is defined as follows: given a cluster of c nodes, it is equal to $\frac{E}{M_c}$, where $M_c = \binom{c}{2}$ and E is the total weight of edges entirely contained in the cluster. Clusters having a higher density are supposed to be of higher quality. The $k-l$ *connectedness* metric (no relations with the KL algorithm) claims that two nodes shall be clustered if there exist k edge-disjoint paths connecting them such that each path has length at most l. It is not trivial to determine the values of k and l. According to the *degree/separation* metric, two nodes with a higher d/s ratio are of higher quality, where d is the average number of nets incident to each component in the cluster, and s is the average length of a shortest path between two nodes in the cluster. Even if this is a robust measurement, it suffers of high computational requirements. Several other metrics have been proposed; some of them will be investigated later in this work.

2.1.2 MFS global layout

As pointed out at the beginning of this chapter, the global layout phase is composed by partitioning, placement and routing. It is not usual to find in literature

approaches that address all these problem thus providing a complete design flow. Usually only one or two of these tasks are addressed at a time. In this section, a review of some of these approaches is provided, organized on the basis of their completeness and methodology.

2.1.2.1 Complete MFS design flows

In the following, the descriptions of two existent complete multi-FPGA design workflows are provided. Subsection 2.1.2.1 presents the work made by Scott Hauck ([5, 33]), while Subsection 2.1.2.1 explains the approach due to Muhammad Khalid [6].

Hauck's approach The aforementioned work by Hauck [5, 33] provides algorithms which deal with all the three basic tasks. Partitioning is carried out through recursive bisections. Each bisection is computed using a methodology that resembles the multilevel approach described in section 2.1.1.3. First, clustering based on a connectivity threshold metric is executed, until no new cluster is formed. Then an initial random bipartitioning is produced. At this point, an *unclustering* phase iteratively performs a slightly modified version of the FM algorithm and uncluster to the next finer grained level, until no cluster remains. The unclustering phase is run ten times, and the best result is returned. These bisections are executed recursively in order to obtain a partitioning into an arbitrary number of sets. This recursive bisection process actually embeds placement: the topology of the multi-FPGA architecture is considered. An algorithm determines the *partition orderings*, that are the portions of the architecture to which the recursively computed bipartitions are assigned. This can be thought as a partitioning problem *per se*: it is better to assign the first two obtained partitions to the least connected portions of the topology, the algorithm must indeed find what are the main bottlenecks in the topology. The proposed algorithm is complex, and it is beyond our scope to provide an accurate description. At the end of this process, each of the

obtained sub-circuits is assigned to a specific FPGA. What is not underlined in the analyzed work is that this process implies a fixed routing solution: the inter-FPGA wires will pass through the architectural cuts extracted by the partition ordering algorithm, in a hierarchical fashion. This work also deals with pin assignment and intra-FPGA routing - which are strictly related - and carries out this task using a force-directed algorithm.

Khalid's approach As previously said, Khalid [6] developed a multi-FPGA tool chain to provide support for comparing different architectural topologies. The proposed algorithms are simpler than Hauck's ones. The partitioning approach is basically a simple implementation of recursive bisection: the circuit is iteratively split using FM until all partitions fits into a single FPGA, considering both area and pin count. The recursion is handled by using a stack structure: non-fitting sub-circuits are placed on the top in order to be further subdivided. Placement is managed through several algorithms, depending on the architecture topology. For mesh topologies, a force directed approach is proposed: the sub-circuits are seen as masses and the connections as springs, whose elasticity is proportional to the connection width. The optimum ideal placement would be an equilibrium configuration of the masses with respect to all the elastic forces. Simplifying the thing, each module is tried to be placed in its ideal position: if it overlaps with another module, the algorithm places it in the nearest vacant place. For partial crossbar topologies, placement is trivial since all pairs of FPGAs have the same distance. Placement in the Hard-Wired Cluster Architecture (described in Section 1.2.1), is performed by exploiting the hierarchy created by the recursive bipartitioning: sibling sub-circuits are tried to be placed in the same clusters.

To cope with the routing problem, Khalid first proposes a topology-independent global router, which transforms a given architecture topology into a graph and execute an algorithm based on maze routing [34]. Since this generic approach does not show encouraging result, the author proposes ad-hoc routing algorithms

for specific topologies. For routing mesh architectures, an algorithm is proposed which, for each net, enumerates all the possible shortest paths and chooses one of them on the basis of the generated congestion. Clearly this approach is hardly scalable to large size meshes. The algorithm for the partial crossbar routes the nets in decreasing order of fan-out. If the routing attempt fails, the routing restart and the net which caused the failure is mapped first. The cost function used to select the FPID used to route a net attempts at guaranteeing a balanced usage of the FPIDs. The approach for hybrid architectures is similar: first, all the possible two-terminal nets are routed on the hardwired connections. Then, the remaining nets in decreasing order of fan-out are mapped on the FPIDs.

2.1.2.2 Partial MFS design flows

Iterative approaches Iterative methods (section 2.1.1.2) have been widely applied to multi-FPGA global layout synthesis. Usually partitioning and placement are treated with such techniques, but some routing solutions have been developed as well. In [14] the authors propose a genetic algorithm for partitioning and placement targeted to 4-mesh topologies. The fitness function is implemented through a fuzzy technique which takes into account FPGA logic capacity and pin utilization. The algorithm is designed in such a way that the circuit being mapped is fairly distributed on *each* FPGA composing the mesh. This sounds like an over-demanding requirement, since it is likely that the imposition of using all the available FPGAs forces the algorithm to introduce inter-FPGA connections which could be avoided by using less chips. In [35] the authors show a faster parallel implementation for the same algorithm.

Other approaches apply simulated annealing to MFS global layout. In [15] the authors describe an approach which is claimed to deal with N-way intra-chip and inter-chip partitioning and placement. It is evident that the method is applicable only to mesh topologies. Each chip is divided into a number of bins disposed as a matrix: the nodes of the graph to be partitioned are assigned to these bins instead

that entire chips, such that the method performs also a kind of local placement. This is an arguable point, since one can wonder whether it is worth to perform local placement simultaneously with global layout. However, the approach appears smart in the application of the move acceptance function in the simulated annealing algorithm, adopting both single element moves and pairwise exchanges. The described tool also performs pin assignment, even if the correspondent routing algorithm is only hinted. Furthermore, the input circuit undergoes a preprocessing clustering stage in order to reduce the size of the problem and make the subsequent simulated annealing more effective.

Another MFS mapping tool for mesh topologies based on simulated annealing is proposed in [16]. The authors call their methodology *Placement&Routing-based Partitioning* (PRP). They propose a solution to two similar problems: Fixed Mesh and Adapted Mesh. The first is the usual multi-FPGA global layout problem, in which a given circuit must be mapped on a given architecture. The latter problem is to construct the minimum size multi-FPGA model that can house a given circuit. The process is carried out using a structure similar to the approach described in [15] and presented above: the multi-FPGA system is modeled as a large single FPGA where the borders among neighboring chips are described as a superimposed template. A noticeable feature used in the simulated annealing cost function is that the cutsize is computed using an approximation of the Rectilinear Steiner Minimum Tree routing algorithm. An evident weakness that these last two presented approaches share is that they lack any generality in the MFS topology: they indeed address solely mesh architectures.

Hierarchical approaches During recent years, some innovative approaches have tried to go beyond classical partitioning schemes, which deal with large, flat netlists. As a matter of fact, several researchers noticed that the the conversion of a design into a flattened hypergraph involves the loss of much information [32, 36, 37]. To deal with the ever increasing complexity of logic circuits, VLSI

developers adopts nowadays hierarchical design styles. In other words, we can refer to design hierarchy as a way to handle design complexity [32]. Despite design hierarchy provides a natural way to decrease the complexity of CAD design problems, most current approaches totally ignore such precious information by using large flat hypergraphs.

The work reported in [32] was one of the first approaches to suggest the exploitation of design hierarchy in MFS synthesis. The mentioned paper only addresses the partitioning problem. The objectives are to minimize the number of used devices and the interconnections among them. The approach is quite elaborated: the hierarchy is visited and the elements which fit in a single FPGA are added to different partitions; unfeasible elements are therefore "split" and non-leaf sub-modules belonging to the same parent are tried to be placed in the same partition. The remaining leaves, usually of small dimension, are inserted in already existent partitions or are partitioned using a FM k-way algorithm. Moreover, non-leaf modules which do not fit into a single device because of slightly margins - either of dimension or I/O pins - are not split: instead, the smallest element whose elimination causes the module to fit is extracted and treated separately. Eventually, a final merging step combines small partitions to larger ones.

Another approach to partitioning, presented in [38, 39], introduces an important feature: it includes an early HDL synthesis step. As a matter of fact, the described tool is able to take as input a Verilog description of the design and turn it into a hierarchical tree. One point of weakness of this synthesis approach is that it extract a hierarchical tree made of four levels, namely *top*, *modules*, *processes* and *functions*. It does not take into account hierarchical information that goes beyond this fixed layered tree. When the tree is created, it is treated as a *hierarchical connected graph*, and a top-down set-covering procedure is applied to generate the partitions. The algorithm method starts the covering from the top level nodes (modules). If no more feasible covers can be found at that level and the hierarchical hypergraph is not entirely covered, the set-covering process continues

Table 2.1: Summary of the main MFS design approaches.

<table>
<tr><th></th><th>HAUCK</th><th>KHALID</th><th>HIDALGO ET AL.</th><th>ROY ET AL.</th><th>HERMIDA ET AL.</th><th>BEHRENS ET AL.</th><th>FANG ET AL.</th></tr>
<tr><td>TOPOLOGIES</td><td>Mesh</td><td>Mesh, Crossbar, Hybrid (HTP, HCGP, HWCP)</td><td>Mesh</td><td>Mesh</td><td>Mesh</td><td>-</td><td>-</td></tr>
<tr><td>PARTITIONING</td><td rowspan="3">Clustering + Rec. Bisection + Unclustering</td><td>Recursive Bisection</td><td rowspan="2">Genetic Algorithm using Fuzzy Logic</td><td rowspan="2">Simulated Annealing</td><td rowspan="3">Simulated Annealing (using Approx. Steiner Tree for wire length estimation)</td><td>Hierarchical heuristic</td><td>Synthesis from Verilog + Hierarchical hypergraph seet covering</td></tr>
<tr><td>PLACEMENT</td><td>Force directed (Mesh), Partitioning Hierarchy Exlpoitation (HWCP)</td><td>-</td><td>-</td></tr>
<tr><td>ROUTING</td><td>Specific on the topology</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
</table>

on lower level nodes.

2.1.3 Comparison of existing approaches

Table 2.1 summarizes approaches to MFS design analyzed in the previous sections. The slash ('-') contained in some cells means that the corresponding aspect of the workflow is not addressed by the considered approach. It is clear that usually researchers focus only on some aspects of the design flow, providing algorithms which address only some parts of the whole problem. The creation of a complete workflow is usually hard, since it has to be bound to the particular input and output specifications. Nevertheless, a methodological design flow is proposed in this thesis, which, though fitted on specific input-output requirements, contains core algorithms which are not necessarily bound to any input-output format.

So far in this section, the shape of a general MFS design flow has been traced, and the state-of-the-art in this field has been described, together with some background and seminal methods in partitioning. In the next section, a review of the few existing approaches to dynamic reconfigurable MFS's, that constitute the leading topic of this work, is provided.

2.2 Dynamic reconfigurable MFS's

In [40], the authors describe a partitioning and synthesis system for dynamically reconfigurable MFS's. The presented workflow takes as input a behavioral VHDL specification of the application to be mapped onto the multi-FPGA architecture. A high-level synthesis tool, called DSS (Distributed Synthesis System), transforms the input specification into a directed task graph. This constitutes the input for the MFS design workflow, which is based on two phases: temporal partitioning and spacial partitioning. Intuitively, temporal partitioning carries out the division of the input specification into *time segments*. Each of these segments is then spatially partitioned over the FPGAs constituting the target architecture. The temporal partitioner has an abstract view of the architecture resources, such as the overall resource availability and memory size. The time partitioning process, which inherently performs a list-based scheduling of the tasks, is based on a *Binary Non-Linear Programming* model: this model takes into account several constraints, such as the data dependencies between tasks, the overall available FPGA resources, and the amount of memory which can be used during reconfiguration. Practically, the time partitioner isolate portions of the scheduled task graph which are estimated to fit onto the target architecture at a given instant. Then, a spatial partitioning is performed for each temporal segment. This task is carried out through a genetic algorithm, which takes into account the available resources of each FPGA. From what has been said, it is evident that the application is split into discrete time segments: once each segment is executed, a *total* reconfiguration of the system takes place, with temporary results being stored in local memories.

For a better understanding, a graphical representation of this approach is provided for a simple task graph in Figure 2.3. It is clear that the reconfiguration of the system is *not* dynamic in the sense we intended in Chapter 1. As a matter of fact, each time a reconfiguration is performed, intermediate results are stored in memory, the system halts, and the execution restarts once the reconfiguration has

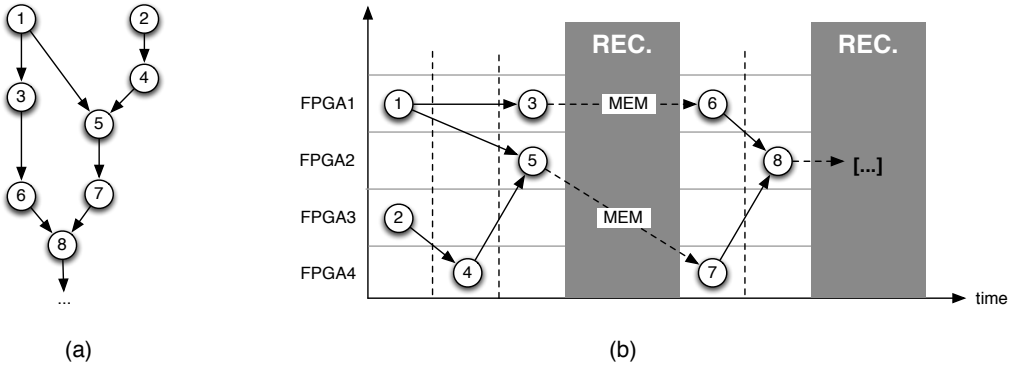


Figure 2.3: Example solution for a reconfigurable MFS with the approach proposed in [40]

been completed. Therefore, this approach is dynamic in the sense that an application is executed automatically over time, even if it does not fit entirely on the target architecture, but it is static if it is considered that the execution is stopped during reconfiguration. Hence, this approach does not try to mask the time spent by the reconfiguration processes, which constitutes the main source of time consumption.

The work presented in [8] deals with the creation of a hardware/software architecture which can host dynamically reconfigurable systems. Although it does not address problems in the MFS design flow, it is an example of how dynamic reconfiguration can operationally be carried out in multi-FPGA architectures. The approach is targeted to a specific hardware architecture, called *Raptor2000* (www.raptor2000.de). Such architecture consists of a motherboard and up to six application specific extension modules (ASMs). These modules can contain either programmable hardware chips or other devices, such as Ethernet or USB ports. Basically, the motherboard provides the communication infrastructure between ASMs and connects the board to a host computer via the PCI bus. The reconfiguration operations are managed through a light-weight Linux operating system, enhanced with support to dynamic reconfiguration of the devices,

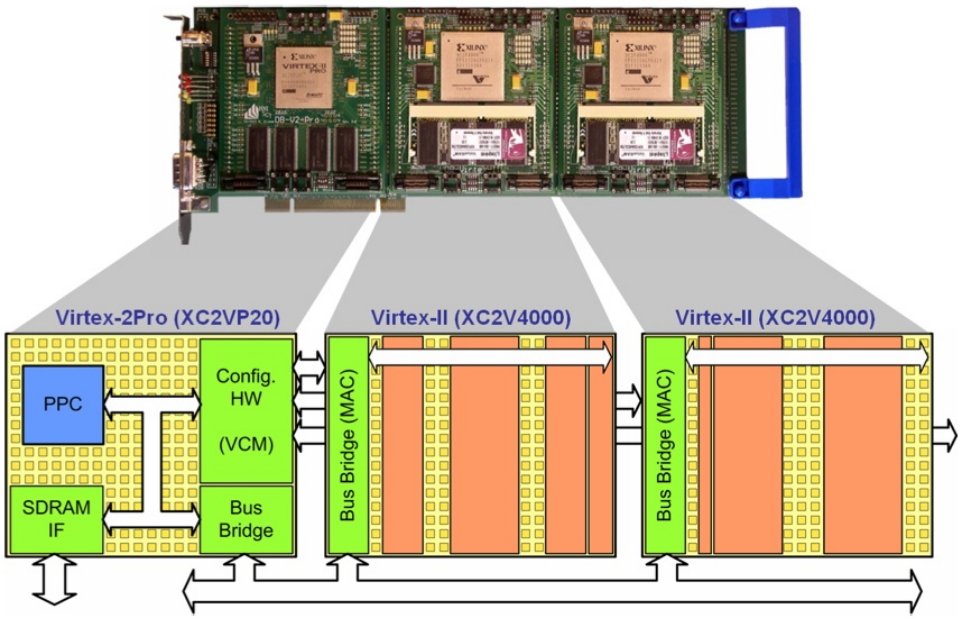


Figure 2.4: Multi-FPGA environment on Raptor2000 architecture. From [8].

that can be used to control the reconfiguration by means of simple function calls. The OS runs on the PowerPC processor embedded in a Xilinx Virtex-2Pro FPGA, which is plugged on one of the ASMs. The specific architecture tested in [8] is depicted in Figure 2.4. In this figure, the leftmost ASM contains the *static* part of the architecture, composed of the embedded PowerPC processor, the configuration controller, the bus bridge, and a memory port. The other ASMs are provided with a communication interface to the bus. The remaining part is the reconfigurable area, in which the reconfigurable modules (IP-Cores) can be placed. When an IP-Core has to be plugged into the system, the corresponding driver is mounted in the OS by means of a function call, and the IP-Core bitstream is loaded from memory and downloaded onto one of the FPGA boards. This scheme also contains a dynamic relocator tool, which, by manipulating precise portions of the bitstream, is able to place the IP-Core in the desired position.

Chapter 3

Methodology

This chapter provides a methodological description of the design workflow for multi-FPGA systems that is the topic of this thesis. After providing a general overview on the proposed methodology (Section 3.1), the chapter proceeds by describing the phases which constitute the design flow. Section 3.2 describes the design extraction phase and the rationale of the choices that have been made. Section 3.3 provides a methodological view of the global layout phase, which is in charge of trying to find a feasible static solution. Due to the fact that it represents the main innovation source of this thesis work, the description of the third part of the workflow, which is the methodology to be applied in case a design does not statically fit in the architecture, and deals with reuse and dynamic reconfigurability, is demanded to Chapter 4.

3.1 Overview

The aim of this work is the development of a workflow for the design of dynamically reconfigurable multi-FPGA systems. In order to achieve such a result, it is necessary to define a methodology which copes with all the steps of the design flow. First, a distinction has to be stated, between the case in which the

input application fits on a given multi-FPGA architecture, and the case in which novel solutions have to be adopted, in order to make the implementation of larger over-requiring applications feasible on multi-FPGA architectures with bounded resources. Such novel solutions imply the adoption of dynamic reconfiguration techniques. A remarkable point is that dynamic reconfigurability is in this case viewed as a feature which is not strictly desirable, because of the degrading of performance it causes. Nevertheless, it becomes necessary in the case the application does not *statically* fit on a given architecture. Therefore, a novel methodology is needed to minimize the impact of this feature that the design flow is forced to adopt.

From what has been said, the workflow should be composed of two main phases. The first one is the attempt at finding a solution for statically implementing the input application on a given multi-FPGA architecture. As it should be clear from the definitions given in Section 1.3, this attempt may either succeed or fail. In the first case, the flow ends. In the second case, the flow undertakes a branch which has the goal of having the application implemented on the architecture, even if it does not entirely fit into it. To achieve this result components reuse over time and dynamic reconfigurability techniques are considered. The main phases which compose the workflow are shown in Figure 3.1.

The workflow takes as input an application which has to be implemented on the multi-FPGA architecture, specified as a VHDL description. This description has to be parsed in order to build the intermediate representation to work on in the following steps. Therefore, a preliminary design extraction phase has to be carried out, which operates on the VHDL code provided as input. Along with the description of the application, the proposed design chain also needs the specification of the architecture the application has to be plugged on, provided by a separate file. Then the two-phase process described above is carried out. The outputs of the workflow are the VHDL files which compose the multi-FPGA system and information such as global routing and, if necessary, reconfiguration specifications.

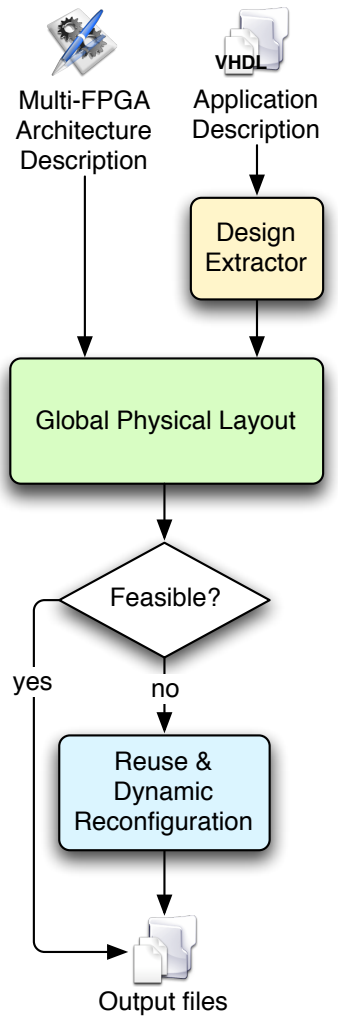


Figure 3.1: Outline of the proposed multi-FPGA design workflow.

3.2 Design extraction

One of the goals of this work is to provide a workflow which handles high-level specifications of hardware circuits. This choice allows the flow to work with structures that can be manipulated and implemented in several ways, and are not already subject to the design constraints typical of low-level netlists. This concept will become clearer later in this chapter, when we will talk about reuse and dynamic reconfigurability, in Section 4. VHDL has been chosen as the input hardware description language for the workflow. Therefore, a design extraction phase is needed, which takes as input the VHDL code of the application to be implemented and produces as output the structures the subsequent steps of the design chain work on. The output produced by this first phase is an intermediate representation which provides information about the design *structure* and *hierarchy*. Since this choice needs to be carefully explained and justified, in the next section we provide an explanation of the concept of "hierarchy" in circuit design and its main advantages.

3.2.1 Hierarchical circuit design

This section aims at introducing the concept of hierarchical circuit design. First, in Section 3.2.1.1 some formal definition and design scenarios are provided. Then, Section 3.2.1.2 describes some advantages of adopting hierarchy in the design of logic circuits.

3.2.1.1 Definitions and scenarios

In this section some useful definitions are provided, which may ease the understanding of the forthcoming parts of the thesis.

Definition 3.1. *A **design block**, or **module**, is a (portion of a) logic circuit with a given list of inputs and outputs signals. A* port *is an element of such list.*

Definition 3.2. *A **structural representation** of a logic circuit, or simply **structure**, is composed by design blocks interconnected by wires, usually called* nets*. The blocks connected through a net are the net* terminals*. Generally, a net can have more than two terminals.*

This definition is almost identical to the definition of *netlist*, but this latter name is usually used when referring to structures composed of low-level logic components, such as logic gates.

Definition 3.3. *A **hierarchical circuit** is a circuit in which some of the blocks contained in the structural representation can be further split into sub-blocks. These sub-blocks and their interconnections constitute a* sub-structure*. Design blocks which cannot be split are called* leaves*. The least detailed structural representation of the circuit design is composed of a single block, correspondent to the whole circuit. Such structural representation is called the* top *of the design. A sub-block contained in a block is called* child *of the block, while the super-block containing a block is called* parent *of the block. It is straightforward that leaves do not have children and the top does not have a parent.*

Intuitively, we can view the design hierarchy as a tree, where edges are the parent-child relationships. An example of the structure of a hierarchical circuit and the relative hierarchy tree is shown in Figure 3.2(a,b). Every cut of the tree which makes the leaves to be all on the same side provides a complete non-hierarchical structural representation of the design, i.e. the one containing solely the blocks which are immediatly below the cut. An example is shown in Figure 3.3(a,b). The cut composed by all the edges which connect a leave to its parent provides the finest granularity structural representation of the design, as shown in Figure 3.3(c,d).

We can identify two scenarios for hierarchy in VLSI design. The first one is when the circuit is explicitly designed in a hierarchical fashion, using suitable tools and specification languages, such as VHDL. In such case, the hierarchy is

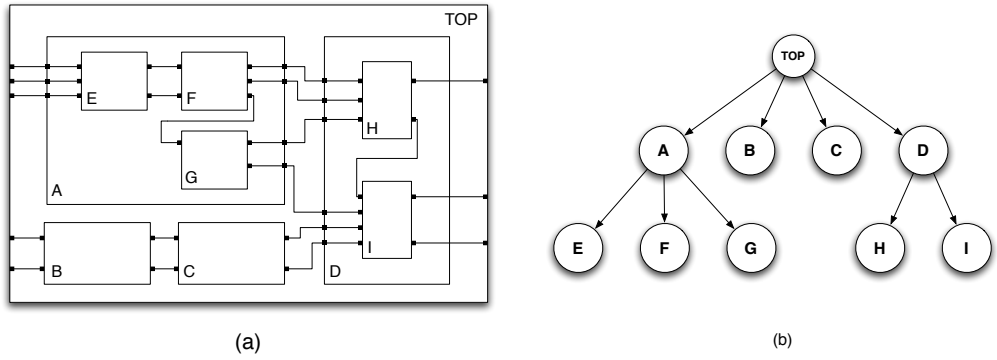


Figure 3.2: Structural representation of a hierarchical circuit (a) and its hierarchy tree (b).

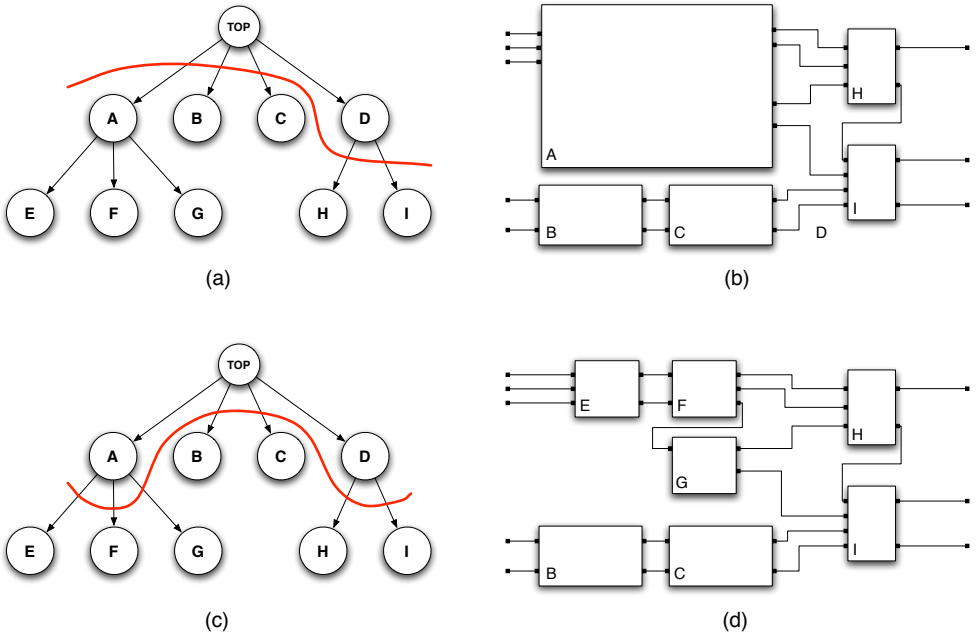


Figure 3.3: Cuts of the hierarchical tree and corresponding structural representations of the circuit.

extracted by "reading" the circuit description. The other scenario is the case in which the circuit has been designed in a standard way, usually as a flattened netlist. In that case, a hierarchical structure may be identified as a step of the design chain, usually using a clustering bottom-up approach. As a matter of fact, we explained in Subsection 2.1.1.4 how clustering can be recursively applied until obtaining a single node. That node is the *top* of the design hierarchy that is being extracted. It is evident that different clustering metrics imply different hierarchies to be built. In the remainder of this work, we assume that the circuits received as input have a hierarchical nature.

3.2.1.2 Advantages of hierarchical design

Hierarchy offers to designers many advantages in the creation of logic circuits. Hierarchical design is usually presented as a solution to handle the rapidly growing complexity of circuits [32, 38, 13]. The reasons lays in several considerations. First, hierarchy offers to the designer a way to create pieces of the application independently. To create a parallelism, this is somehow similar to object oriented design of software applications: once the interface and the function that a component has to carry out is defined, then the actual internal implementation of the class which represents that object becomes a hidden detail. It is important to notice that this advantage is crucial in both top-down and bottom-up approaches. In a top-down design, the circuit can be divided recursively into smaller blocks, which are possibly implemented independently. In a bottom up approach, smaller components are implemented and then connected together to form bigger blocks, until the overall design is formed. This is strongly related to the fact that hierarchy implies *abstraction*[1] [41]: designers can handle components as they were high level functionalities and plug them into the circuit.

The other crucial advantage due to hierarchy and, in general, to structural

[1] In [41], abstraction is defined as "a method to replace an object by a simplified one that only defines the interactions of the object with its environment".

design, is the reuse of components. As a matter of fact, basic components can be implemented once, and then reused every time they are needed. This advantage is even more effective in case high level description languages (such as VHDL) are adopted, since the specification of a component is not bounded to a particular physical implementation, and the module can be reused in several contexts. The reuse of modules, besides having the obvious advantage that different parts of the circuit are designed only once, thus saving time, has also other advantages. To give an example, in design testing, if the the functioning of a block is proved to be correct, then all the blocks of the same kind that have been inserted in the circuit are correct. The reuse of hardware blocks can be further extended to the case in which IP-Cores (Intellectual Property Cores) are available. In this context, the term IP-Core refers to hardware component that are ready to use, meaning that they can be plugged directly into the description of a circuit. In that case, already implemented and tested blocks can be picked from an existing library and plugged into the application being created: it is straightforward that such a methodology allows enormous time savings and improves the reliability of the circuit being created.

In this subsection some of the advantages that the use of hierarchy involves in the creation of a hardware design have been described. In Section 3.3 and 4, the importance that hierarchy information have in the subsequent multi-FPGA phases will further justify the choice of extracting hierarchical information from VHDL.

3.3 Global Layout

As previously said, the *global layout* phase deals with the search of a feasible implementation of the input application on a given multi-FPGA architecture, while attempting to optimize some objectives. The adjective “global” refers to the fact that this phase is not concerned about the local internal layout of each FPGA, which is carried out later. In our workflow, we let already existent third-party

CAD tools (such as Xilinx ISE [3]) to be in charge of internal layout.

The solution looked for during the global layout phase is *statical*, which means that the algorithm tries to fit the entire application into the multi-FPGA architecture. More precisely, what is pursued is a function which assigns a host FPGA to each module of the input application and finds a route for each interconnection between two modules. This relation is required to minimize a cost function, which is commonly the length of the interconnections between blocks belonging to different FPGAs. The rationale behind this usual choice is that off-chip wires are undesirable because of several reasons [42]. They cause a performance degradation, since signals flows at a slower speed on external wires. They reduce the reliability of the system, as printed wiring is more likely a source of troubles than internal wires. Finally, the presence of off-chip wires increases the need of I/O pins, which is known to be one of the crucial constraint of multi-chip design. In the following subsection, it will be explained how global layout is carried out in the proposed methodology.

3.3.1 Global layout tasks

As described in Section 2.1, the global layout phase is composed of three tasks - partitioning, placement, and routing - which need to be aware of the features of the target architecture. Therefore, besides taking as input the structures produced by the design extraction stage, this phase receives a file which contains information about the used multi-FPGA architecture. This file describes the topology of the architecture and characterizes the used FPGAs in terms of available area and I/O pins. Moreover, if a crossbar topology is chosen, the features of the routing chips have to be provided as well.

There are many possible ways to find a solution to the global layout problem. First of all, it has to be decided how to handle partitioning, placement and routing. As a matter of fact, it has already been pointed out and shown (Section 3.3) that

these three tasks, though all necessary, are not required to be carried out separately. In this work, two ways for finding a solution to the global layout problem are considered. First, a methodology is taken into account in which partitioning and placement are performed as a unique step, followed by routing. Second, an approach in which the three steps are executed separately is considered.

The rationale behind the first of these two approaches it is that partitioning and placement goals are strongly related. Placement has indeed the goal of minimizing the *estimated* wire length, and this naturally implies the sub-goal of minimizing the cutsize between partitions. Nevertheless, the complexity of the resulting problem is high, since the solution space is wider than the ones of the two problems taken separately, due to the increased number of dimensions of the solution space. Since iterative methods are known to operate better than others when the solution space has high dimensionality, a simulated annealing approach has been chosen to cope with such combined problem, . Random moves and solutions global evaluation functions, which are typical of such algorithms, allow the partitioning-placement process to avoid considering all the variables involved in the problem, such as finding the better move at a given point on the basis on the estimated maximum local gain. Another advantage of using an iterative technique lays in the fact that it can be used for every multi-FPGA topology - once the notion of "distance" between two FPGAs is provided - while other methods only address the problem for particular topologies (see for example the force-directed placement for mesh architectures proposed in [5]).

The second approach, that is to separately execute the three steps in sequence, aims at exploiting the effectiveness of a partitioning algorithm specifically designed for the task of reducing the cutsize. The idea is that a combined partitioning and placement carried out using an iterative algorithm is feared to be less effective for the precise goal of minimizing the cutsize than a specifically designed heuristic. In this case, after partitioning is executed, the generated partitions are placed on different FPGAs, during what we call a *1-to-1 placement* process. The par-

titioning approach proposed in this scenario is a bottom-up clustering algorithm based on different closeness metrics, executed taking into account the hierarchical nature of the design. The 1-to-1 placement is carried out using iterative techniques, because of the several advantages explained above. The implementation details of all these approaches are described in Chapter 5, while their evaluation through experimental results is provided in Chapter 6.

In order to better understand the nature of the two partitioning and placement scenarios, namely the one in which they are performed separately and the case in which they are carried out as a single step, we introduce a very simple mathematical formulation. This formulation covers only some aspects of the problems, but is anyhow useful to clarify the ideas. We define the following sets:

- $A = \{a_1, a_2, \ldots, a_n\}$ is the set of the blocks of the application to be implemented.
- $P = \{p_1, p_2, \ldots, p_m\}$ is the set of partitions.
- $Q = \{q_1, q_2, \ldots, q_m\}$ is the set of the FPGAs that forms the target multi-FPGA architecture. Notice that the cardinalities of P and C are equal.

At this point, a *partitioning* is defined as a function

$$f_{PART} : A \longrightarrow P$$

Moreover, a *(1-to-1) placement* is defined as a function

$$f_{PLAC} : P \longrightarrow Q$$

It follows than the partitioning and placement resulting from the *sequential* execution of the two processes is the composition of the two functions, hence:

$$f_{PART} \circ f_{PLAC} : A \longrightarrow Q$$

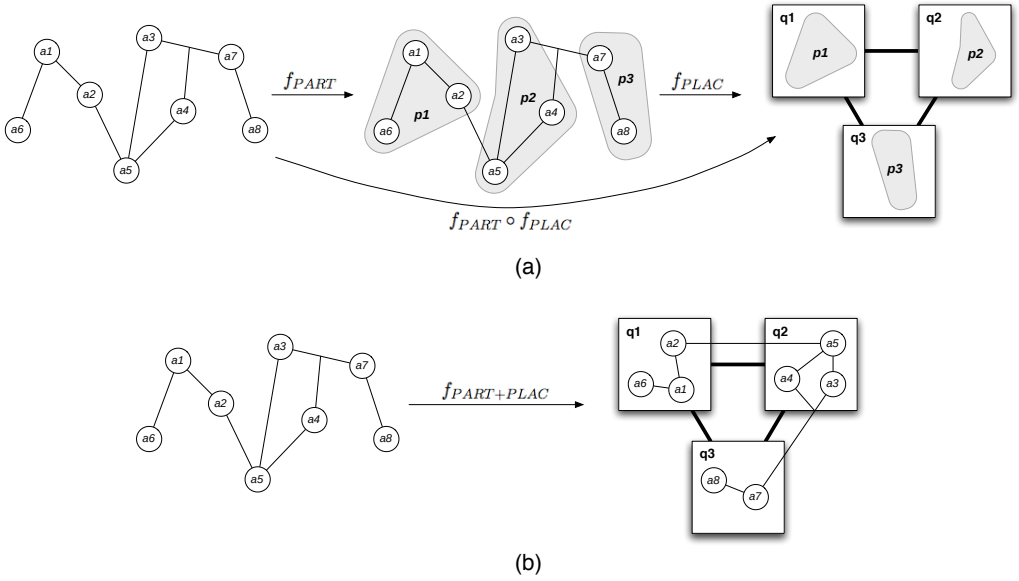


Figure 3.4: Example of separated (a) and integrated (b) partitioning and placement.

A partitioning and placement process performed as a single step is defined as another function

$$f_{PART+PLAC} : A \longrightarrow Q$$

which is in general different from $f_{PART} \circ f_{PLAC}$.

Figure 3.4 shows graphically through the use of a simple example the two different ways to cope with partitioning and placement described by the functions above. The algorithms which in our case implements these functions are described in Chapter 5.

The following subsection explains the reasons why information relative to the design hierarchy are useful in placement and partitioning algorithms.

3.3.1.1 Combining hierarchy and global layout

In Section 3.2.1.2 the advantages of using hierarchy *for the description* of the hardware application to be implemented have been described. The present section

explains what are the advantages of extracting and using hierarchy information in the global layout phase. In other words, it is explained how algorithms can exploit hierarchy to produce better results, in particular for partitioning and placement. The exploitation of hierarchy provides two main advantages, described in the following.

The *first* one is related to the same reasons that bring the designer to consider a particular hierarchy in the design of a circuit. What is argued here is that if in creating a circuit smaller modules are recursively aggregated to create larger ones, thus generating a hierarchy, there must be a criterium that drives the designer in such choices. Consider the following situation, in which there is a filter which performs some data manipulations and a controller that handles such filter by sending and receiving data. It is likely that this two components, which may be in turn composed by several sub-modules, are aggregated by the designer in a single module. The rationale behind this choice is that the two elements are strongly interconnected. The one just described is a case of bottom-up design, but the situation does substantially not change if the circuit is designed in a top-down style. In such case the designer tends indeed to split the circuit by creating sub-modules which are weakly connected among them. It is not difficult to see that the criterium that drives the designer in this kind of choice is very similar to the one which may be applied by a partitioning algorithm. This means that hierarchy carries important information that can be used in solving the partitioning problem, both when this is addressed singularly or is faced with an integrated partitioning/placement approach. There are several ways in which hierarchical information can be exploited. In a top-down set-covering of the design hierarchy, for instance, the hierarchy itself represents the structure on which the algorithm is applied. In a bottom-up clustering, on the other hand, the fact of belonging to the same parent or to a common ancestor can be considered as a closeness metric which is explicitly used to create clusters.

The *second* advantage of using hierarchy in global layout is related to the con-

cept of *regularity*. Regularity can be defined as the "repeated occurrence of computational patterns" [43] and has been exploited in VLSI design in several ways. To provide an example, the approach in [44] exploits the regularities of the circuit in logic synthesis by applying similar transformations to the gates belonging to the same regularity group. These regularity groups are extracted by an algorithm through the use of "regularity signatures". Although it is not our scope to describe in depth the mechanism of such approach, it anyhow suggests how regularities can be exploited to expand the scope of decisions that an algorithm takes on a local basis. The big advantage is that hierarchy implicitly carries information about the regularities of a design, that therefore do not need to be *extracted* by any particular algorithm, as they are already provided by the hierarchy "for free". As a matter of fact, if the same component is used more than once in a circuit, it is evident that its sub-blocks constitute *regular patterns*, identical one to each other. This information can be effectively used in clustering: when a pair of nodes is chosen for being collapsed on the basis of a given closeness metric, every occurrence of the same pair of nodes in other blocks is detected and the nodes are clustered as well. The methodological advantage of using this feature in a clustering algorithm is that regularities represent a source of *global* information, which is usually hard to be obtained in clustering due to the high computational requirements needed for calculating global closeness metrics, as known from section 2.1.1.4.

3.3.2 High-level granularity and blocks features retrieval

One methodological issue to be addressed before designing the algorithms mentioned in the previous section is to decide the *granularity* they will work with. Intuitively, the granularity of a circuit description can be defined as the abstraction level of the modules it contains. Going from the physical level up to behavioral modules, the granularity gets coarser and less implementation details are specified. In the case of a hierarchical design, it is evident that different levels of the

hierarchy provides the description of the same circuit, but at different levels of granularity, as exemplified in Figure 3.3.

The main advantage of using lower-level structures in VLSI design problems like partitioning and placement is that algorithms work on a more detailed representation, which is supposed to lead to better results, because fine-grained optimization operations can be carried out. One evident drawback is the higher execution time, because the size of the input is larger. On the other hand, higher-level structures lead to faster algorithms, since they represent the same circuit with a smaller number of blocks, hence the execution time is lower. Evidently, using such high-level blocks does not allow low-level optimizations, thus providing sub-optimal results. Nonetheless, it is useful to further analyze this last claim, taking into account the considerations made in Section 3.3.1.1 about the advantages of hierarchy in partitioning. Since designers create hierarchical blocks putting together sub-blocks which are strongly related, in the sense that they have many interconnections, it is a good idea to exploit such information and use high-level structures in partitioning and placement, in order to reach good results while keeping the computational cost of algorithms low. Relying on these considerations, the approaches proposed in this work deal with granularity at the process level of VHDL descriptions, which means that every leaf of the extracted hierarchy is a VHDL process. Another argument that justifies the use of high-level - and thus larger - blocks is that the proposed algorithms deal with *global* partitioning and placement in a multi-FPGA architecture, where the partitions correspond to whole FPGAs. This implies that it is reasonable to deal with relatively large blocks, and to delay the intra-chip layout to subsequent phases of the design flow.

Section 2.1 mentioned the issue of technology mapping in a multi-FPGA design flow. What is commonly stated is that it is better to perform technology mapping *before* the algorithms are executed, so that they essentially work on netlists of CLBs, instead of netlists of logic gates [5, 6]. The main reason is that in this way algorithms are provided with a real estimation of the area and the timing of the

blocks of the circuit that is being partitioned and placed. Since the algorithms proposed in this work use high-level blocks, it is important to have good estimations of the characteristics of such modules. In this case, though, technology mapping is not enough, because high-level blocks also need to be synthesized. To address this problem, the proposed framework uses the synthesis software tools provided by the FPGA vendor, such as XST (Xilinx Synthesis Technology) [45]. The choice of using third party software raises from the fact that designing from scratch a complete and effective logic synthesizer is outside the scope of this thesis, and it may take several years of work for a team of engineers. Moreover, innovation and improvement of logic synthesis are not a goal of this thesis project. More information about how synthesis software is used are provided in Chapter 5.

Although the choice of granularity and the subsequent issues belong theoretically and functionally to the design extraction phase, they have been discussed in this section because of the relation they have with partitioning and placement algorithms.

Chapter 4

Reuse and Dynamic Reconfigurability

As pointed out at the beginning of Chapter 3, an attempt to find a static implementation of an application on a given multi-FPGA architecture can fail, due to strict area constraints. In such case, a blocks reuse technique is adopted in order to make the implementation of the application feasible on the architecture. Since the reuse of blocks - in some cases carried out through the dynamic reconfiguration of the blocks interconnections - inevitably introduces some extra delays in the execution of the application, a methodology which is in charge of minimizing such extra time is needed, and constitutes the topic of the present chapter.

The chapter is organized as follows. First, Section 4.1 provides motivations behind the introduction of hardware blocks reuse. Then, a methodology to deal with reuse choices is described in Section 4.2. Section 4.3 outlines a workflow for the design of circuits that exploit blocks reuse, which embeds the methodologies explained in the previous subsections.

4.1 Design blocks reuse

In this section the concept of design blocks reuse is introduced. First, some precise definitions are provided in Subsection 4.1.1. Then, the arguments that justifies the adoption of reuse in a multi-FPGA system are discussed in Subsection 4.1.2. Subsection 4.1.3 defines the problem the proposed methodology deals with and Subsection 4.1.4 briefly describes some architectural scenarios suitable to implement blocks reuse.

4.1.1 Definitions

First, the notion of block reuse has to be formally defined. Consider the structural representation of a circuit resulting from any cut of the design hierarchy, as described in Subsection 3.2.1.1. For defining precisely what reuse is, it is necessary to enrich the definition of structural representation given before. What is needed is to provide each net of the structure with the information of the ports of the terminals which are actually connected. Therefore, we can think a net belonging to a structure as a list of pairs $\langle\langle t_1, p_1\rangle, \langle t_2, p_2\rangle, \ldots, \langle t_n, p_n\rangle\rangle$, where port p_i belongs to the interface of terminal t_i, for any i between 1 and n (recall from Subsection 3.2.1.1 that a terminal of a net is a hardware block). The normal situation of a static design is that each port of each block is connected to exactly one net. On the other hand, it can be claimed that a design block is *reused* in a structure if each of its ports is assigned to more than one net. This naturally generates one big concern, since ports cannot normally be assigned to more than one net. To solve this problem, *time* has to be introduced in the discussion. By seeing the structure as an entity which evolves in time, we further enhance the previous definitions to capture the nature of designs where blocks are reused. The key idea is to see nets as temporary connections, which can be added and deleted at any given moment of time. A net with such feature is called *dynamic net*. To formally indicate this feature, a time interval $[t_{start}, t_{finish}]$ is assigned to each dynamic net.

At this point, the informal and incomplete definition given above can be corrected by introducing the notion of *dynamically-interconnected structure*.

Definition 4.1. *A **dynamically-interconnected structure** is a structure in which nets have the capability of being added and deleted at any given moment of time. Such nets are called **dynamic nets**. A port cannot be connected to more than one dynamic net in the same moment.*

Definition 4.2. *A block of a dynamically-interconnected structure is **reused** if at least one of its ports is connected to a different dynamic net in two different instants of time.*

From this point on, the following nomenclature is used: each block actually present in a structural representation is called simply block or *instance*. Each instance is the actual realization of a *component*. More than one instance of the same component can be present at the same time in a structure. More concisely, each block is an instance of a component.

4.1.2 Motivations

It is straightforward that the reuse of blocks in a dynamically-interconnected structure allows to reduce the area needed by the implementation of the input application. As a matter of fact, once provided the circuit with the capability of dynamically modify the interconnections between blocks, then it is evident that the same implementation of the block can be used for several instances of the same component in different instants of time. This effect may be desirable in many situations. In our workflow, the primary reason for which blocks reuse is useful is to implement on a given multi-FPGA architecture an application which statically does not fit into it. This is the situation this thesis work is mostly focused on, therefore the following sections will deal with this scenario.

Nevertheless, other reasons could justify the adoption of blocks reuse. Imagine the case in which a given logic circuit fits on an architecture, but the number of

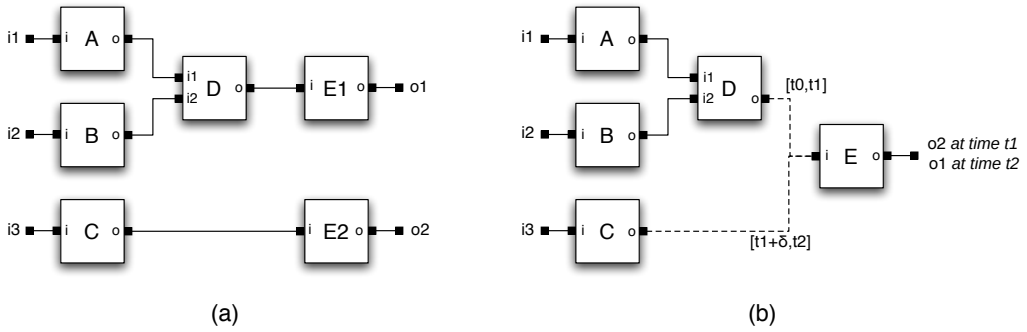


Figure 4.1: Example of a structural circuit (a) and a possible implementation which exploits reuse (b).

FPGAs which are actually used is wanted to be minimized; blocks reuse represent a technique to address such problem. Another scenario is the one in which two big logic circuits have to be implemented on the same multi-FPGA architecture: in such a case the problem would be to decide whether the sequential exclusive implementation of the two circuits on the architecture provides higher performance than a simultaneous one, in which the two applications could also share design blocks through their reuse.

4.1.3 Problem statement

At this point, it is necessary to define the problem addressed in the next sections. Informally speaking, the problem is to decide which instances of the input structure are mapped to which blocks at the implementation level. To provide a more precise definition of the problem, the key idea is to view the structural representation of the logic circuit as an abstract functional representation. What is pursued is the creation of a dynamically-interconnected structure, and of a mapping function between each design block of the input "abstract" circuit and a block of such structure.

Figure 4.1 provides a simple example of a structural circuit and a dynamically-

interconnected structure which implements it. The blocks *E1* and *E2* of Figure 4.1(a) are two instances of the same component. A single module is used in to implement both instances in Figure 4.1(b), with the net $\langle\langle C,o\rangle,\langle E,i\rangle\rangle$ present in the time interval $[t0,t1]$ and the net $\langle\langle D,o\rangle,\langle E,i\rangle\rangle$ present in the interval $[t1+\delta,t2]$, where δ is the time needed for dropping one net and adding the other.

The problem to be faced is somehow similar to the traditional resource-bounded *assignment* and *binding* operations performed to map a data-flow graph application onto a hardware architecture with a finite number of resources of each given type, such as adders, multipliers and so on. The main difference depends on the fact that in the problem addressed requires the decision of *which* are the "resources". From a slightly different point of view, the problem can be defined as composed of two points:

- Which blocks of the input representation should be reused?
- How many instances of such blocks may be present in the dynamically-interconnected structure being created?

The second of these questions takes into account the fact that it is not mandatory to implement exactly one instance of the blocks which are reused. To give an example, if four instance of a component are present in the input structure, it could be the case that the ideal solution is to use two of such blocks in the dynamically-interconnected structure being created, each one reused twice. This evidently implies a wider solution space to be explored by the algorithm.

The above questions have to be addressed under some constraints and with the goal of optimizing a cost function. The constraints are naturally represented by the features of the target architecture. The considered cost function is the total execution time of the system.

4.1.4 Architectural scenarios

This subsection aims at defining what are the types of multi-FPGA architectures which are potentially able to host a multi-FPGA system which adopts blocks reuse. The complete design of such architecture is beyond the aim of this work, and the goal of this section is to concisely show that the realization of dynamically-reconfigurable multi-FPGA systems is fully possible.

At first glance, it is evident that hard-wired topologies are not suitable for implementing a dynamically-interconnected structure. Actually, one solution would be to look at the architecture as a network of hops, and perform information transmission in a packet-switching fashion. Each packet of data would carry the information about its target block, and therefore dynamic interconnections would be fairly easy to be implemented. Such solution implies that each hop (FPGA) in the network is provided with some logic to handle the routing of packets. Due to the delays in handling the packets and the dimension of packets headers that would probably be larger than the useful data, this solution seems not practicable. For this reasons, hard-wired topologies are not a good solution to implement dynamic interconnections.

On the other hand, crossbar and partial crossbar topologies seem to provide the needed flexibility: the reprogrammability of routing chips can be exploited to implement the changes in interconnections. In particular, we want these interconnections to be dynamically modified, meaning that they are asked to change while the system is running. Since a methodology for dynamic reconfigurability on FPIDs is beyond the scope of the present work, FPGAs are considered as routing chips. An approach for the dynamic reconfiguration of portions of FPGAs has already been briefly described in Section 1.3.1. Since this approach deals with module-based reconfiguration (Section 1.3), the following solution is considered: the routing FPGAs contains several modules, each one implementing a *switch-box*, which essentially provides an arbitrary pattern of interconnections between any of its its I/O ports. Each switch-box can be reconfigured while the others

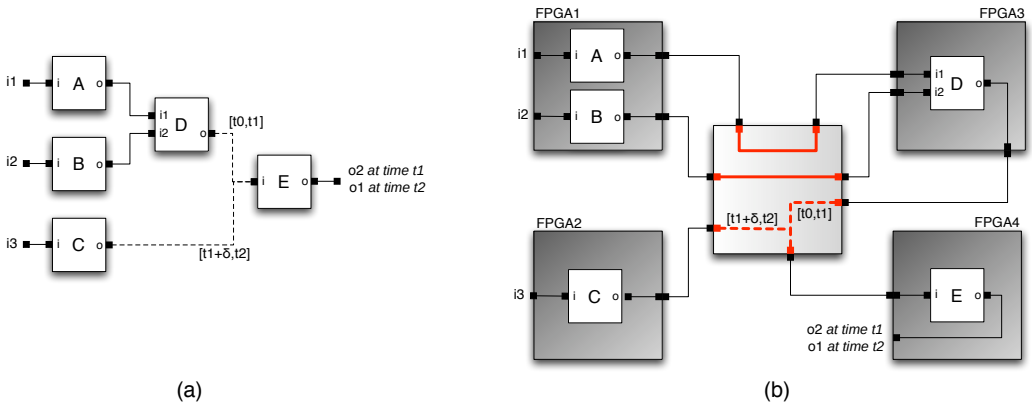


Figure 4.2: A dynamically-interconnected structure (a) and its possible implementation on a crossbar multi-FPGA architecture.

continue to provide the implemented interconnections. Therefore, each time a net has to be removed and a new one has to be added the relative switch-box is reconfigured. Figure 4.1 takes the example shown in Figure 4.1 and show one possible implementation on a crossbar multi-FPGA architecture.

A bus topology allows mutable interconnections between blocks to be naturally implemented, even more easily than the crossbar or partial crossbar topologies. As a matter of fact, if the FPGAs that constitute the architecture are connected one each other through a bus, it is enough to establish a connection between any pair of FPGAs when it is needed. This model of communication has been already widely exploited for the creation of system-on-chip's (SoCs). To provide an example, Xilinx EDK [46] allows the creation of SoCs with modules connected through local busses. These busses belong the the IBM CoreConnect [47] family, and are provided with a communication protocol which allows to establish a connection between any pair of modules connected to the bus. A similar solution can be easily implemented on a multi-FPGA architecture: in such case, the bus would be constituted by physical wires on the board, and each partition would be provided with a communication interface. A tool which automatically

creates the interface for busses belonging to the CoreConnect family is already available [48], therefore this step would not require extra work by the designer.

Although we claimed that fully hard-wired architectures are not suitable for implementing dynamically-interconnected structures, some hybrid solutions can be designed in order to provide a trade-off between the flexibility of crossbar and bus and the speed of hard wires. The hybrid topologies proposed in [6] and described in Section 1.2.1 represent an example of this kind of architectures. In such case, static connections can be routed using hard wires, while the mutable ones can be implemented through routing chips.

4.2 Design blocks reuse methodology

In this subsection a methodology for blocks reuse is described. First of all, the problem to be addressed is expressed in a semi-formal way (Subsection 4.2.1), in order to make it suitable to be treated analytically. The result of such discussion is that the problem can be viewed as composed of two subproblems, namely how to find isomorphic clusters and how to choose which ones to reuse. The description and a possible solution to the first one is given in Subsection 4.2.2, while the second subproblem is treated in Subsection 4.2.3.

4.2.1 Problem analysis

The problem of the reuse of circuit parts has been informally defined in Subsection 4.1.3. In this section, the problem is analyzed to define its peculiarities and outline a way to cope with them. First of all, the parts of the input application that can be considered for being used more than once have to be identified. One natural and straightforward choice would be to consider each design block as a reusable portion of the circuit. In a hierarchical scenario as the one we are considering, it is possible to consider each block at each level of the hierarchy. Of course, some

constraints have to be applied in this case, and they are discussed in Subsection 4.2.3. Nevertheless, it can be noticed that restricting the choice of which portions are considered for reuse to single blocks is a limitation. As a matter of fact, any pattern of one or more blocks which is repeated more than once in the circuit that has to be implemented should be considered. This problem of finding repeated patterns has already been studied. In graph theory, these recurrent patterns are called *isomorphic subgraphs*. Subsection 4.2.2 proposes a solution to this problem which takes advantage of the hierarchical structures that are considered in this thesis.

Once these patterns have been identified, it is necessary to choose, among them, which ones are suitable for being reused. For this sake, a quality measure has to be defined, which makes one solution preferable to another. In order to understand what is a "good" choice for reusing blocks, it is important to recall the reasons why it is necessary to introduce reuse in the circuit under consideration. In Subsection 4.1.2 it has been stated that reuse is necessary in order to make a circuit fit on a multi-FPGA architecture whose area is insufficient to host it statically. Therefore, a guidance measure for driving the reused block selection is the estimation of the quantity of area which has to be saved by the dynamically-interconnected structure. This quantity is provided by the global layout step, in case of failure, as a percentage of the total area of the circuit. Another remarkable fact which has to be considered is that the reuse of components tends to increase the execution time, because some operations are forced to be executed in sequence on the same block, rather than being possibly performed in parallel, and because the reconfiguration of interconnections needs some time to be performed. Taking into account these facts, it can be argued that a solution which does not save enough area is useless, while one that saves much more area than needed is likely to cause high execution times. In other words, the algorithm should try to save the least amount of area which allows the circuit to be implemented on the multi-FPGA architecture, thus providing acceptable timing results. This part of the

problem is addressed in Subsection 4.2.3.

4.2.2 Finding isomorphic clusters

The first part of the problem deals with the identification of isomorphic structures in the input circuit, so to provide a wide range of possibility for choosing the best structures to be reused. As a convention, any portion of a structure will be called *cluster*, implying that it is generally a group of design blocks. The problem of extracting isomorphic subgraphs has already been object of research, and an extensive discussion about this topic is found in [49]. A formal definition of isomorphic cluster, derived from the previously cited work, will help in the subsequent part of this section.

Definition 4.3. *Two structures $S_1(B_1,N_1)$ and $S_2(B_2,N_2)$, where B_i and N_i are respectively the set of blocks and the set of nets composing structure S_i, are said to be* ***isomorphic*** *if they are connected and*

1. *There exists a bijection $f : B_1 \rightarrow B_2$ such that $b \in B_1$ and $f(b) \in B_2$ are two instances of the same component and*

2. *There exists a bijection $g : N_1 \rightarrow N_2$ such that $n = \langle\langle b_1, p_1\rangle, \ldots, \langle b_k, p_k\rangle\rangle \in N_1$ implies $g(n) = \langle\langle f(b_1), p_1\rangle, \ldots, \langle f(b_k), p_k\rangle\rangle \in N_2$ where $\{b_1, \ldots, b_k\} \subseteq B_1$*

A cluster can be identified by the set of the instances it contains. Intuitively speaking, two clusters are isomorphic if they comprise the same blocks and the same nets connecting them. Trivially, all the instances of a single component are isomorphic clusters, since the first condition is true and the second is trivially satisfied. A simple example of isomorphic structures is provided in Figure 4.3, where the letters indicate the component each block is instance of. The two clusters shadowed in grey, $c1 = \{B1, D1, E1\}$ and $c2 = \{B2, D2, E2\}$, are isomorphic.

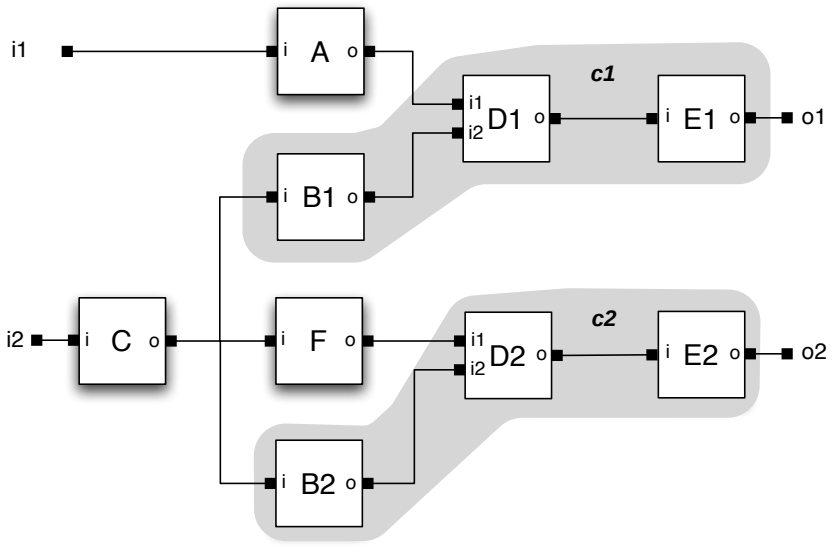


Figure 4.3: Example of isomorphic clusters.

Moreover, it can be noticed that each *sub*-cluster contained in $c1$ has one relative isomorphic cluster in $c2$. As a matter of fact, the cluster $\{B1, D1\}$ is isomorphic to $\{B2, D2\}$ and $\{D1, E1\}$ is isomorphic to $\{D2, E2\}$.

Finding all the isomorphic clusters of a graph is a NP-complete problem [49]. Therefore, it is better to focus on an algorithm which is able to find *some* isomorphic clusters. As a general guideline, it would be better to identify isomorphic clusters such that they are suitable to be subsequently treated as blocks in partitioning and placement on the target architecture. The main goal of partitioning and placement is to minimize the estimated length of external interconnections. Choosing clusters such that their amount of external communication is low causes the subsequent global layout tasks to be less expensive and to achieve better performances, since such clusters are composed by strongly interconnected groups of blocks. The proposed approach aims indeed at finding clusters with such quality.

The advantages of exploiting hierarchy in global layout tasks has been described in Subsection 3.3.1.1. One of them was due to the regularity information

that hierarchy provides. Such notion of regularity can help also in finding isomorphic clusters. In literature, a similar idea of exploiting regularities for the reuse of computational elements can be found in [43], where the authors deal with the assignment and binding of operations to computing elements. The exploitation of regularities allows in such case to find a solution which uses the minimum amount of interconnecting elements (e.g. wires, multiplexers, etc...), thus reducing power consumption. The key idea is to *reuse* the same computational elements and interconnections for repeated operational patterns. On such basis, the proposed approach uses the regularity information given by the hierarchy to identify isomorphic clusters which minimize external connections.

The proposed algorithm, called ISOMORPHIC-CLUSTERS, is similar to a normal bottom-up clustering, hence the name *clusters* for defining portion of structures. Based on the definition of isomorphic structures given above, the notion of *type* of a cluster is here introduced: two clusters belong to the same type if they are isomorphic structures. In other words, the relation of isomorphism generates equivalence classes[1], and such classes determine the type of the elements they contain.

The algorithm starts from the leaves of the hierarchy, which represent the circuit at the lowest level of granularity. Each leaf is considered as a cluster. The starting point is that instances of the same component represent isomorphic clusters, hence they are assigned to the same type. Then, a given metric (a suitable set of metrics will be defined in Subsection 5.2.3.1) identifies which pair of clusters has to be collapsed, thus generating a new cluster. The choice of such pair of clusters is subject to area and I/O pin constraint: if the new cluster resulting from their collapsing has a dimension higher than the area available on a single FPGA, or its amount of external communication requires more I/O pins that the ones available on a single chip, then the new cluster cannot be formed, and another pair of

[1]It is simple to see that isomorphism is an equivalence relation, since the properties of reflexivity, symmetry and transitivity are satisfied.

clusters is considered. When a new cluster is formed, two situations are possible:

1. If the nodes in the collapsed pair belong to the same parent, occurrences of such cluster type are searched in the hierarchy to apply the same transformation, if the corresponding nodes have not been involved in other collapsing operations before. Then, the same type is assigned to these newly created clusters. Moreover, the resulting clusters are added as children of the blocks that contained the merged clusters.

2. If the nodes in the collapsed pair do not belong to the same parent, they are clustered and removed from the current position, and the resulting cluster is added as a child of TOP, which is a reasonable choice, being such cluster unique and therefore not involved in any regularity pattern.

Moreover, if the collapsing causes a block to be left with a single child, such block can be removed since it does no longer carry useful information about any regular patterns it might contain. This process iterates until a single cluster, the *TOP*, is obtained, or no more new clusters can be formed due to the area and I/O pins constraints.

The representation of the hierarchy which results from the application of a clustering algorithm is called *dendrogram*. An example of the execution of the described algorithm on a simple structure is provided in Figure 4.4.

The obtained dendrogram constitutes the information that was searched. It is important to point out that the obtained clusters are good with respect to the objective of minimizing the cutsize in a subsequent partitioning step, since they have been obtained from a clustering which uses closeness metrics for creating clusters.

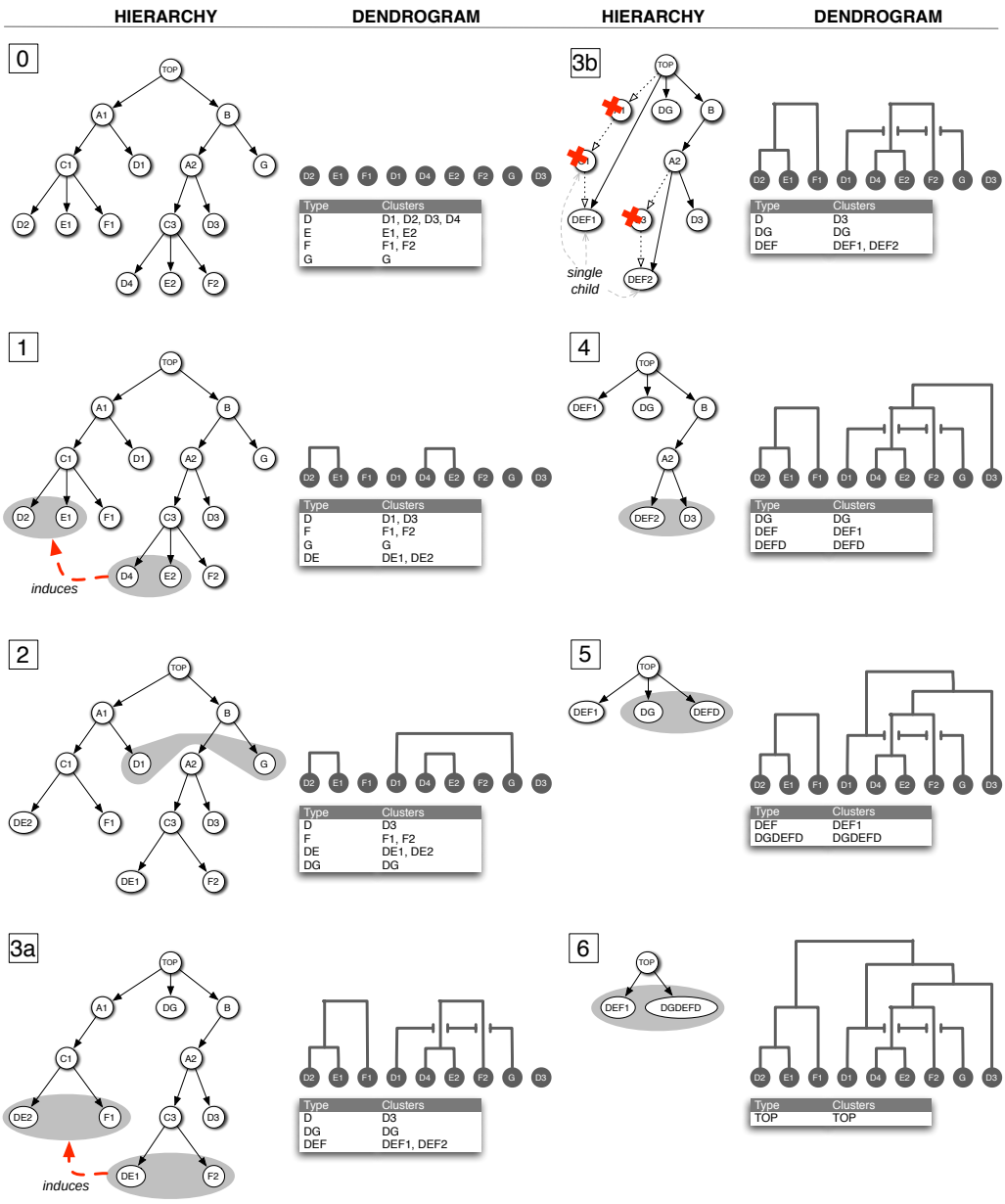


Figure 4.4: Some steps of the execution of the isomorphic clustering algorithm on a sample hierarchy.

4.2.3 Selecting the clusters to be reused

What is produced from the ISOMORPHIC-CLUSTERS algorithm is a hierarchy of clusters (the dendrogram), and the types associated with them on the basis of the extracted isomorphisms. At this point, it is necessary to exploit such information to produce a (flat) dynamically-interconnected structure. There are two factors which contribute to the quality of such resulting structure. First, as said in Subsection 4.2.1, the size of the resulting circuit must be small enough to allow the circuit to be mapped on the target architecture, but big enough not to waste too much space on the chips. Moreover, it is known from [49] that reconfiguration is an operation which requires a large amount of time, so the choice of which blocks are reused should be targeted at minimizing the time needed for reconfiguring their interconnections.

In the following, a formal definition of this problem is provided. Initially, the hierarchical structure of the input (the dendrogram) is not considered. The input of the problem is therefore a flat structural representation, which may be seen as produced by a given cut of the input hierarchy; each block of such circuit is a cluster which belong to a specific type. We define the set of types of the clusters that belong to the input structure as $T = \{t_1, t_2, \ldots, t_N\}$. Notice that the elements of such set are the types, intended as recurring patterns, and not their instances. Three functions are defined on such set:

- $a(t_i) : T \rightarrow \mathbb{N}$, which returns the area occupied by a cluster of type t_i;
- $n(t_i) : T \rightarrow \mathbb{N}$, which returns the number of instances of a cluster of type t_i;
- $p(t_i) : T \rightarrow \mathbb{N}$, which returns the total dimension in bits of the ports which constitute the interface of a cluster of type t_i (i.e. the amount of external communication).

A number, $A \in \mathbb{N}$, corresponds to the total area occupied by the chip when no block reuse is applied. The number $s \in \mathbb{N}$ is the percentage of the total area that

has to be "saved" in order to make the circuit fit on the target architecture. This percentage is the result of an estimation carried out after the global static layout is attempted.

A solution to the problem is represented by a function $m(t_i) : T \rightarrow \{1, 2, \ldots, n(t_i)\}$ that represents the number of clusters of type t_i which are instantiated in the resulting dynamically-interconnected structure. The clusters having type t_i are not reused when $m(t_i) = n(t_i)$, while they are reused a maximum number of times when $m(t_i) = 1$.

The reconfiguration time of the interconnections in case of reuse of a cluster is reasonably supposed to be proportional to two quantities: the number of reconfigurations that are needed (i.e. $m(t_i) - n(t_i)$) and the amount of the external communication (i.e. $p(t_i)$). This latter proportionality is the result of the following chain of implications. The reconfiguration time is proportional to the amount of area which has to be reconfigured. The amount of area that has to be reconfigured on the routing chip of a crossbar (or partial crossbar) architecture is directly proportional to the number of interconnections which has to be re-routed. This number is equal to the I/O pin requirements (i.e. the amount of external communication) of the block to which such interconnections are attached. Therefore, we impose

$$t_{rec}(t_i) : T \rightarrow \mathbb{N} := k * [m(t_i) - n(t_i)] * p(t_i)$$

to be the estimation of the reconfiguration time for clusters of type c_i. Notice that in case $m(t_i) = n(t_i)$, which means that no reconfiguration is required, the estimated reconfiguration time is equal to zero.

At this point, the problem is reduced to the choice of the solution function $m(t_i)$ such that the total reconfiguration time

$$T_{REC} = \sum_{t_i \in T} t_{REC}(t_i)$$

is minimized and the following area constraint is met:

$$\sum_{t_i \in T} a(t_i)m(t_i) < A(1-s)$$

which states that the resulting instances' total area must be less than the portion of the original circuit area that is supposed to fit onto the target multi-FPGA architecture. As it is possible to see, the request of using as most of the available area as possible is not directly addressed. It is anyhow straightforward that the minimization of the total reconfiguration time implicitly results in optimizing area usage, since the more clusters are not reused - thus increasing area occupation - the less reconfiguration time is required.

In order to exploit the information provided by the design hierarchy, the proposed approach for blocks reuse decisions is based on a two step procedure. The idea is to find a way to cope with the problem without the complexity introduced by the hierarchy constraint, which nevertheless brings important information which are not wanted to be wasted. For this reason, a first phase attempts at finding a suitable cut of the hierarchy, as defined in Subsection 3.2.1.1. In this way, the second phase works on the resulting flat structural representation.

The second step of the process is described first. The blocks reuse problem on flat structures exposed above can be proved that to be *NP*-complete. Despite this, a ILP (*Integer Linear Programming*) solution to the problem is capable to find the optimum in a very short time also for big structures, as proved in Chapter 6. The implemented ILP model will be described in detail in Chapter 5. Being the running times of such algorithms very low even for fairly big structures, it is possible to iterate the execution of the algorithm on flat structures generated by different cuts of the dendrogram resulting from the application of the ISOMORPHIC-SUBGRAPH algorithm, so that providing a better result.

The problem that has still to be addressed is how to identify some suitable cuts of the dendrogram to produce the flat structures for the subsequent phase. The solution consists in the generation of all the structures resulting from *horizontal* cuts

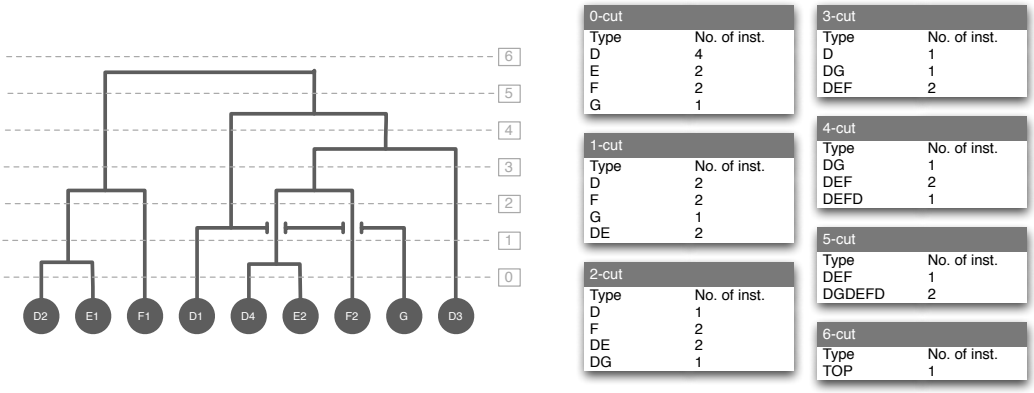


Figure 4.5: Example of extraction of horizontal cuts.

of the dendrogram, as shown in the example in Figure 4.5. This example works on the dendrogram produced by the application of the ISOMORPHIC-CLUSTER algorithm provided in Figure 4.4.

The number of such cuts is in the worst case equal to n, and this happens when the area and I/O pin requirements do not prevent the clustering process to eventually results in a single cluster and no regularity is detected throughout the iterations. Since the extracted flat structures result from cutting horizontally the dendrogram, the clusters they are composed of are internally strongly interconnected. As a matter of fact, any horizontal cut reflects the sequence of collapsing operation carried out by the ISOMORPHIC-CLUSTER algorithm, whose goal is to reduce inter-cluster communication.

4.3 The reuse workflow

The results of the previous sections provide a methodology for identifying which blocks or group of blocks to reuse for optimizing area usage and time expenses. Such result is representable through a dynamically-interconnected structure, defined in Subsection 4.1.1. At this point, such structure have to be partitioned,

placed, and routed on the target architecture. The resulting workflow is showed in Figure 4.6.

For partitioning and placement, the algorithms used for the statical global layout phase can be reused. As a matter of fact, the goal is to minimize the estimated wire length, so it is a good choice to consider all the nets in the dynamically-interconnected structure as always present in the circuit and apply a statical multi-FPGA partitioning and placement process. For what concerns routing, the problem is more complex, as the "dynamic" routing of nets has to be specifically addressed. A specific routing algorithm is not provided in this thesis, and the development of a routing methodology, both for the static and dynamic case, is proposed as a future work.

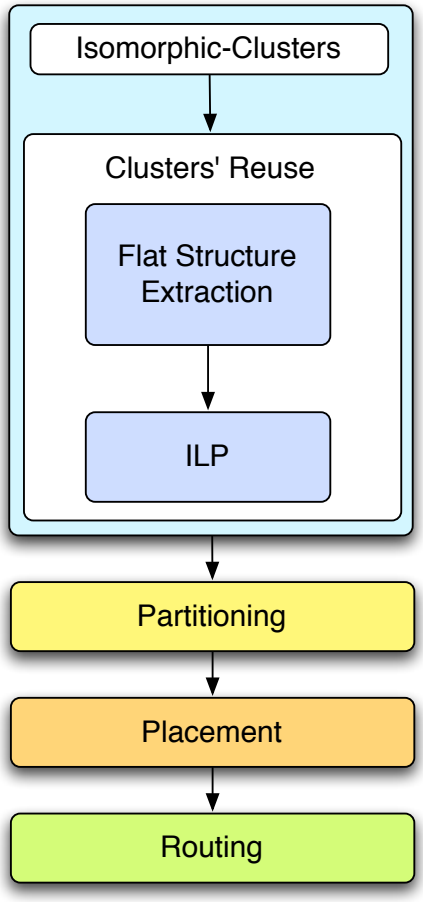


Figure 4.6: Workflow for the reuse of structural blocks.

Chapter 5

Implementation

In Chapter 3 and 4 the proposed workflow for the design of multi-FPGA systems has been presented, and methodologies to deal with the different phases of the design have been introduced and motivated. The present chapter has the purpose of describing *how* such methodologies have been implemented in practice. In order to provide a systematic presentation, each section of the chapter deals with a phase of the workflow. Section 5.1 explains how the design extraction phase is carried out, by providing details on the data structures used for internally represent a structural and hierarchical design. Section 5.2 describes the implementation of the algorithms which carry out the tasks needed by the static global layout phase. Section 5.3 explains how the algorithms which deal with the reuse of structural portions of the input circuit work.

5.1 Design extraction

The extraction of the design structure and hierarchy is the initial phase of the proposed workflow for multi-FPGA design. Besides being embedded in such flow, the extraction framework is also useful in other applications, therefore it can be viewed as a standalone tool, called StructGen. Starting from a VHDL de-

scription, the framework produces as output a specifically designed data structure which integrates information about hierarchy and structural representation of the input application.

The next part of this section is organized as follows: Subsection 5.1.1 briefly introduces the basic aspects of VHDL structural design, while Subsection 5.1.2 describes the data structure constituting the output of the framework. Subsection 5.1.3 describes how the framework works and has been implemented.

5.1.1 VHDL structural design

VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a language for the description of digital circuits, created to fulfill several needs in the design process [50]. Though a detailed description of VHDL syntax is out of our scope, it is important to introduce what are usually called VHDL coding styles or paradigms [13]. There are three different coding styles. *Dataflow* statements are instructions that are executed concurrently one each other. They do not need any supporting construct, and are usually assignments and simple logic operations. *Behavioral* instructions represent a more abstract way to describe hardware, and are contained in portions of code called *process*es. If two or more processes are present in a VHDL description, they are executed concurrently one each other, while the instructions inside each process are executed in sequential order. Similarly to a software programming language, in a process it is possible to use control sequences such as conditional statements and loops. The third style is *structural* VHDL, which provides constructs to declare and instantiate components. This is very similar to the declaration and the instantiation of objects in high level software programming languages. A component is *declared* by describing its name and its interface, i.e. its I/O ports. Once it has been declared, a component can be *instantiated* as many times as desired. The instantiation of a component consists in indicating an instance name and providing an actual mapping for its I/O ports.

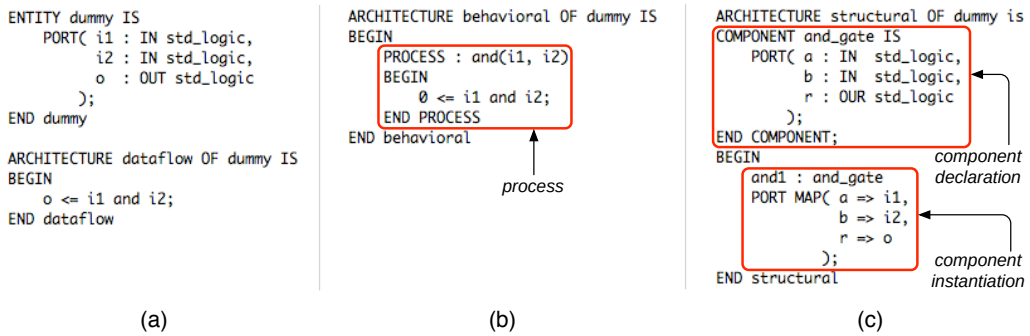


Figure 5.1: Example of VHDL coding styles: dataflow (a), behavioral (b), and structural (c).

Instances communicate one each other using signals, that have to be declared as well. Figure 5.1 provides an example of three equivalent VHDL descriptions, one for each coding style. The three styles are absolutely not mutually exclusive: VHDL specifications are usually written mixing all three styles. A simple example is the use of dataflow instructions as glue logic for structural components.

As the name says, the structural coding style is used to describe structural designs, where the circuit functionality is decomposed into sub-circuits. These, in turn, can be decomposed into smaller sub-blocks, and so on recursively. This is one of the main reasons why VHDL has been chosen as the input format of the multi-FPGA design workflow. Nevertheless, as hinted above, it is hard and usually counter-intuitive to describe circuits in pure structural style. An analysis of existent VHDL designs shows that processes and components instantiations are likely to be found in the same description; one reason for this fact is that processes are suitable for implementing control on such structural blocks. Moreover, it is obviously impossible to design a circuit using *only* structural description. The reason is that, even in the extreme case in which the design is structurally built up from - or decomposed down to - single gates components, the description of such gates has to be found somewhere, and it has to be described by some dataflow or sequential instructions.

5.1.2 Representing a hierarchical design

This subsection describes the data structure created for the representation of hierarchical VHDL circuits. The aim is not only to develop a representation which fits well the needs of the particular workflow proposed in this thesis, but to provide a general and complete representation, suitable to be used in a number of applications. It has been implemented using C++.

The data structure is composed of several levels, from the external interfaces to the core structures, in order to hide the implementation details and provide a robust and easy-to-use Application Programming Interface (API). Such interface is implemented through a class called **Structural**. Such class is a sort of container of all the elements which represent the hierarchical VHDL design and provides all the methods needed for creating and handling it. The Structural class basically contains two objects: a component library, called **CompLib**, and an indexed table - implemented as a C++ *map* - which contains all the circuit structures the hierarchy is composed of. As a matter of fact, each non-leaf node of the hierarchy corresponds to a structural graph which contains several sub-blocks (the *children* of the node). The structural graphs have been implemented using the Boost Graph Library (BGL) [51]. BGL provides graph objects which are easily customizable through C++ templates: in this case, the **StructGraph** data type has been defined, with adjacency list chosen as the graph internal implementation. This choice reflects the fact that the circuit structures have usually a relatively low number of edges, and so they result in sparse graphs. Hence, the table contained in the Structural class maps integer indexes to StructGraph objects. In order to understand what has been just described, an intuitive example is provided in Figure 5.2.

Each StructGraph contained in the Structural contains vertices and edges. BGL allows to associate to these graph elements C++ classes. The class associated with a node is called **Instance**, while the one associated with an edge is called **Wire**. Therefore, it is possible to think of a StructGraph as a set of Instances which are connected by Wires. An Instance object contains several attributes. Most no-

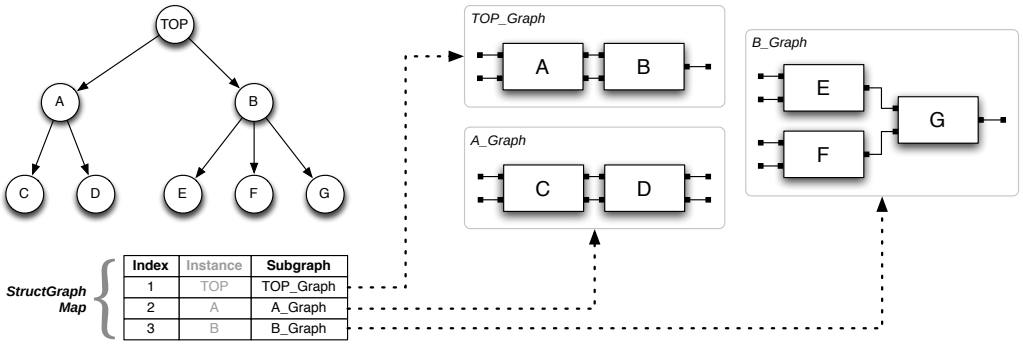


Figure 5.2: Representation of a simple hierarchical structural design.

tably, it maintains a reference to the **Component** it is instance of. It is easy to see here the parallelism between the VHDL structural description style and the developed data structure. The Components are contained in the CompLib that was cited before. The reason why components are not themselves elements of the instances is simply that there could be more than one instances of a single component, and in this way there is not need to store the same information several times and consistency issues are avoided. Another remarkable fact is that the CompLib is referenced by Structural, and not contained in it. In this way, more flexibility is achieved because CompLib can be reused in several Structurals. Instances are identified inside the StructGraph's through *VertexDescriptor*'s provided by the BGL. Therefore, to uniquely locate an instance in a Structural representation, a pair of values is needed: the table index of the StructGraph which contains it and the VertexDescriptor which identifies the instance inside such graph. Such pair of values is provided by an object named **iLocator**. Hence, an Instance contains an iLocator which identifies its parent in the hierarchy. An important attribute of Instance is the *black-box* Boolean flag. If it is true, then the instance is a leaf of the design hierarchy: in other words, it cannot be decomposed in sub-instances. In such case, another field contains the name of the VHDL file with the descrip-

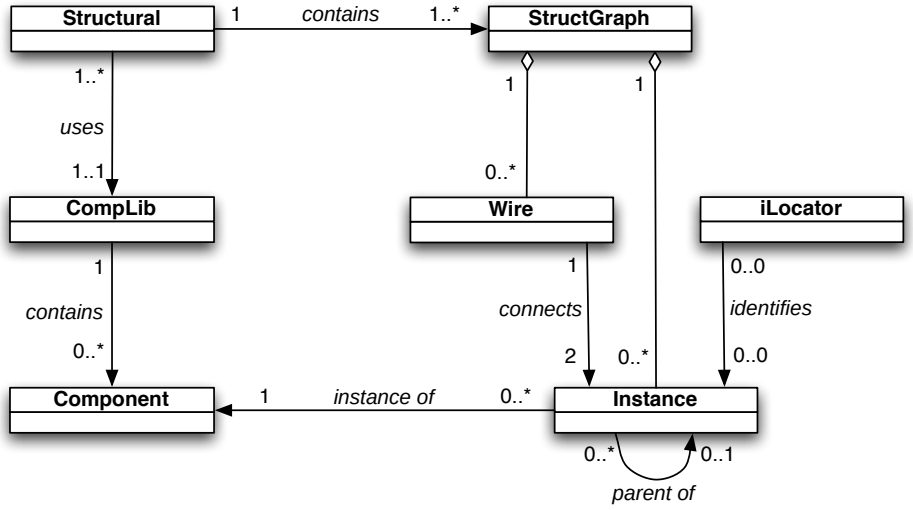


Figure 5.3: UML Class Diagram of the data structures for representing hierarchical designs.

tion of the leaf module. If it is false, the instance is an intermediate module in the hierarchy and it contains an identifier of the StructGraph which represent its sub-structure. The notions introduced so far are summarized in the UML class diagram of Figure 5.3.

5.1.3 The extraction framework

The framework for the extraction of structural and hierarchical information from VHDL, called STRUCTGEN, is composed by two main phases. First, the VHDL code undergoes a preprocessing procedure, which has the aim of transforming it in a suitable form for the subsequent hierarchy extraction phase. An external module is used to obtain information about the features of the blocks being examined. The structure of the framework is shown in Figure 5.4. The remainder of this section is organized as follows: Subsection 5.1.3.1 explains how the preprocessing is carried out, while Subsection 5.1.3.2 describes how the structural parser works.

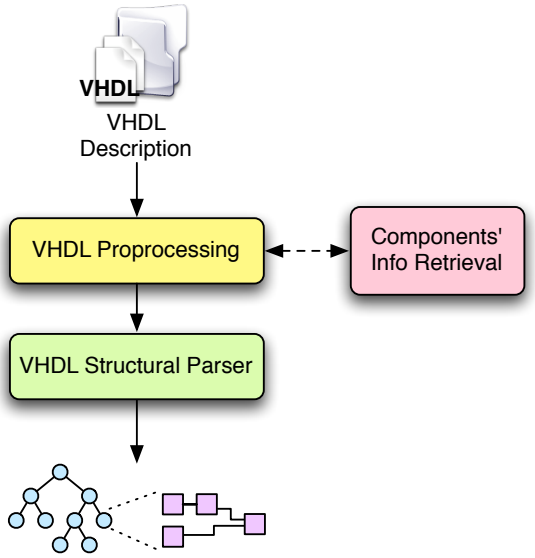


Figure 5.4: Extraction of the design hierarchy: the STRUCTGEN workflow.

5.1.3.1 VHDL preprocessing

As discussed in Subsection 5.1.1, it is nearly impossible to find a purely structural description of a circuit, which uses dataflow and behavioral instructions only in the leaves of the hierarchical tree. A mixed description style is predominant all throughout the hierarchy: for instance, instantiations of sub-components are likely to be placed beside dataflow instructions and processes. The problem is therefore how to treat such non-structural elements to produce a hierarchical tree. It has already been mentioned in Subsection 3.3.2 that the algorithms of the proposed multi-FPGA flow deal with granularity at the the process level, meaning that each process represents a leaf block in the hierarchy. Therefore, processes must be extracted and turned into sub-components. Moreover, dataflow instructions must undergo the same treatment. This can be carried out automatically without changing the semantic of the hardware description. The case of creating a sub-component out of a process is examined: what is necessary is to detect which

are the Input/Output ports of the new component. Informally speaking, it can be said that the data that are "read" by the process should become IN ports of the new component, while the data that are "written" should become an OUT port. More precisely, the following elements become IN ports:

- Signals and ports listed in the *sensitivity list* of the process;
- Signals and ports found at the *right* of an assignment instruction;
- Signals and ports which are used in conditional clauses, like *if-then-else*, *case*, or *loop* statements.

The case of OUT ports is simpler, since they are constituted by all the identifiers found on the *left* of assignment instructions.

After the I/O ports of the new component have been detected, a new component with such interface is created in a new file, and the process is copied entirely into it. Such component will be composed only by this single process, and is considered to be a leaf of the hierarchy. For what concerns dataflow instructions, the procedure is almost identical, despite the fact that in such case there is not a sensitivity list and sequential constructs. Similarly to what happens with processes, a component containing only dataflow instructions is considered to be a leaf of the hierarchy and is no further decomposed. A simple example of the application of this sub-component extraction is shown in Figure 5.5.

The preprocessing tool has been implemented using Flex and Bison [52, 53], which are suitable tools for creating parsers and compilers in C/C++. The files belonging to the input VHDL application are parsed singularly, and new VHDL files are generated when transformations like the ones described before are needed. Moreover, when a leaf is detected or created, an external module is called. This module is in charge of retrieving information about the component, such as the dimension (expressed in number of occupied slices) and the operating frequency. This is done by invoking XST [45], a logic synthesis tool for FPGAs by Xilinx.

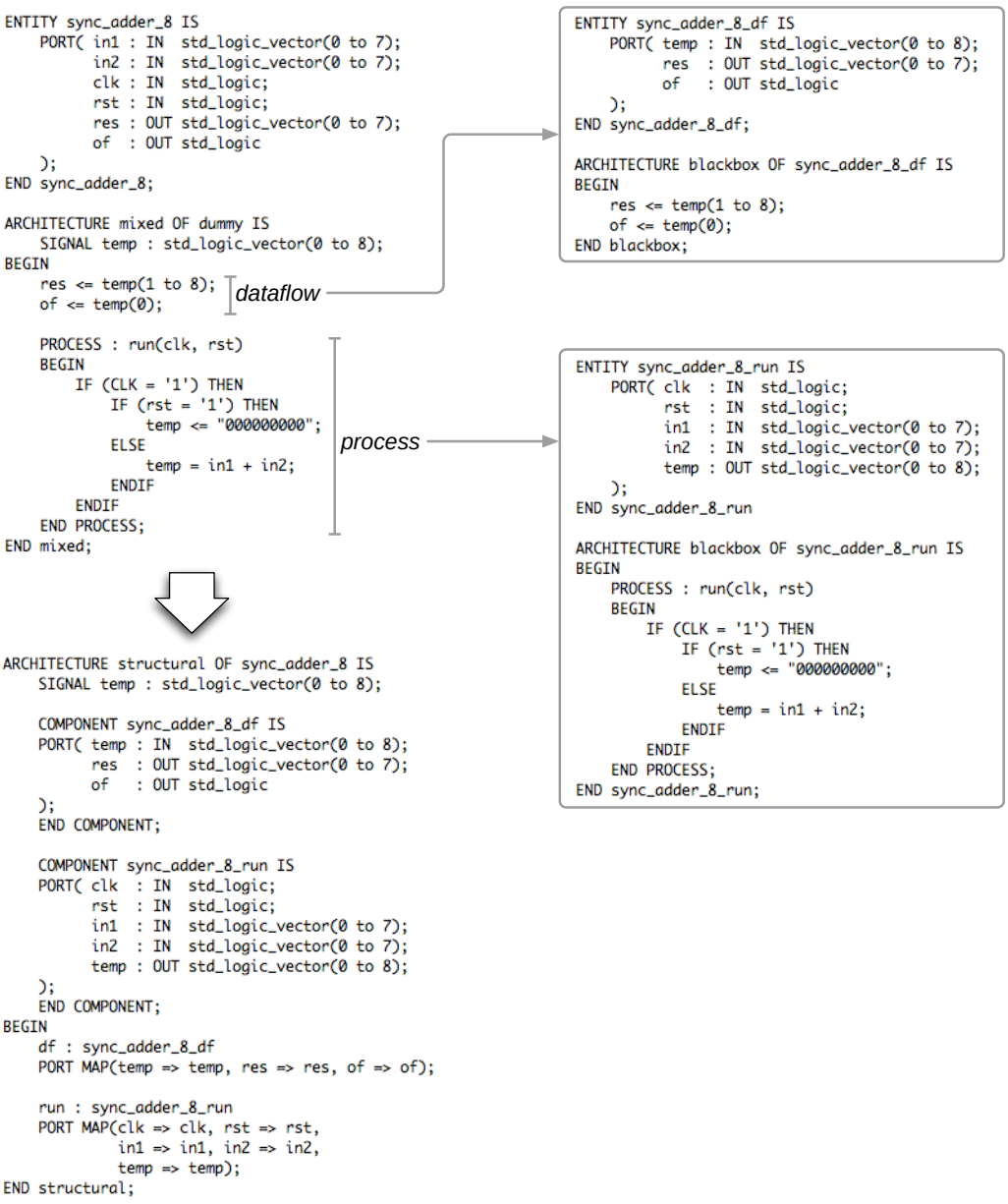


Figure 5.5: Example of VHDL preprocessing.

The reasons of this choices have already been discussed in Subsection 3.3.2. The information retrieved by this module, along with the name and the I/O ports of the leaf component, are stored as a Component object in a component library (CompLib), described in Subsection 5.1.2.

5.1.3.2 VHDL structural parser

Once a VHDL description has been turned into a pure structural design, with the obvious exception of the leaf nodes, it is ready to enter the second phase of the execution of STRUCTGEN. The aim of this phase is to parse the VHDL descriptions and extract information about the structural representation and the design hierarchy.

First of all, a Structural object is created which receives as input a reference to the CompLib, provided by the preprocessing step. Then, the parsing process starts from the file which contains the description of the TOP level of the design hierarchy. An initial entry in the StructGen indexed table (described in Subsection 5.1.2) is created. Basically, when parsing a file, two situations are possible.

If the file contains, in the architecture description, only one process or only dataflow instructions, such component cannot be further decomposed, and the relative instance in the Structural is indeed marked as a black-box.

If the file contains one or more component declarations and instantiations, it means that the current instance is composed by a number of sub-blocks. In such case, an entry in the StructGraph table is created, which corresponds to the sub-graph of the instance being analyzed. One issue rises at this point: consider the sub-structure which corresponds to the current analyzed instance; besides being composed of sub-instances and wires which connect them, there are connections which link the sub-blocks with the ports of the parent block. This type of "hierarchical" interconnection is not captured by the formalism of standard graphs, hence the BGL does not offer a way to implement such connections. The adopted solution to represent such connections is therefore to create, in the StructGraph

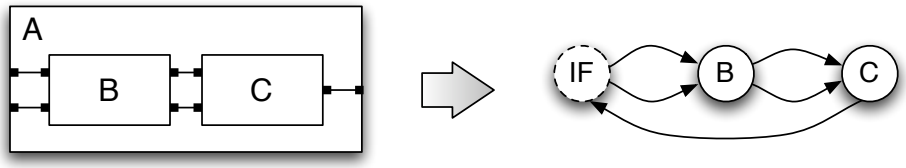


Figure 5.6: Implementation of interface interconnections in a StructGraph.

corresponding to the current instance, a special instance, called *interface* (IF), which has the same I/O ports of the parent block. The wires which are theoretically connected to the I/O ports of the parent block are attached to this "dummy" instance, which is a representation of the I/O interface of the parent block. This particular node has dimension equal to zero. Therefore, every time a StructGraph is created, a dummy interface instance is automatically created inside it. This fact is better explained by the example reported in Figure 5.6.

For each component declaration in the VHDL file, the corresponding I/O ports are locally stored in the parser. Moreover, every time a signal declaration is found, a particular data structure is created for that signal, the use of which is explained later. When a component instantiation is found, a node is added to the current StructGraph, and the corresponding VHDL file name is stored in a queue. The file to be parsed after the current one is identified by the first element of such queue. This mechanism implements a breadth-first parsing of the VHDL files which compose the design hierarchy. Then, the "port map" of the instantiation is analyzed. We call the mapping of each port of the instantiated component *link*. There are two kinds of links: *port-to-port* and *port-to-signal*. The first ones represent a connection between IN (OUT) port of the instantiated component and an IN (OUT) port of the instance whose description is being parsed. When such a link is found, a wire in the StructGraph is immediately added, between the block being instantiated and the interface dummy instance. The other kind of link connects a port of the instantiated component with a signal. As hinted before, every

signal is associated with a data structure, composed by the name of the signal, the identifier of *one* OUT port and a *set* of identifiers of IN ports. This is because, as it represents a connection between two blocks of the VHDL structural description, a signal cannot be "fed" by more than one port, otherwise causing a *double-driven* fault. Therefore, if the link is of type *port-to-signal*, the identifier of such port is inserted in the corresponding record of the signal data structure, depending on the direction of such port. At the end of the parsing of the current file, the signal structures are analyzed to generate the wires which connects two instances of the StructGraph.

5.2 Static global layout algorithms

In this section a description of the algorithms to carry out global layout tasks is provided. To proceed with a clear description, the organization of the section reflects the order in which the approaches has been introduced in Section 3.3. First, an explanation of the data structures used by the algorithms is provided (Subsection 5.2.1). Next, Subsection 5.2.2 describes the simulated annealing algorithm designed for carry out the integrated partitioning and placement task. Subsection 5.2.3 explains how the sequential partitioning and placement approach works, by presenting the algorithms defined.

5.2.1 Data structures

The data structure that represents structural and hierarchical VHDL designs introduced in Subsection 5.1.2 provides a complete description of the elements which compose the circuit. Nevertheless, some of the information contained in it is useless from an algorithmic point of view. In other words, the information carried by the data structure presented in the above section is too much for the execution of the algorithms. Nevertheless, it is necessary, because after the algorithms have

been executed what is asked is to generate an output, possibly in VHDL, which contains all the information about the design. Hence, the solution is to create, from the structure provided by STRUCTGEN, a lightweight data structure which the algorithms works on, but always maintaining a bijection between the elements of such structure and of the original one, in order to be able to re-build, after execution, a complete description of the system.

To fulfill this need, an *Application* class has been created. Such class contains the graph which represents the circuit at the lowest level of hierarchy. The class provides a simple interface to add nodes and edges, and retrieve information. The graph is internally implemented as an adjacency list. Despite this, the adjacency matrix of the graph can be obtained by calling a method, since it represent a more efficient data structure for some algorithms. Each node of the application is identified by an integer number, computed sequentially when adding new nodes.

Moreover, a *Hierarchy* class has been implemented, which provides information about the design hierarchy. The internal representation of the hierarchy is a rooted tree whose leaves corresponds to the nodes in the Application object. The rooted tree is implemented by nodes which contains a pointer to the parent instance and a list of pointers to the children instances, thus providing efficiency in both parent and children retrieval operations.

Since partitioning and placement must be aware of the features of the target multi-FPGA architecture, an *Architecture* class has been created. Such class has several different implementations, dependent on its topology. Therefore, several classes, one for each considered architectural topology, inherit from Architecture. They must all provide the methods declared the Architecture interface. Such methods return information about the architecture, such as the dimension of each FPGA, the number of I/O pins and the estimated distance between a pair of FPGAs. To provide an example, the *MeshArchitecture* class is briefly described. The internal representation of the architecture is a graph, each node corresponding to a FPGA, which implements a 4-neighbour pattern, given as input the number of

rows and of columns of the matrix. The distance between two FPGAs is computed as the *Manhattan distance*, which is the length of the semi-perimeter of the smallest rectangle containing the two nodes. Usually, a distance equal to one is associated with each wire. The cases of crossbar and bus topologies are simpler, since the distance between any two FPGAs is in such cases always equal to one, as explained in Subsection 1.2.1.

5.2.2 Integrated partitioning and placement

As reported in Subsection 3.3.1, one approach for addressing partitioning and placement is to handle them in a unique process with an integrated algorithm. Since the number of the variables involved in searching a solution grows by combining the two problems, an iterative approach provides a suitable way to cope with such incremented complexity. A simulated annealing algorithm has indeed been designed and implemented. The simulated annealing technique has been presented Subsection 2.1.1.2.

A solution to the partitioning and placement problem is represented by using an array, whose length is equal to the number of nodes in the considered application. The value of the elements identify which FPGA hosts the node labeled by the correspondent index. Mathematically, a general solution is an array $S_n = [c_i]$ of size n, $c_i \in (0, 1, \ldots, f-1)$, where n is the number of nodes in the application and f is the number of FPGAs in the architecture; $c_i = j$ means that the node of the application having index i is hosted by FPGA j.

There are two kinds of random moves for generating a new solution starting from an existing one. The first one is to *move* a randomly picked node of the application to a randomly chosen FPGA. The second is to randomly *swap* two random nodes between their relative partitions. At each iteration, one of these moves is chosen with equal probability. The first kind of move is useful when the current partitions are not balanced, while the second helps in optimizing balanced

partitions, especially in the case where the move of a single node causes an area constraint to be violated.

The cost function used by the simulated annealing algorithm is the *weighted estimated wire length* (WEWL), that is the sum of the distances of each pair FPGAs in the architecture multiplied by the width (in bit) of the interconnections between them. The weighted estimated wire length is given by the following formula

$$WEWL = \sum_{i,j \in (0,\ldots,n-1): c_i \neq c_j} w(i,j) d(c_i, c_j)$$

where $w(i,j)$ is the width of the interconnections between nodes i and j in the application and $d(c_i, c_j)$ is the distance between FPGAs c_i and c_j in the architecture.

Each time a random move is performed, the cost function of the new solution has to be evaluated. In the following, the complexity of this operation is analyzed. The distance $d(c_i, c_j)$ is provided directly by the Architecture class. In the case of mesh topology, such distance is retrieved in constant time since the architectural graph is internally represented as an adjacency matrix. For bus and crossbar topologies, it is constant as well, because the distance for any pair of FPGAs is always the same, and in particular it is assumed to be equal to one. Therefore, to compute the cost, it is enough to consider all the edges of the application, and add the correspondent width if the two terminals belongs to different partitions. Since the Application class provides a method to retrieve the outgoing edges of a node in linear time using the underlying adjacency list structure, the overall operation requires $O(m)$ time, where m is the number of edges in the application[1]. This computation is necessary only to compute the cost of the initial solution, because successively the cost of a solution can be obtained by updating the previous one. For a single-node move, it is indeed enough to subtract from the current cost the connections weighted considering the node as being in the old partition, and add the connections weighted considering the node as being in the new partition.

[1] It is reasonably assumed that $m > n$

Mathematically, the formula for updating the cost when node i is moved from partition r to partition s is:

$$\begin{aligned} WEWL_{new} &= WEWL_{old} + \sum_{(i,j)\in\delta(i)} w(i,j)d(r,c_j) - \sum_{(i,j)\in\delta(i)} w(i,j)d(s,c_j) \\ &= WEWL_{old} + \sum_{(i,j)\in\delta(i)} w(i,j)[d(r,c_j) - d(s,c_j)] \end{aligned}$$

where $WEWL_{old}$ and $WEWL_{new}$ are the costs before and after the random move is executed, respectively, and $\delta(i)$ is the set of the edges incident to node i in the application graph. Such set can be retrieved in $O(|\delta(i)|)$ time since the Application class uses an adjacency list data structure. Therefore the computation of the new cost takes on average $O(d)$ time, where d is the average degree of the nodes in the application. When a swap move is performed, it is enough to apply two times the formula above for updating the cost.

Besides searching a solution which minimizes a cost function, the algorithm must also satisfy the capacity constraints of the FPGAs in the architecture. There are two constraints that a solution must satisfy in order to to be acceptable:

- *Area constraint*: the sum of the area (expressed in number of slices) of the nodes assigned to each FPGA must be less than the area available on the FPGA. In order to make this constraint more realistic, it is better to consider the actual available area as a fraction of the total available area on a FPGA, because it is natural that even a powerful floorplanning[2] algorithm leads some space to be wasted, because of the shape of modules and the configuration techniques. Therefore, we impose that the sum of the area of the nodes belonging to a partition must be less than the 80% of the total available area of the FPGA. Mathematically, this constraint is expressed as:

$$\sum_{i\in(0,\ldots,n-1):c_i=j} a_i < 0.8 * A \qquad \forall j \in (0,1,\ldots,f-1)$$

[2]In [42], *floorplanning* is defined as a generalization of placement, where the shape and pin position of circuit components have a specified flexibility.

where a_i is the number of slices required for physically implement node i, and A is the number of slices available on a FPGA.

- *Pin constraint*: the sum of the width of the interconnections between the nodes belonging to a partition and the nodes belonging to other partitions must be less or equal than the I/O pins available on the FPGA. Mathematically,

$$\sum_{h\in(0,\ldots,n-1):c_i=j} \sum_{(h,k)\in\delta(h)} w(h,k)) \leq P \quad \forall j \in (0,1,\ldots,f-1)$$

where $\delta(h)$ is the set of the edges incident to node h in the application graph, and $w(h,k)$ is the width of the interconnection between nodes h and k in the application.

In order to fulfill the area and the pin constraints described above, the notion of *penalty* is introduced. The penalty is obtained by summing the quantities of area and pin counts which exceed the limit imposed by the relative constraints. For a solution to be feasible, the penalty must be equal to zero. To compute the penalty, two auxiliary arrays of length f are used: $[o_i]$ contains the sum of the area occupations of the nodes in each FPGA, while $[p_i]$ contains the amount of I/O pins required by the nodes in each FPGA. Therefore, the penalty can be computed as

$$Penalty = \sum_{i=(0,\ldots,f-1)} [(0.8*A-o_i)+(P-p_i)]$$

The initialization of such arrays requires $O(m)$ time, while the initial computation of the penalty requires $O(f)$ time. To update the penalty of a solution after node i has been moved from partition r to partition s, the following formula is applied:

$$\begin{aligned} Penalty_{new} = Penalty_{old} &- \min(0,o_r-0.8*A)+\min(0,o_s-0.8*A) \\ &- \min(0,p_r-P)+\min(0,p_s-P) \end{aligned}$$

where $Penalty_{old}$ and $Penalty_{new}$ are the penalties before and after the random move is executed, respectively. The time complexity of such updating is $O(d)$, obtained by making the same considerations done for the cost updating.

To provide a good solution that has penalty equal to zero (if it exists) an algorithm as been designed which is composed by two independent simulated annealing processes. The first one aims at finding a feasible solution, and stops when the penalty reaches a value of zero. The obtained solution is then optimized by the second annealing process, which aims at minimizing the cost function. In such process, the moves which cause the penalty to become more than zero are immediately rejected without considering the cost variation they provide. The complete algorithm is outlined in the pseudocode showed in Algorithm 2. Details as the *Metropolis* acceptance function are not provided since they has been already explained in Subsection 2.1.1.2.

Based on the complexity of the single operations explained above, the overall complexity of the proposed algorithm is $[O(m)+O(f)+H*O(d)]+[O(m)+K*O(d)]$, where H and K are the number of performed iterations of the first and the second annealing process, respectively. Although H and K are constant, it is reasonable to suppose that an effective annealing process performs a number of random moves (iterations) which is far bigger than the number of the nodes in the application (n). Therefore, the complexity becomes $H*O(d)+K*O(d) = (H+K)O(d)$.

5.2.3 Sequential partitioning and placement

In the previous subsection an integrated partitioning and placement approach has been described. In the present subsection, on the other hand, the implementation of an approach in which partitioning and placement are handled separately and sequentially is explained. In this methodology, first a partitioning of the input application is created, such that each partition fits onto a single FPGA. Once such

Algorithm 2 Pseudocode of the integrated partitioning and placement algorithm.

```
Generate an initial random solution
T = T_0
Penalty = ComputePenalty()
while t_stop > 0 do
    AcceptMove = FALSE
    for i = 1 to M do
        Perform a random move (either single-node or swap)
        NewPenalty = UpdatePenalty()
        if AcceptMove(ΔPenalty, T) then
            Penalty = NewPenalty
            AcceptMove = TRUE
            if NewPenalty = 0 then
                Exit the annealing process
            end if
        else
            Move i to its original partition
        end if
    end for
    if AcceptMove then
        t_stop = t_s
    else
        t_stop = t_s − 1
    end if
    T = T ∗ α
end while

Cost = ComputeCost()
while t_stop > 0 do
    AcceptMove = FALSE
    for i = 1 to M do
        Perform a random move (either single-node or swap)
        NewPenalty = UpdatePenalty()
        NewCost = UpdateCost()
        if ΔPenalty ≤ 0 then
            if AcceptMove(ΔCost, T) then
                Cost = NewCost
                AcceptMove = TRUE
            end if
        else
            Move i to its original partition
        end if
    end for
    if AcceptMove then
        t_stop = t_s
    else
        t_stop = t_s − 1
    end if
    T = T ∗ α
end while
```

partitions have been obtained, they are placed on the actual FPGAs. The advantages of such approach have already been discussed in Subsection 3.3.1. Subsection 5.2.3.1 describes the implementation of the *bottom up clustering* partitioning algorithm, while Subsection 5.2.3.2 explains how the *1-to-1 placement* algorithm works.

5.2.3.1 Bottom-up clustering

This algorithm implements a bottom-up clustering approach. First, each leaf of the hierarchy is considered as a cluster. Then, the two clusters whose union maximizes a given closeness metric are collapsed together to form a new cluster. This operation can be performed only if the resulting new cluster satisfies the area and pin count constraints of the used FPGAs. This process is repeated until no more clusters can be formed, due to the constraints, or only one cluster is left.

A solution to this problem is represented by a set of clusters $C = \{c_1, c_2, \ldots, c_k\}$; each cluster c_i is a set of nodes of the application. The metrics that the algorithm considers are:

- *Connection* (*Conn*), which is the volume of communication between two clusters. Mathematically, the Connection metric of two clusters c_p and c_q is obtained as:

$$Conn(c_p, c_q) = \sum_{i \in c_p, j \in c_q} w(i, j)$$

  where $w(i, j)$ is the amount of communication, in number of bits, between node i and node j.

- *Communication Ratio* (*CR*), which is the ratio between the Internal Communication (*IC*) and the External Communication (*EC*) of the cluster. *IC* is the amount of communication between the nodes inside the cluster. Mathematically, the Internal Communication of the union of two clusters c_p and

c_q is obtained as:

$$IC(c_p, c_q) = \sum_{i \in c_p \cup c_q} \sum_{(i,j) \in \delta(i): j \in c_p \cup c_q} Conn(i, j)$$

where $\delta(i)$ is the set of the edges incident to node i in the application. EC is the sum of the communications that the nodes belonging to a given cluster have with external nodes. The following formula returns the EC value of the union of two clusters c_p and c_q:

$$EC(c_p, c_q) = \sum_{i \in c_p \cup c_q} \sum_{(i,j) \in \delta(i): j \notin c_p \cup c_q} Conn(i, j)$$

Therefore, the Communication Ratio is:

$$CR(c_p, c_q) = \frac{IC(c_p, c_q)}{EC(c_p, c_q)}$$

- Communication Density (CD), which is the ratio between the Internal Communication and the number of wires of a clique of size equal to the number of nodes in the cluster. For the union of two clusters c_p and c_q, this number is obtained by:

$$CliqueSize(c_p, c_q) = \frac{n_{c_p \cup c_q}(n_{c_p \cup c_q} - 1)}{2}$$

where $n_{c_p \cup c_q}$is the number of nodes contained in the union of clusters c_p and c_q. Therefore, the formula for computing the Communication Density of the union of two clusters c_p and c_q is:

$$CD(c_p, c_q) = \frac{IC(c_p, c_q)}{CliqueSize(c_p, c_q)}$$

- Common Parent (CP). For a pair of clusters, this metric is equal to one if the two clusters have the same parent in the design hierarchy, and equal to zero otherwise. By indicating the parent of a cluster c_i with the notation

$p(c_i)$, the value of the Common Parent metric for clusters c_p and c_q is given by the formula:

$$CP(c_p, c_q) = \begin{cases} 1 & \text{if } p(c_p) = p(c_q) \\ 0 & \text{otherwise} \end{cases}$$

These metrics can be combined in several ways, by normalizing them and performing a weighted sum. The results for different combinations of the above metrics are showed in Chapter 6. To compute these metrics, the algorithm uses some auxiliary data structures, such that the computation of each metric can be done efficiently. Such data structures are:

- A symmetric connectivity matrix *CONN* of integers. A generic element $conn_{ij}$ of such matrix represents the amount of communication between nodes in cluster c_i and nodes in cluster c_j, expressed in number of bits.
- An internal communication vector *INTCOMM* of integers. A generic element $intcomm_i$ of such vector represents the sum of the communication between pairs of nodes contained in cluster c_i.
- An external communication vector *EXTCOMM* of integers. A generic element $extcomm_i$ of such vector represents the amount of communication between nodes contained in cluster c_i and external nodes.
- A parent vector *PAR* of integers. Two elements of this vectors are equal if the corresponding clusters' parents are instances of the same component.

It is simple to see how these data structures allow a constant time computation of the metrics expressed above. This is shown by the following formulae:

$$Conn(c_p, c_q) = conn_{pq}$$

$$IC(c_p, c_q) = intcomm_p + intcomm_q + conn_{pq}$$

$$EC(c_p, c_q) = extcomm_p + extcomm_q - conn_{pq}$$

$$CP(c_p, c_q) = \begin{cases} 1 & \text{if } par_p = par_q \\ 0 & \text{otherwise} \end{cases}$$

The critical point is the computation of such data structures. At the beginning of the execution of the algorithm, the *CONN* matrix can be retrieved from the Application class in $O(n^2)$ time. The elements of the *INTCOMM* vector are initially all equal to zero, therefore the vector can be initialized in constant time. The *EXTCOMM* vector initially contains the sums of the width of the edges incident to each node. Its initialization requires $O(n)$ time, because such values are stored in the nodes of the Application class. Similarly, the *PAR* vector is initialized in linear time using the information provided by the Hierarchy class.

As said before, the algorithm works by recursively collapsing clusters whose union has the maximum value of the chosen matrix. Finding this pair requires $O(n^2)$ time. When these two clusters are identified, the procedure is to delete all the nodes from the cluster with smaller cardinality and add them to the other cluster. When this happens, the above data structures must be updated. The row and column of *CONN* corresponding to the bigger cluster are updated by adding the values taken from the row and column of *CONN* corresponding to the smaller cluster, element by element. Then, the elements of the row and column corresponding to the smaller cluster are all set to the value -1, which means that such cluster does actually no longer exist. This updating takes $O(n)$ time. To update the other two vectors, it is enough to apply the formulae for the computation of *IC* and *EC* to the element of the vectors corresponding to the bigger cluster, and set the element corresponding to the smaller cluster to -1. This is done in constant time. The *PAR* vector is updated in the following way: the element corresponding to the smaller cluster is set to -1, while the one corresponding to the bigger cluster is left unchanged if the *CP* metric for the two clusters is equal to one, and is set to -1 otherwise. Moreover, if a cluster contains all and only the nodes which compose the structural of their common parent, the corresponding element of *PAR* is set to

the value which identifies their grandparent. This check can be performed in $O(b)$ time using the Hierarchy class, where b is the *branching factor*, that is the average number of children of the non-leaf nodes of the hierarchy.

As said before, the area and I/O pins constraints must be satisfied each time a new cluster is formed by collapsing two smaller clusters. These constraints are expressed in a very similar way to the ones described for the annealing algorithm in Subsection 5.2.2, therefore their formulae are here omitted. Two arrays, one containing the current dimension and one containing the I/O pin requirements of each cluster, are used for checking the constraint. Their initial computation requires linear time, while their updating is done in constant time.

The pseudocode of the resulting algorithm, called CLUSTERING, is showed in Algorithm 3.

Algorithm 3 Pseudocode of the bottom-up clustering algorithm (CLUSTERING).

```
Initialize the data structures
while A new cluster is formed or there is only one cluster do
    Select the two clusters with max metric value
      which do not violate area and pin count requirements
    Collapse them in a new cluster
    Update the data structures
end while
```

From what has been said regarding the complexity of the various steps of the algorithm, it comes out that the most time consuming operation *in each* iteration of the algorithm is the selection of the two clusters to be collapsed, which requires $O(n^2)$ time. What has to be evaluated is the number of iterations of the algorithm. The worst case execution time is when the algorithm ends producing a single cluster. In that case, the algorithm performs n iterations. Therefore, the overall complexity of the algorithm is $O(n^3)$.

Besides providing information to compute *CP* metric, as explained above, the design hierarchy can be exploited in another way. As a matter of fact, the design hierarchy implies *regularities* (Subsection 3.3.1.1) in the design. These regular-

ities can be exploited in the clustering process, as already described in Subsection 4.2.2. The enhanced version of the algorithm, which considers the regularities of the design in the clustering process, is the basis of the ISOMORPHIC-CLUSTERING algorithm. In order not to be redundant in the explanation, such algorithm will be presented in Subsection 5.3.1.

5.2.3.2 1-to-1 placement

Once the partitions of the circuit have been provided, they have to be placed on the target architecture. Each partition is assigned to exactly one FPGA, and each FPGA receives exactly one partition. This rather simple placement problem can be addressed by a simulated annealing algorithm which is less complex than the one described for the integrated partitioning and placement approach in Subsection 5.2.2. As a matter of fact, after a random assignment is computed, only swap moves are executed and no constraint checks have to be carried out, since the provided partitions already fulfill all the multi-FPGA architectural requirements. The cost function to be minimized is the weighted estimated wire length, which is computed identically to the annealing algorithm cited above. For these reasons, the implementation of the *1-to-1 placement* algorithm is not explicitly provided, as it represent a straightforward generalization of the integrated partitioning and placement simulated annealing.

5.3 Blocks reuse algorithms

In this section the implementation of the algorithms for the reuse of blocks in multi-FPGA systems is described. Subsection 5.3.1 describes how the algorithm for finding isomorphic clusters in the design is implemented. The description of the algorithm to extract the horizontal cuts of the resulting dendrogram, is provided in Subsection 5.3.2. Then, Subsection 5.3.3 describes the ILP model for

choosing which blocks to reuse in the execution of the application in order to minimize the interconnections' reconfiguration time.

5.3.1 Finding isomorphic clusters

The algorithm for finding isomorphic clusters of in the application graph, called ISOMORPHIC-CLUSTERS, has been methodologically described in Subsection 4.2.2. In this subsection, its implementation is explained. The algorithm exploits the regularities of the design in order to extract isomorphic structures. Such regularities are extracted in a bottom-up clustering process. Therefore, the bulk of the algorithm is the clustering procedure which has been described in Subsection 5.2.3.1. The most remarkable addition is that in this case the hierarchy is exploited intensively, not only as a look-up table for finding nodes' parents. As a matter of fact, the Hierarchy provides some methods which implements the transformation described in Subsection 4.2.2 and exemplified in Figure 4.4. Essentially, the design hierarchy "evolves" during the execution of the clustering algorithm in order to provide it updated information. To make this feasible, a direct mapping between the clusters considered by the algorithm and the nodes of the mutable hierarchy must be identified and maintained. Such mapping is implemented with a C++ map, whose pairs are represented by the index of the clusters in the algorithm data structures and an identifier of the corresponding node in the hierarchy class.

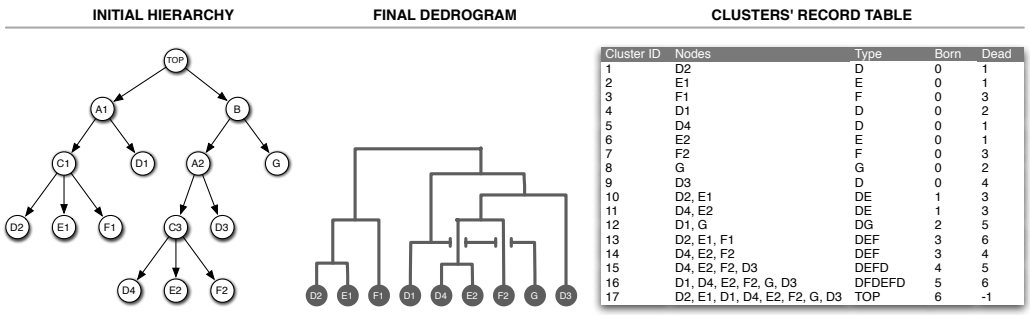
The output of the ISOMORPHIC-CLUSTERS algorithm is a set of records, one for each cluster that has been formed and possibly later merged into a bigger one during the execution. The information carried by each record is:

- the identifier of the cluster,
- the nodes of the application contained into it,
- the overall dimension of the cluster, in number of occupied slices,
- the amount of external communication, in number of bits,

- the type of the cluster,
- the iteration of the clustering algorithm in which the cluster is created, called *born*, and
- the iteration of the clustering algorithm in which the cluster is merged to another to form a new one, called *dead*.

The latter two fields determine what is called the "lifetime" of a cluster, which is the interval of iteration in which the cluster is "alive", from when it is created by collapsing two smaller clusters to the iteration in which it is merged with another cluster to form a bigger one. If this never happens, the value of the last field is conventionally set to -1.

The procedure is very similar to the "standard" clustering algorithm described in section 5.2.3.1. The difference is that, each time a pair of clusters is selected to be collapsed on the basis of the chosen metric, a method of the Hierarchy class is called. Such method receives as input the identifiers of the two selected nodes, and, if they have the same parent, looks for instances of the same type of the parent throughout the hierarchy. For each of such instances, it extracts the pairs of nodes which constitutes isomorphic structures with respect to the one identified by the two initial nodes. This procedure requires $O(n)$ time, where n is the number of the initial clusters, that is equal to the number of leaf nodes in the hierarchy. The set of all isomorphic pairs is then returned. At this point, all the returned pairs are collapsed in the same iteration by the clustering algorithm. For each collapse operation, besides updating the metric arrays as described in Subsection 5.2.3.1, the algorithm performs two operations. First, it calls a method of the Hierarchy class which performs the collapsing on the hierarchy, in order to keep it updated with the execution of the clustering algorithm. This method operates in $O(n)$ time, needed for erasing the collapsed node from the hierarchy. Second, it sets the *dead* field of the records corresponding to the merged clusters to the number of the current iteration, and create a new record corresponding to the newly created cluster.



Cluster ID	Nodes	Type	Born	Dead
1	D2	D	0	1
2	E1	E	0	1
3	F1	F	0	3
4	D1	D	0	2
5	D4	D	0	1
6	E2	E	0	1
7	F2	F	0	3
8	G	G	0	2
9	D3	D	0	4
10	D2, E1	DE	1	3
11	D4, E2	DE	1	3
12	D1, G	DG	2	5
13	D2, E1, F1	DEF	3	6
14	D4, E2, F2	DEF	3	4
15	D4, E2, F2, D3	DEFD	4	5
16	D1, D4, E2, F2, G, D3	DFDEFD	5	6
17	D2, E1, D1, D4, E2, F2, G, D3	TOP	6	-1

Figure 5.7: Example of clusters' record table produced by ISOMORPHIC-CLUSTERS algorithm.

This is executed in constant time. An example of the generated set of records is shown in Figure 5.7, that refers to the methodological example of Figure 4.4. In the example, the information about the dimension and the external communication of clusters is not provided.

Algorithm 4 Pseudocode of the algorithm for finding isomorphic clusters (ISOMORPHIC-CLUSTERS).

```
Initialize the data structures and the clusters' record table
while A new cluster is formed or there is only one cluster do
    Select the two clusters c_p and c_q with max metric value
      which do not violate area and pin count requirements
    Retrieve the isomorphic pairs of clusters from the hierarchy
    for Each isomorphic pair do
        Collapse the two clusters to form a new one
        Update the entries of the clusters' record table corresponding to the collapsed clusters
        Create a new entry in the clusters' record table
        Update the metrics data structures
    end for
end while
```

The pseudocode of this algorithm is provided in Algorithm 4. To complete the analysis of the complexity of the algorithm, it is necessary to consider the number of isomorphic pairs that are retrieved in each iteration. At a first look, such number seems to be of order $O(n)$, since it is possible that in the first iteration $\frac{n}{2}$ isomorphic

pairs are provided. Nevertheless, a more careful analysis leads to a different result. As hinted before, the worst case for the complexity of the first iteration is when $\frac{n}{2}$ pairs of clusters are retrieved, which means that every node of the input application participates in such recurrent pattern. At the second iteration, under the same worst case consideration, the number of retrieved isomorphic pairs is $\frac{n}{4}$. At the third iteration, it is $\frac{n}{8}$, and so on. Therefore, the *total* number of isomorphic pairs extracted throughout the execution of the algorithm is in the worst case equal to

$$\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\ldots+\frac{n}{n}=\sum_{i=1}^{log_2 n}\frac{n}{2^i}$$

Then, the following inequality holds:

$$\sum_{i=1}^{log_2 n}\frac{n}{2^i}=n\sum_{i=1}^{log_2 n}\frac{1}{2^i}<n\sum_{i=1}^{\infty}\frac{1}{2^i}=n$$

that proves that the *total* number of isomorphic pairs for all the iterations of the algorithm is of order $O(n)$.

From what has been said, the features added with respect to the basic clustering algorithm discussed in Subsection 5.2.3.1 does not increase the its asymptotic complexity, which is still equal to $O(n^3)$.

5.3.2 Extracting the horizontal cuts

As explained in Subsection 4.2.3, the horizontal cuts of the dendrogram resulting from the application of ISOMORPHIC-CLUSTERS have to be determined, in order to extract the flat structures the following phase of the workflow works on. This is done by using the clusters' record table provided by the aforementioned algorithm.

For generating each cut, the records of the table are scanned sequentially, in order to extract an auxiliary table in which each record corresponds to a type of clusters, and is associated to a list of clusters of that type which are "alive" at the considered iteration. Then, this auxiliary structure is used to generate the data file

that constitutes the input of the ILP model which implements the subsequent step of the flow.

For each cut, the sequential scan of the clusters' record table requires $O(n)$ time, since the number of records is at most equal to $2 * n$. As a consequence, also the number of cluster types is at most equal to $2 * n$, therefore the loop for generating the output data file executes $O(n)$ iterations. The number of cuts that have to be provided by the algorithm is equal to the number of iterations performed by the ISOMORPHIC-CLUSTERS algorithm, which has a maximum value of n. Therefore, the overall complexity of the cut extraction algorithm is $O(n^2)$.

5.3.3 ILP model for blocks reuse

In this subsection the integer linear programming (ILP) model used to take decisions about blocks reuse is presented. The problem to be solved has already been described in Subsection 4.1.3. The input to the ILP model is represented by a set of types of clusters of size L. Each cluster is provided with the following information:

- the occupied area on the FPGA, expressed in number of slices,
- the amount of external communication (pin count), expressed in number of bits,
- the number of occurrences of the cluster in the considered circuit.

Moreover, one additional parameter has to be provided, that is the number of slices available on each FPGA.

The ILP model can be simply derived from the description of the problem given in Subsection 4.1.3, and is showed in Table 5.1. The parameters pin_i, $area_i$, and num_i represent, respectively, the I/O pin requirement, the occupied area, and the number of occurrences of the correspondent cluster. The parameter $AMax$ is

Set:	$N = \{1, 2, \ldots, L\}$	Set of isomorphic clusters
Parameters:	$pin_i, 1 \leq i \leq L$	I/O pins used by cluster i
	$area_i, 1 \leq i \leq L$	Slices occupied by cluster i
	$num_i, 1 \leq i \leq L$	Number of occurrences of cluster i
	$AMax$	Available number of slices
Variables:	$x_i \geq 0,\ 1 \leq i \leq L$	
Obj. function:	$min \sum_{i \in N} pin_i * x_i$	Estimated reconfiguration time
Subject to:	$\sum_{i \in N} area_i * (num_i - x_i) \leq AMax$	Area constraint
	$x_i \leq num_i - 1$	Max reuse constraint

Table 5.1: ILP model for blocks reuse.

the maximum available area on the multi-FPGA architecture. The set of variables is the vector $[x_i]$. Each value x_i, ranging from 0 to $num_i - 1$, represents the number of times the correspondent cluster i is *reused*. The value $x_i = 0$ means that the cluster is never reused, and therefore it has to be instantiated num_i times. The value $x_i = num_i - 1$ means that the cluster is reused for all of its occurrences in the application, and therefore must be instantiated once.

The ILP instances of the problem are solved using the *Gnu Linear Programming Kit* (GLPK) [54]. Such tool support the GNU MathProg language [55], which is a subset of the AMPL language.

Chapter 6

Experimental results

This chapter describes the experimental verification that has been carried out in order to validate the methodology proposed in the present thesis work. The numerical results of such experiments, along with comments and considerations, are provided. All tests have been executed on an Intel Core 2 Duo 2.2 GHz machine. In order to provide a clear exposition, the experimental results are presented in the same order of presentation as the corresponding algorithms in Chapter 5. Section 6.1 introduces the benchmarks circuits used in the remainder of the chapter. Section 6.2 provides the results for the design extraction phase. The performances of the algorithms used in the global layout phase are discussed in Section 6.3. Section 6.4 gives the results of the application of the blocks reuse algorithms. Eventually, Section 6.5 describes a case study that shows how the workflow can be used in a practical scenario.

6.1 Benchmarks description

The multi-FPGA design flow presented in this thesis starts with the extraction of the structures and the hierarchy out of a circuit VHDL description. Therefore, the benchmarks used for the validation of the work are VHDL specifications. Al-

though it is common and good practice to describe large circuits by exploiting the hierarchical construct of VHDL, it is quite hard to find freely available specifications that are suitable to be used for the validation of the proposed methodology. Nevertheless, four benchmark circuits have been identified and are described in the following.

- *Triple-DES encryption+decryption core* (3DES) [56]. This circuit is an extension of Data Encryption Standard (DES) that provides a more secure block ciphers to be used in cryptography applications.
- *Finite Impulse Response filter* (FIR). This circuit is a VHDL implementation of a general Finite Impulse Response digital filter.
- *Noekeon cypher* (NOEK) [57]. This circuit implements the Noekeon cyphering algorithm, which has the peculiarity of allowing a compact hardware implementation and provides high resistance to attacks. The considered VHDL implementation of Noekeon has been produced internally to the DRESD research group [58]

Moreover, in order to evaluate the performances of the proposed algorithms on an heterogeneous and realistic circuit, a design has been created which contains an instance of the FIR filter and one of the Triple-DES core, connected with a communication core. In the following, such circuit will be referred as 3DES-FIR.

6.2 Design extraction

The design extraction framework has been validated on the circuits described in the previous section. The design extraction creates, as explained in Section 5.1, one structural graph for each intermediate node of the hierarchy. The hierarchy information is implemented by storing in each intermediate node an identifier of the graph of the relative substructure. In order to obtain the lowest level cut of the

Table 6.1: Design extraction results.

Circuit	*Extraction Time*	*Flattening Time*	*# Nodes*	*# Leaves*	*Tot. Dimension*
3DES	36 ms	96 ms	67	52	1613 slices
FIR	87 ms	691 ms	231	211	561 slices
NOEK	48 ms	12 ms	29	25	958 slices
3DES-FIR	101 ms	2230 ms	301	264	2141 slices

hierarchy, that is the circuit composed by all the leaves of the hierarchical tree, a *flatten* function, provided by the Structural API, is called. In the results reported in Table 6.1, the actual extraction time and the flattening time are reported separately.

The other three columns of the table report the number of nodes that constitute the hierarchy, the number of leaves of the hierarchy (i.e. the number of nodes of the flattened circuit) and the total area size in number of occupied slices on a Virtex Spartan-3 FPGA. It is possible to see that the actual extraction time is fairly low, while the time needed by the flattening function rapidly increases as the number of nodes of the design grows. Moreover, it can be noticed that the FIR circuit, which has 211 leaf nodes, is three times smaller than the 3DES circuit. This is caused by the fact that 3DES has bigger leaves. This fact will be taken into account for considerations about the algorithm performances in the following sections.

The extraction phase also provides some graphical representation of the circuit, under the format of *dot* files [59], that can be viewed as graphs using the *Graphviz* tool [60]. Figure 6.1 shows the extracted hierarchy tree fr the 3DES benchmark, while Figure 6.2 depicts the flattened 3DES circuit.

6.3 Global layout

In this section the results of the executions of the algorithm for the global layout phase proposed in this thesis are provided. In Subsection 6.3.1 the simulated an-

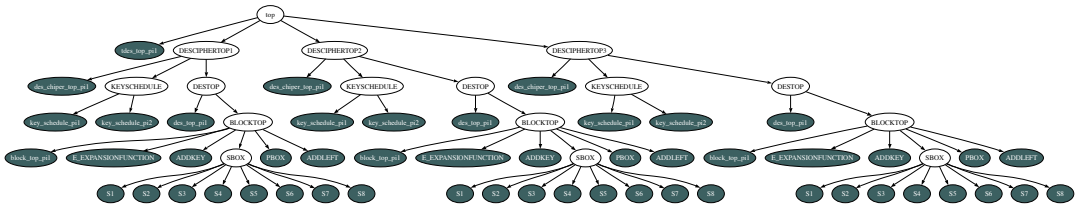


Figure 6.1: Hierarchical tree of the 3DES benchmark circuit.

nealing algorithm to address the partitioning and placement problems in the same time is evaluated. On the other hand, Subsection 6.3.2 provides the results relative to the sequential approach, in which partitioning and placement are faced separately as two sequential steps. Subsection 6.3.3 provides a comparison between the two approaches.

6.3.1 Integrated partitioning and placement

The simulated annealing approach evaluated in this section has been methodologically described and motivated in Subsection 3.3.1, and its implementation has been explained in Subsection 5.2.2. In the following, several architectural solutions are considered.

Recalling what has been said in 5.2.2, the cost function for partitioning and placement is the Weighted Estimated Wire Length (WEWL), that is the sum of the amounts of communication for any pair of application nodes weighted on the relative distance of the FPGAs which host the two nodes. As concerns bus-based or crossbar-style topologies, the integrated approach is essentially reduced to a partitioning process, since the distance of any two pairs of FPGA is considered to be equal to one. For mesh topologies the cost function is instead weighted on the Manhattan distance between the FPGAs. The annealing parameters have been tuned on the basis of observation of its behavior. The starting temperature has been set to 5000, with a cooling rate equal to 0.99. For each temperature value, the algorithm performs 10 random moves, and it stops if there is no change in the

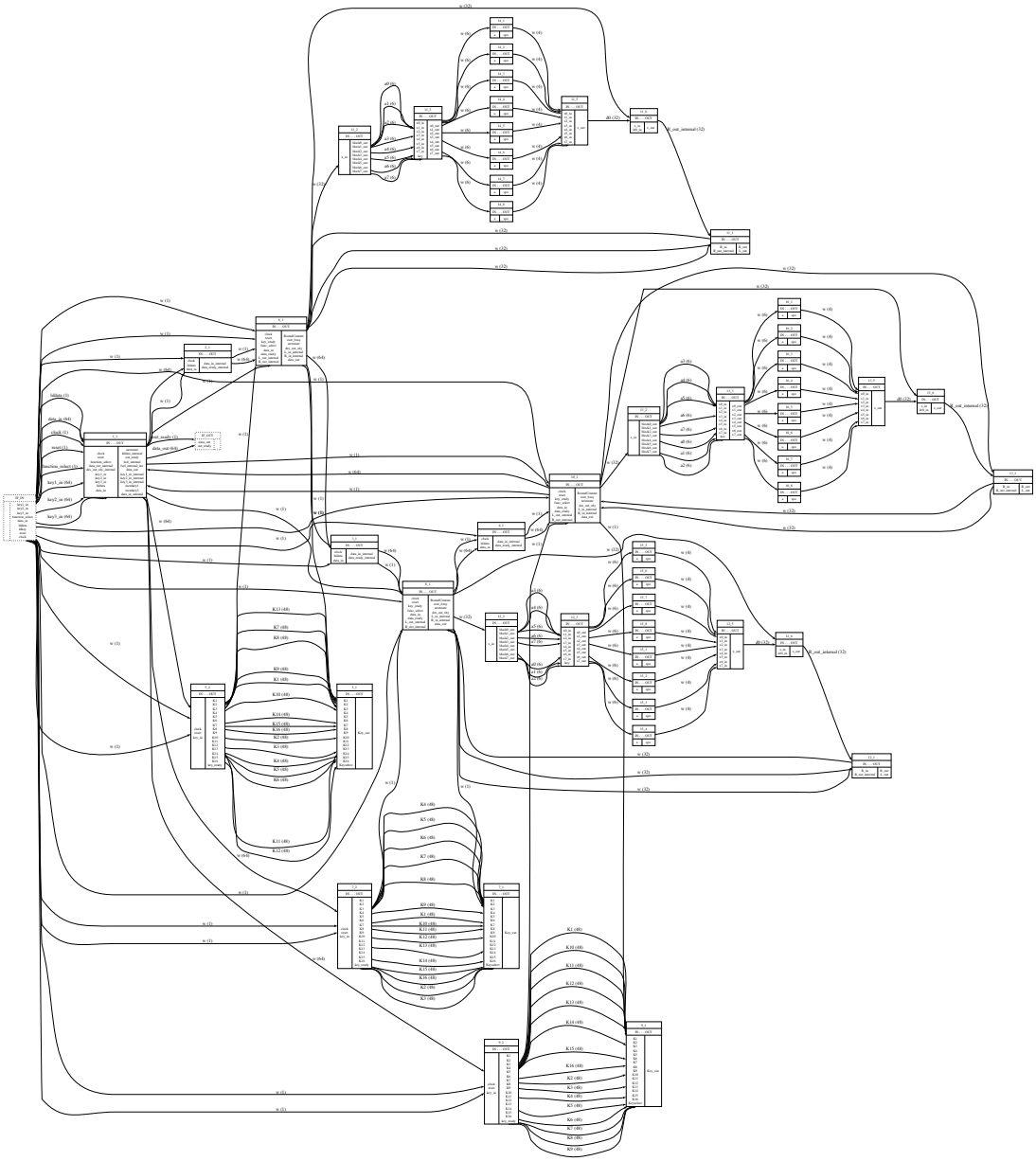


Figure 6.2: Flattened circuit view of the 3DES benchmark circuit.

Table 6.2: Results for the integrated partitioning and placement algorithm on bus/crossbar architectures.

Circuit	**3DES**		**FIR**	
FPGA Dim.	*WEWL*	*Time*	*WEWL*	*Time*
300 slices	773	1945 ms	54	2324 ms
400 slices	616	2029 ms	53	2025 ms
600 slices	468	1666 ms	0	939 ms
Circuit	**NOEK**		**3DES-FIR**	
FPGA Dim.	*WEWL*	*Time*	*WEWL*	*Time*
300 slices	2061	1571 ms	862	5249 ms
400 slices	1880	1461 ms	756	4950 ms
600 slices	1128	1291 ms	619	4277 ms

cost function after 1000 moves.

First, the results for bus/crossbar architectures are provided. In such case, an architecture is completely specified by the number of chips, their dimension, and their number of I/O pins of each FPGA. Fixing the number of I/O pins to 300, three FPGA sizes are considered: 300 slices, 400 slices and 600 slices. The number of chip the architecture is composed of varies according to the dimension of the considered benchmark circuit: more precisely, it is set to the value $\left\lceil \frac{dim_{FPGA}}{dim_{circuit}} \right\rceil$. The results are shown in Table 6.2. Since the simulated annealing is a randomized process, ten runs of the algorithm have been executed for each architectural topology, and the average values are reported in the table. Times are expressed in milliseconds.

As expected, the algorithm has higher execution times as the dimension of the circuit increases. The zero value for the WEWL obtained in partitioning the FIR circuit is explained by the fact that the application fits entirely on a chip having a number of slices greater than the total dimension of the circuit. In order to

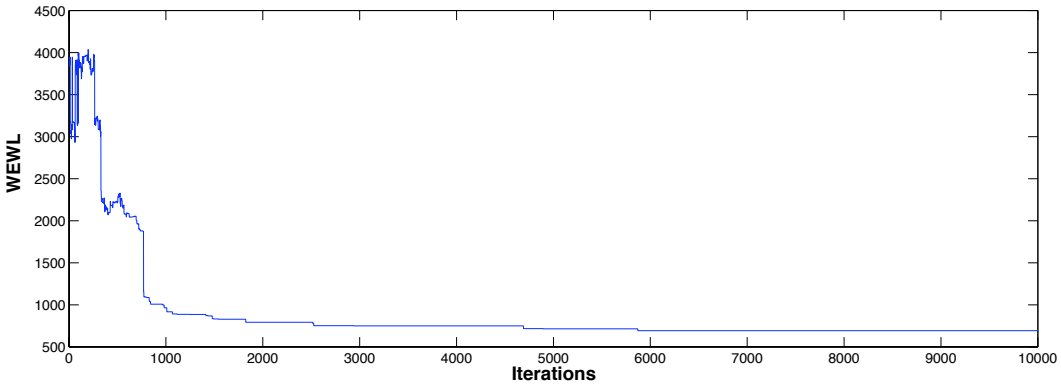


Figure 6.3: Plot of the annealing cost function for the FIR circuit partitioning.

show how the annealing process works, the plot of the cost function is provided in Figure 6.3. The figure refers to the 3DES-FIR circuit and a FPGA size of 600 slices.

For mesh topologies, the number of chips, given the FPGA dimension, has been set in a slightly different way. As a matter of fact, narrow mesh topologies are avoided, by favoring mesh shapes which are close to a square. To give an example, given a FPGA dimension of 100 slices and a circuit of 1050 slices, a more common 3×4 mesh is used rather than a 1×11 one. The results are shown in Table 6.3. To describe a particular mesh architecture, the notation $dim_{n \times m}$ is used, which means that the mesh has n rows and m columns, and there are dim available slices on each FPGA. The *WEWL* column reports the value cost function that results from the placement, while *Time* is the execution time.

From the table, it can be inferred that the execution times are only slightly higher than the bus/crossbar topology case. The results of the two tables provided in this subsection will be compared to the results obtained by the execution of the sequential partitioning and placement algorithms in Subsection 6.3.2.

Table 6.3: Results for the integrated partitioning and placement algorithm on mesh architectures.

3DES			**FIR**		
Arch.	*WEWL*	*Time*	*Arch.*	*WEWL*	*Time*
$300_{2\times3}$	1126	1955 ms	$300_{1\times2}$	96	2972 ms
$400_{2\times2}$	1527	1857 ms	$400_{1\times2}$	62	2294 ms
$600_{1\times3}$	796	1671 ms	$600_{1\times1}$	0	998 ms
NOEK			**3DES-FIR**		
Arch.	*WEWL*	*Time*	*Arch.*	*WEWL*	*Time*
$300_{2\times2}$	2609	1667 ms	$300_{2\times4}$	1565	6068 ms
$400_{1\times3}$	2337	1510 ms	$400_{2\times3}$	1206	4917 ms
$600_{1\times2}$	1130	1348 ms	$600_{2\times2}$	844	4736 ms

6.3.2 Sequential partitioning and placement

In this subsection, the sequential partitioning and placement approach is evaluated.

First, the results for the partitioning algorithm based on clustering are presented. As described in Subsection 5.2.3.1, there are several metrics that can be considered in the clustering process. Although four metrics were described, only three of them are explicitly used, namely *Conn* (Connection), *CR* (Communication Ratio), and *CD* (Cluster Density), while the *CP* (Common Parent) metric is intrinsically used for extracting regularities from the design hierarchy. In the following, comparisons among the metrics are provided, based on tests carried out on the benchmark circuits. The architectural parameters that the algorithm has to consider are the pin count, fixed to 300, and the FPGA dimension. As for the verification of the integrated approach described in the previous Subsection, three different FPGA dimensions are considered, namely 300, 400 and 600 slices.

Moreover, in order to provide a comparison with an existent approach, the results obtained with the *Metis* partitioning framework (presented in Subsection 2.1.1.3) are provided as well.

It is important to notice that the clustering algorithm does not receive as input the number of FPGAs used in the architecture. As a matter of fact, the clustering process implicitly aims at finding a solution which minimizes the number of partitions. On the other hand, the *Metis* partitioner takes as input a parameter which represents the number of desired partitions. In our evaluation, *Metis* is executed by giving as input the number of partitions obtained by the clustering process; this ensures that a fairly comparison is carried out. In the following, four tables provide the results for the different benchmark circuits: Table 6.4 contains the results obtained for the 3DES circuit, Table 6.5 contains the results for the FIR circuit, Table 6.6 contains the results for the NOEK circuit, and Table 6.7 contains the results for the 3DES-FIR circuit. Each table provides the results obtained by means of the three considered metric. The *FPGA dim.* field is the size of the FPGA in the considered architecture, *Cutsize* is the amount of communication, in bits, between the different partitions, *No. part.* is the number of the obtained partitions, *No. it* is the number of iterations performed by the clustering process, and *Time* is the execution time in milliseconds. The last column report the cutsize obtained by running the *Metis* partitioner. As concerns Metis, the running time is not provided as it is supposed to be always less than 100 milliseconds. As a matter of fact, the tool returns the running time in seconds chopped to the first decimal value.

In the following, an analysis of the results provided in the tables is carried out in order evaluate the three considered metrics. The first metric, *Conn*, produces better cutsizes with respect to the other two. Nevertheless, the number of obtained partitions is in most cases slightly higher. The second metric, *CR*, creates partitions having a better area exploitation, being the number of partitions always smaller or equal than the first metric. Moreover, it also has lower execution times for all the considered instances. On the other hand, the value of the obtained cut-

Table 6.4: Results of the application of the clustering algorithm for the *3DES* benchmark circuit.

Metric: **Conn**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No. it.*	*Time*	*Cutsize*
300	547	7	42	18 ms	712
400	550	5	34	17 ms	1536
600	349	3	46	19 ms	1335
Metric: **CR**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No it.*	*Time*	*Cutsize*
300	1604	6	45	13 ms	577
400	1296	5	46	13 ms	1536
600	1193	3	48	13 ms	1335
Metric: **CD**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No. it.*	*Time*	*Cutsize*
300	915	6	44	13 ms	577
400	692	5	44	13 ms	1536
600	417	3	46	14 ms	1335

Table 6.5: Results of the application of the clustering algorithm for the *FIR* benchmark circuit.

Metric: **Conn**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No. it.*	*Time*	*Cutsize*
300	36	2	186	970 ms	27
400	50	2	186	952 ms	27
600	0	1	187	974 ms	0
Metric: **CR**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No it.*	*Time*	*Cutsize*
300	130	2	190	639 ms	27
400	197	2	210	663 ms	27
600	0	1	210	660 ms	0
Metric: **CD**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No. it.*	*Time*	*Cutsize*
300	111	3	101	337 ms	58
400	53	2	102	334 ms	27
600	0	1	103	342 ms	0

Table 6.6: Results of the application of the clustering algorithm for the *NOEK* benchmark circuit.

Metric: **Conn**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No. it.*	*Time*	*Cutsize*
300	1965	4	22	3 ms	2128
400	2061	3	23	3 ms	1518
600	1314	2	24	3 ms	1421
Metric: **CR**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No it.*	*Time*	*Cutsize*
300	2672	4	22	2 ms	2128
400	2478	3	23	2 ms	1518
600	1643	2	24	2 ms	1421
Metric: **CD**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No. it.*	*Time*	*Cutsize*
300	1965	4	22	2 ms	2128
400	1995	3	23	2 ms	1518
600	1506	2	24	2 ms	1421

Table 6.7: Results of the application of the clustering algorithm for the *3DES+FIR* benchmark circuit.

Metric: **Conn**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No. it.*	*Time*	*Cutsize*
300	610	9	228	2034 ms	1827
400	620	6	229	2019 ms	2826
600	434	4	230	1990 ms	1683
Metric: **CR**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No it.*	*Time*	*Cutsize*
300	1371	8	219	1165 ms	2148
400	1482	6	256	1255 ms	2826
600	1344	4	257	1261 ms	1683
Metric: **CD**					**Metis**
FPGA dim.	*Cutsize*	*No. part.*	*No. it.*	*Time*	*Cutsize*
300	1348	8	200	1187 ms	2148
400	1001	6	187	1082 ms	2826
600	976	4	210	1167 ms	1683

sizes range in most cases between 2 and 3 times the ones obtained by adopting the first metric. The third metric, *CD*, seems to represent a middle ground between the first and the second ones. As a matter of fact, the cutsizes are always lower than the ones obtained with the second metric and higher than the ones obtained with the first metric. Besides that, it generates in all cases the same amount of partitions as the *CR* metric. Moreover, the running times are lower than the other two metrics. This last result comes from the fact that this metric tends to cause a lower number of iterations of the clustering process with respect to the other two. To conclude this comparison, it is possible to say that the *Conn* metric is a good choice in case the critical objective is the minimization of the cutsize. In the case the stress is one the number of used FPGAs, the *CD* metric seems to be the most favourable choice. The *CR* metric, instead, does not emerges as a good choice in neither on of this two scenarios.

It is interesting to compare the obtained results with the ones obtained by the *Metis* partitioner. The cutsizes produced by *Metis* are in most cases much higher than the one obtained using the proposed clustering algorithm with the *Conn* metric. The only instances in which *Metis* behaves better are the partitionings of the FIR circuit. By observing the ratio between the number of nodes and the total area of the circuits provided in Table 6.1, it can be noticed that *Metis* works well when the dimension of the nodes is on average small with respect to the area available on the FPGAs. Since the proposed methodology addresses circuit structures having process-level granularity, choice due to the several advantages explained in Subsection 3.3.2, a clustering approach seems to represent a better choice than an algorithm intended for working on classic low-level netlists, such as *Metis*.

Once the partitioning is carried out, the obtained clusters are subject to a 1-to-1 placement one the actual FPGAs. This process, as said in Subsection 5.2.3.2, is carried out with an annealing algorithm which is simpler than the integrated approach presented in the previous section, due to the facts that the solution space is narrower and the architectural constraints are already fulfilled by the partitioning

step. In the following, the results of such 1-to-1 placement on mesh architecture are provided. Since the result of partitioning represents by itself the output placement in case a bus partial crossbar topology is adopted (remember than the distance between two FPGAs in such architectures is always equal to one), the algorithm is not executed in case such topologies are chosen.

In order to provide comparable data with the ones obtained with the integrated approach in the previous Subsection, the considered mesh structures are the same. Table 6.8 shows the obtained results. The best possible metric is used in each case, given that the produced number of partitions is compatible with the used architecture. The *Arch.* column provides a specification of the target architectural topology, *WEWL* is the cost function, *Time* is the running time of the 1-to-1 placement considered singularly, and *Tot. time* is the whole execution time of the clustering and placement steps. One of the architectures is marked with a '*', meaning that none of the metrics produces the desired number of partitions; therefore, a bigger architecture is chosen.

From the table, it is possible to see that running times have a high variance. This is due to the fact that in some cases the solution to the 1-to-1 placement problem is trivial, like the case in which there are only two FPGAs in the mesh.

In the next Subsection comparisons between the integrated approach and the sequential one are provided.

6.3.3 Comparisons between integrated and sequential approach

First, the results relative to bus and crossbar architectures are provided. These two topologies do not need a placement phase that minimizes the estimated wire length. The results that are considered for comparison can be found in Table 6.2 and the set of four tables (Tables 6.4, 6.5, 6.6, 6.7) for the integrated and the sequential approach, respectively.

One difference in the two approaches that must be pointed out is that the

Table 6.8: Results for the sequential partitioning and placement approach on mesh architectures.

3DES				**FIR**			
Arch.	*WEWL*	*Time*	*Tot. time*	*Arch.*	*WEWL*	*Time*	*Tot. time*
$300_{2\times3}$	1224	1335 ms	1348 ms	$300_{1\times2}$	36	2 ms	946 ms
$400_{2\times3}$*	621	770 ms	789 ms	$400_{1\times2}$	50	2 ms	946 ms
$600_{1\times3}$	494	527 ms	540 ms	$600_{1\times1}$	0	1 ms	947 ms

NOEK				**3DES-FIR**			
Arch.	*WEWL*	*Time*	*Tot. time*	*Arch.*	*WEWL*	*Time*	*Tot. time*
$300_{2\times2}$	2287	655 ms	3 ms	$300_{2\times4}$	2099	1658 ms	2837 ms
$400_{1\times3}$	2061	528 ms	531 ms	$400_{2\times3}$	775	892 ms	2870 ms
$600_{1\times2}$	1314	2 ms	5 ms	$600_{2\times2}$	545	753 ms	2743 ms

annealing-based integrated approach is imposed to use the lowest number of FPGAs as possible, while the clustering algorithm for partitioning does not work with a fixed number of partitions. This can constitute an advantage for the integrated method when the design requirements impose the implementation of the circuit on a specific architecture whose total available area is close to the dimension of the circuit.

As concerns cutsize values, clustering behaves slightly better than the annealing algorithm. Nevertheless, the best cutsizes obtained with clustering usually implies the use of one FPGA more than the solutions obtained with annealing. A comparison obtained by considering in clustering the results which use the same number number of partitions as in the annealing solutions shoes that the cutsize values are mostly comparable. On the other hand, the running times of the integrated annealing process are higher than the clustering ones, even for relatively small problem instances.

The results for mesh architectures, contained in Table 6.3 and Table 6.8 for

the integrated and the sequential approach, respectively, are compared in the following. Again, there is not a neat predominance of one approach over the other as regards the cost function, even if the sequential method seems in general to perform slightly better. The running times resulting by the execution of the annealing process are higher than the ones obtained with the sequential approach.

From what has been said in this Subsection, it emerges that the two methods provide overall comparable values of the objective function, with the sequential approach showing lower execution times. Moreover, the clustering performed in the sequential method is essentially the same algorithm which has to be carried out for finding isomorphic clusters if a static feasible solution to global layout cannot be found. Therefore, it is possible to conclude that the sequential partitioning and placement approach represents a more coherent and effective choice in the particular design flow proposed in this thesis.

6.4 Blocks reuse

In this Subsection the blocks reuse algorithms are evaluated. As concerns the ISOMORPHIC-CLUSTERS algorithm, which is in charge of finding isomorphic structures for implementing reuse in the following step, its functioning is identical to the clustering partitioning algorithm whose results have been provided in the previous Subsection. The only difference is that ISOMORPHIC-CLUSTERS outputs a data structures which contains information about the clusters to be used by subsequent algorithms, as explained in Subsection 5.3.1. This fact does not directly affects the performance of the algorithm, therefore the results are not explicitly reported, in order not to cause useless repetitions.

In the following, the time needed for solving ILP model described in Subsection 5.3.3 is considered. The ILP model is solved for the data set extracted from the dendrogram resulting from the application of ISOMORPHIC-CLUSTERS. It is important to recall that every horizontal cut of the dendrogram generates a data

Table 6.9: Time results for solving the ILP models for blocks reuse.

	3DES		**FIR**		**NOEK**		**3DES-FIR**	
	Depth	*Time*	*Depth*	*Time*	*Depth*	*Time*	*Depth*	*Time*
300	42	420 ms	186	2103 ms	22	2173 ms	228	2763 ms
400	34	492 ms	186	2142 ms	23	2325 ms	229	2685 ms
500	46	474 ms	187	2078 ms	22	2296 ms	230	2753 ms

set that is given as input to the ILP model. Table 6.9 reports the overall execution times resulting from executing the ILP solver on all such data sets. In order not to provide an intractable amount of results, the set of problem instances is restricted: considering only the *Conn* metric in the clustering executed by the ISOMORPHIC-CLUSTERS algorithm. In the table, for each benchmark circuit, the column *Depth* is the depth of the dendrogram resulting from the ISOMORPHIC-CLUSTERS algorithm, which is the number of data sets the ILP model has to be solved on, and *Time* is the overall execution time for solving such ILP instances.

The results reported in the table show that there is not a direct correlation between the execution times and the number of instances that have to be solved by the ILP model. As a matter of fact, the ILP models solving process takes more time in the case of NOEK with respect to the FIR circuit, though the last one produces dendrograms that are almost ten times deeper. Generally, it can be seen that the running times are acceptable, even for fairly big circuits as the 3DES-FIR benchmark.

The proposed ILP model minimizes the estimated interconnection reconfiguration time for implementing blocks reuse, given a maximum usable area. Among the horizontal cuts of the dendrograms, the one which provides the least estimated reconfiguration time is then chosen. In order to show the result relative to such estimation, the ILP result for every cut is plotted. Figure 6.4 represents the plot of the objective function for the 3DES-FIR circuit, considering a FPGA dimension

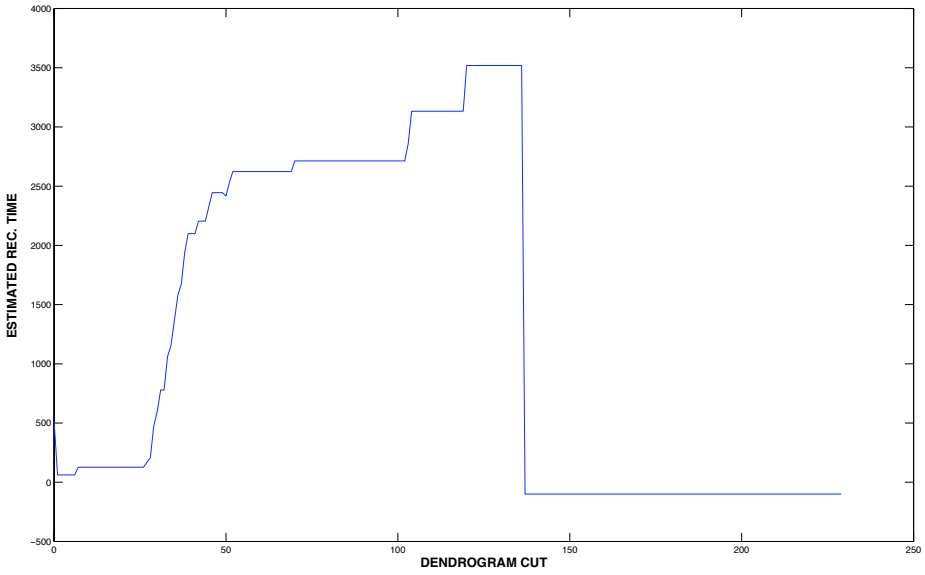


Figure 6.4: Estimated reconfiguration time varying the dendrogram cut.

of 600 slices and a maximum available area of 1600 slices.

For cuts of the dendrogram higher than iteration 139 the value of -100 means that there is no solution to the ILP model. In other words, even reusing all the recurrent patterns of blocks the maximum amount of time, the area constraint cannot be met. The plot shows that after an initially slightly decreasing, the estimated reconfiguration time grows by cutting the dendrogram at higher levels. From the observation of other instances, this comes out to be a general trend. In this particular case, the best dynamically-interconnected structure - with respect to reconfiguration time - for implementing the circuit on an architecture that does not statically hosts it is obtained by considering one of the horizontal cuts of the dendrogram between level 2 and 6.

6.5 Case study: JPEG decoder

In this section a case study is discussed, in order to provide evidence of how the design workflow presented in this thesis could actually be used in practice. The situation where the case study takes place is simple: we want to deploy a JPEG decoder on a multi-FPGA system. In order to achieve higher performances, the core is composed by two decoder modules that operate in parallel. First, we will find a good partitioning of the application over the multiple FPGAs, so that the inter-chip communication volume is minimized. Later, we will assume that we have another multi-FPGA architecture that does not statically contain the whole module. Therefore, we will look for the best block reuse solution that minimizes the required reconfiguration time.

6.5.1 JPEG decoder core

In this subsection, the JPEG decoder core that is considered in this case study is described. The main components are two identical decoding units that run in parallel. Such units have a strongly modular design, that arises from the modularity of the JPEG decoding. Such modularity does in turn derives from the different steps involved in the JPEG compression algorithm - and consequently in the decompression (decoding) for image rendering. The decoding unit has been taken from opencores.org [61], where it is provided with an extensive documentation.

JPEG encoding is carried out applying different transformations to the incoming raw image specification. In the following, we take a brief look to the steps composing the *baseline process* encoding algorithm, which is the most used for JPEG compression. Figure 6.5 gives a representation of the sequence of transformations involved in this algorithm.

The first step is a conversion from the RGB (Red, Green, and Blue) components of every pixel in the raw image to the **YCrCb** (Luma, red-difference Chroma, and blue-difference Chroma) encoding of the JPEG image. This is basi-

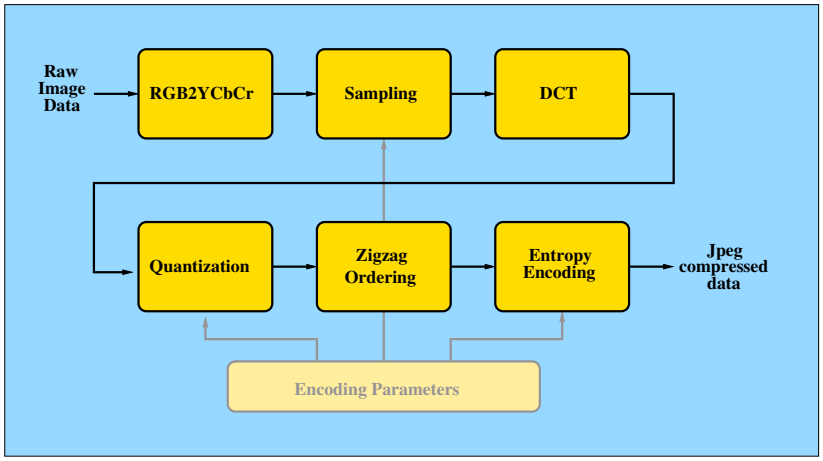


Figure 6.5: Schematic of the baseline process JPEG encoding algorithm (from documentation at [61]).

cally a linear transformation of the RGB vector for every pixel. Since the Chroma components brings less information to the human eye than the Luma one, the Ch-components are sampled in order to reduce the image size. This **sampling** does not involve loss in visual perception quality. A common method is to take the average of Cr and Cb over a 2x2 pixel grid. Anyway, the particular sampling used for encoding is specified, along with other important information, in the JPEG header. The next step in JPEG encoding is the **discrete cosine transformation** (DCT). The DCT is a particular form of Fourier Transformation that ouputs real values only, and provides results identical to Direct Fourier Transformation (DFT) if the input is real. The DCT causes the image to be encoded in the frequency domain and is carried out on two-dimensional 8x8 blocks. Since low-frecuency components carry more information than high-frequency ones, a **quantization** that reduces the image size is possible without apparent loss of quality. This is carried out according to a 8x8 quantization table that is included in the JPEG header. This operation has the effect of setting to zero many high frequency values, that are in the lower right area in every 8x8 block. For this reason, a **zig-zag mapping**

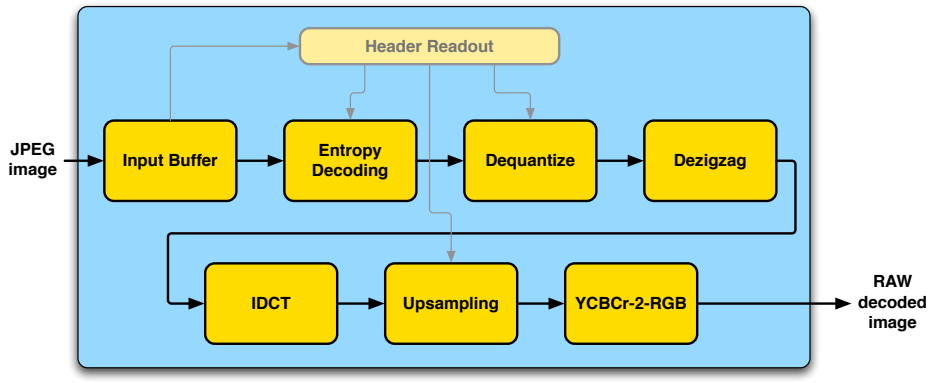


Figure 6.6: Schematic of the JPEG decoding algorithm (adapted from documentation in [61]).

is used to rearrange the order of the squares in the encoding such that long sequences of zeros are at the end, making the final encoding more efficient. The last step, called **entropy encoding**, is the combination of three techniques: *run length encoding*, *variable length encoding* and *Huffman encoding*. These methods allows to pass from a pixel-based representation of the image to a sequence-based one. For example, in run length encoding, a YCrCb color is specified together with the string of consecutive pixels that have that color. The detailed discussion of these encoding techniques is beyond the goal of this section.

The JPEG decoding unit used in this case study carries out the inverse of the described operations in exactly the opposite order, thus providing a raw image -renderizable by a VGA module- starting from its JPEG encoding. The basic structure of the decoder is represented in Figure 6.6.

6.5.2 Design extraction

As said, the core we want to deploy is composed by two units like the one showed in Figure 6.6. The input of of the design workflow is represented by a hierarchy of VHDL files The preprocessing step (Subsection 5.1.3.1) converts any behavioral

and dataflow part of such files. into instantiated components. A new hierarchy of VHDL files is produced by this step. Such specification is then fed into the structural parser (Subsection 5.1.3.2) that saves all the information regarding the structure and the hierarchy of the design in a specifically created data structure. The result of this parsing can be viewed in the form of several graphs which are automatically generated by the parser. There is a graph that represents the structure of each non-leaf node of the hierarchy, In particular, we take a look to the structural graph of the JPEG decoding unit, depicted in Figure 6.7.

In the figure, the components implementing the steps of the decoding algorithm are highlited, so to prove the correspondence of the circuit to the theoretical framework. The other components in the structural graph are the modules for synchronizing and allowing the communication among the algorithmic blocks. In general, every of these extra-modules corresponds to a process -or dataflow instructions- in the original VHDL specification. In order to inspect the hierarchy of the design, the design extraction also provides a hierarchy tree, which is not reported here because of its size.

In the following, some properties of the design are provided. The size of the circuit is 3978 slices[1]. The hierarchy tree is composed of 174 nodes, 162 of which are leaves. The average size of leaf nodes is 24.56 slices, while the standard deviation of leaves size is 114.22, which means that the size of nodes in the flattened circuit highly varies. The extraction of the design requires 117 milliseconds, while 454 milliseconds are required for the creatiion of the corresponding flattened circuit.

6.5.3 Global layout

The next phase of the proposed design flow is the global layout. In this phase, the input application is partitioned and placed over a given multi-FPGA architec-

[1]Dimensions are obtained by performing sysntheses with Xilinx XST [45].

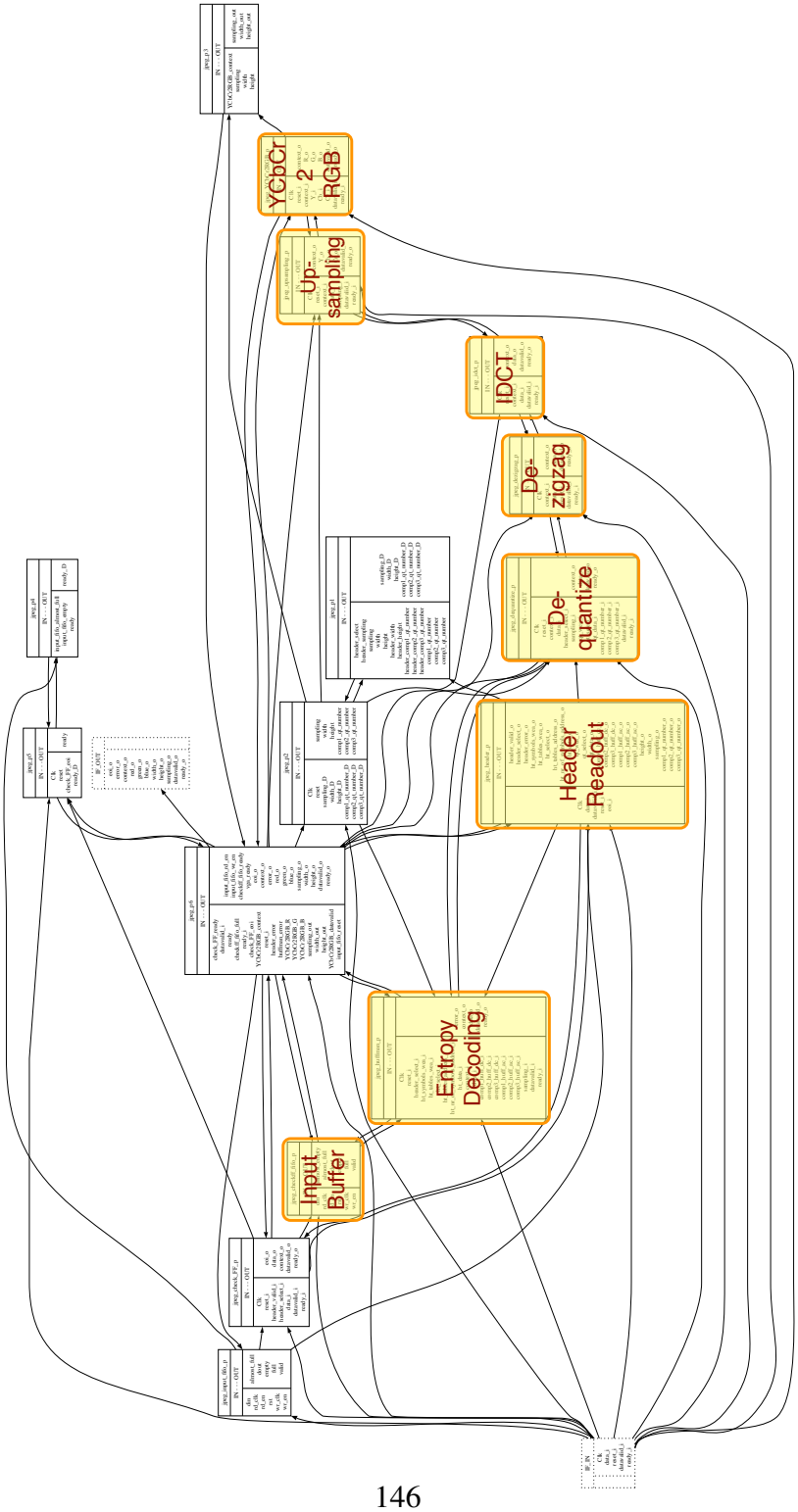


Figure 6.7: Graph representing the structure of the decoding unit.

Table 6.10: Result of different clustering metrics for the case study circuit.

Metric	**Conn**	**CR**	**CD**	*Metis*
# Partition	5	10	7	5 (imposed)
Edge Cut	198	887	591	388
# Iterations	118	137	121	-

ture. Due to the good results reported in Subsection 6.3.2, the bottom-up clustering algorithm (described in Subection 5.2.3.1) is used for partitioning. We assume that the application is to be deployed on an architecture composed by Xilinx XC3S100E devices, belonging to the Spartan3E family [62]. The properties that are important to us are the number of available slices, which is 1960, and the number of I/O pins, equal to 108. Furthermore, we assume that the FPGAs are connected through a crossbar topology: there is a chip which implements the communication channels among the devices. Such communication channels can be reconfigured, thus providing a suitable scenario for our blocks reuse algorithm, whose application to this use case will be trated in the next section. There is no architectural specification regarding the crossbar chip: different types of devices can be used, such as FPGAs or FPIDs (Field Programmable Interconnection Devices (FPIDs). The crossbar can be assumed to be either total or partial. The particular choice of crossbar architecture doesn't affect the methodology and the results proposed in this work. Since a crossbar topology is used, there is no point in actually executing the 1-to-1 placement algorithm, because in such a topology the distance between any pair of FPGAs is equal. Therefore, the remainder of this section will focus on the partitioning step.

The bottom-up clustering algorithm for partitioning is run for all the three metrics described in Subsection 5.2.3.1, namely Connection (Conn), Communication Ratio (CR), and Communication Density (CD), obtaining the results in Table 6.10.

It is clear that the Connection metric works better in this case, both in terms of edge cut and number of obtained partitions. An interesting data is the number

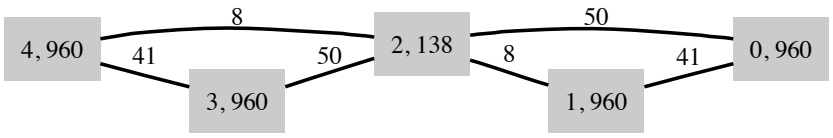


Figure 6.8: Partitions resulting using the bottom-up clustering algorithm.

of iterations: if i is such number, l is the number of leaves in the design, and p is the number of obtained partitions, then the quantity

$$r = l - (i + p)$$

corresponds to the number of times a regularity has been "applied" in the clustering process. The formula is trivially obtained by the fact that the number of iterations in a "normal" clustering algorithm (i.e. that does not exploit any regularity in the design) is $i = l - p$. The quantity r is equal to 39 in out case. This means that throughout the clustering process a collapsing operations has been replicated to another instance of the same parent component -during the same iteration- 39 times. This is not surprising if we take into account the nature of the core under test, which has two identical parallel modules. Nonetheless, the result supports the importance of regularity-driven partitioning in modular-designed applications. As a comparison, the *kMetis* algorithm provides an edge cut equal to 388 when asked to partition the same -flattened- circuit into 5 parts. The clustering algorithm runs in 475 milliseconds. The partitions resulting from using the Connection metric are shown in Figure 6.8. The figure shows a certain symmetry: this is once again due to the exploitation of regularities of the design exploited by the clustering algorithm.

6.5.4 Blocks reuse

In order to show the results of the blocks reuse methodology, we assume that the available FPGA area is smaller than the application dimension. To ground this

fact in a real scenario, we assume that the multi-FPGA system under exam deploys hardware applications on-demand. In this scenario, imagine that several applications are demanded by users prior to the JPEG decoder under exam, leaving just 3000 slices available for deploying it. We then execute the block reuse algorithm bounding the total available area to 3000 slices. The proposed methodology works on the dendrogram produced by the -possibly previously executed- regularity driven bottom-up clustering. Recalling from Subsection 4.2.3, every horizontal cut of the dendrogram -i.e. iteration of the algorithm- corresponds to a complete specification of the original circuit, at different levels of granularity[2]. The information relative to each of these circuits are given as input to the ILP solver which runs the model described in Subsection 5.3.3. Hence, for each horizontal cut, we obtain a solution and a corresponding estimate of the time needed to reconfigure the connections on the crossbar chip. In particular, running the clustering algorithm using the Connection metric, and feeding the ILP solver with the data from each horizontal cut, the reconfiguration time estimations plotted in Figure 6.9 are obtained.

From the plot, it can be inferred that the lowest reconfiguration time estimation is obtained by running the ILP solver on circuit descriptions retrieved from the dendrogram at iterations 14 through 18. In order to understand what this means, we analyze the ILP solutions of one of these five horizontal cuts, namely the one corresponding to iteration 14. The ILP solution says that two blocks of the circuit are *re*used once, i.e. they are used twice in the execution of the application. Of these two blocks, one is a leaf of the original hierarchy, which means that it is composed by a single atomic component, while the other one results from a collapsing operation the clustering algorithm carries out at iteration 14. In order to provide a realistic picture of what have been said, the structure of this latter block is depicted in Figure 6.10.

[2]As already pointed out, the clustering process defines a new hierarchy on the design, that could resemble in some points the original design hierarchy.

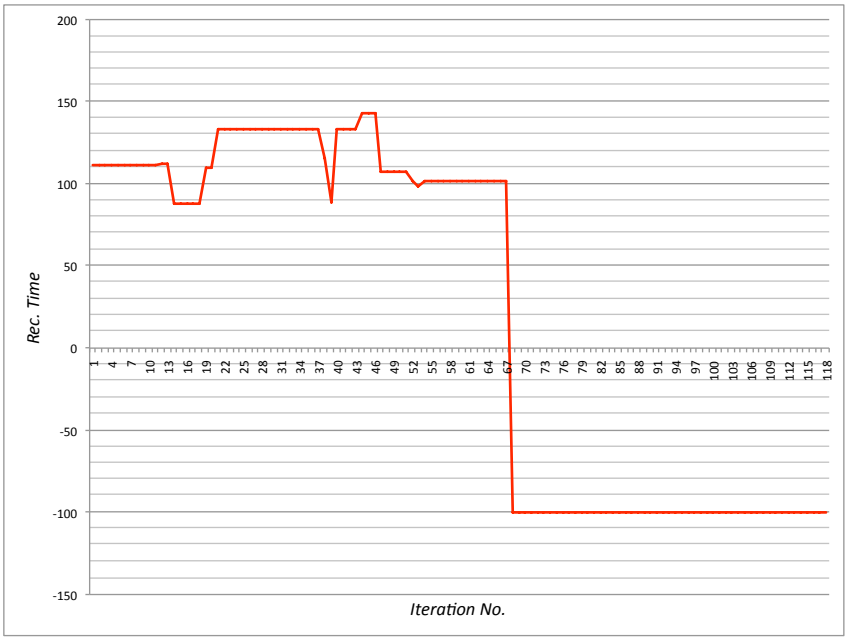


Figure 6.9: Estimated reconfiguration time for different cuts of the dedrogram obtained by the regularity-driven clustering.

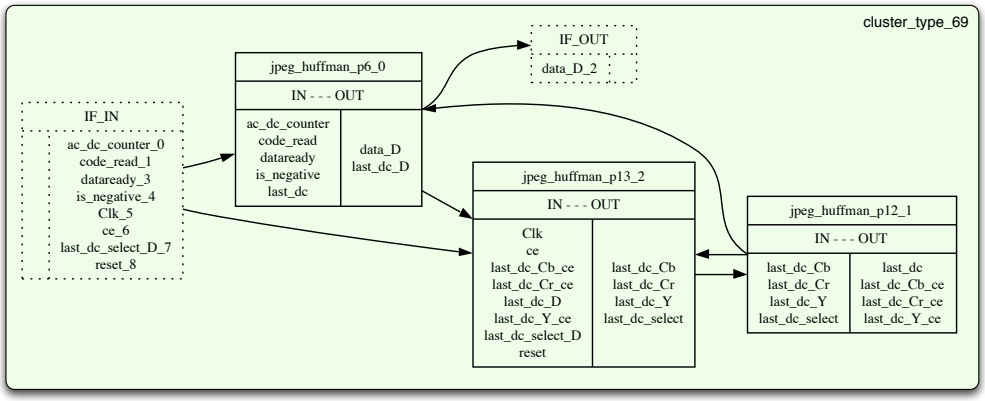


Figure 6.10: Structure of one block which is suggested for reuse in order to minimize the interconnections reconfiguration time.

We are not considering here the impact that such reuse could have on the parallel execution of the two decoders, since it goes beyond the scope of this simple case study. Of course, the introduction of a scheduler on the execution of the application could refine the results provided by the ILP models. Chapter 7 will discuss this and others improvements to the current work.

Chapter 7

Conclusion and future work

In this thesis work a novel multi-FPGA design flow has been proposed, which makes use of blocks reuse through dynamic reconfigurability to make the implementation of large systems feasible even on multi-FPGA architectures with strict physical constraints. The validity of the proposed solutions has been proved in Chapter 6 through several experimental results.

One important result is the development of a global partitioning and placement approach which exploits the design hierarchy and cope with specifications at a process level of granularity. The sequential approach composed by bottom-up clustering followed by a 1-to-1 annealing-based placement has shown to be preferable to an integrated simulated annealing methodology. Besides providing slightly better performances, the sequential method represents a more suitable choice since it already provides elements for the possible execution of the branch of the workflow which copes with reuse and dynamic reconfigurability, which is necessary in case a static global layout is not found.

Another important result is the development of a technique for carrying out choices for the reuse of components in case the application does not statically fits onto the architecture. The design hierarchy is in such case fundamental in order to find the isomorphic structures which are considered for being reused.

The future work must address both the improvement of the proposed techniques and algorithms and the creation of the modules of the workflow which have not been developed in this work.

As concerns the improvements, the proposed bottom-up clustering algorithm can be ameliorated with the addition of more powerful clustering metrics, and the development of solutions to address the intrinsic greediness of the clustering approach. Moreover, an implementation of the algorithm with a smaller time complexity - now of order $O(n^3)$ - has to be provided, since the execution times grows too rapidly as the number of nodes of the application increases. The blocks reuse approach can be enhanced by designing a more accurate estimation function of the reconfiguration time of the system, in order to provide precise information for a subsequent scheduling phase.

As concerns the modules of the workflow, a large amount of work still has to be carried out. First, a robust and effective routing algorithm for both static and dynamic implementations has to be developed, as already pointed out in Section 4.3. Then, a suitable interconnection between the global layout and the reuse phases of the workflow has to be provided, by precisely defining the information that must flow from one phase to the other. The reuse methodology has to be expanded by the design of an algorithm for the scheduling of blocks usage. After the scheduling is performed, the dynamically-interconnected system has to be partitioned and placed; hence, a modification of the proposed algorithms has to be provided in order to cope with such typology of systems. Moreover, a module which carries out the generation of VHDL code at the end of the flow has to be implemented.

Another direction of future works deals with the creation of the architectural solutions for hosting the system resulting from the workflow proposed in this thesis. As explained in Subsection 4.1.4, a bus-based or crossbar multi-FPGA architectural topology are suitable for such implementations.

Bibliography

[1] S. Hauck, "The roles of fpgas in reprogrammable systems," 1998. [Online]. Available: citeseer.ist.psu.edu/hauck98roles.html

[2] Xilinx inc., "Virtex 2.5 v field programmable gate arrays," 2001.

[3] Xilinx Inc., "Ise 9.1i quick ISE 9.1i quick start tutorial."

[4] P.-A. Mudry, F. Vannel, G. Tempesti, and D. Mange, "Confetti : A reconfigurable hardware platform for prototyping cellular architectures," *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–8, 2007.

[5] S. Hauck, "Multi-fpga systems," Ph.D. dissertation, University of Washington, 1995.

[6] M. Khalid, "Routing architecture and layout synthesis for multi-fpga systems," Ph.D. dissertation, University of Toronto, 1999. [Online]. Available: citeseer.ist.psu.edu/khalid99routing.html

[7] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal, "Logic emulation with virtual wires," 1997. [Online]. Available: citeseer.ist.psu.edu/babb97logic.html

[8] V. Rana, M. D. Santambrogio, D. Sciuto, B. Kettelhoit, M. Köster, M. Porrmann, and U. Rückert, "Partial dynamic reconfiguration in a multi-fpga clustered architecture based on linux," in *IPDPS*, 2007.

[9] G. Estrin, "Organization of computer systems–the fixed plus variable structure computer," *Proc. Western Joint Computer Conf., Western Joint Computer Conference, New York*, pp. 33–40, April 1960.

[10] M. D. Santambrogio, "Hardware/Software codesign methodologies for dynamically reconfigurable systems," Ph.D. dissertation, Politecnico di Milano, 2007.

[11] Xilinx Inc., "Two Flows of Partial Reconfiguration: Module Based or Difference Based," Xilinx Inc., Tech. Rep. XAPP290, November 2003. [Online]. Available: http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf

[12] M. Khalid and J. Rose, "Experimental Evaluation of Mesh and Partial Crossbar Routing Architectures for Multi-FPGA Systems," *IFIP IWLAS97, Grenoble, France*, pp. 119–127, 1997.

[13] P. P. Chu, *RTL Hardware Design Using VHDL*. John Wiley and Sons, 2006.

[14] J. Hidalgo, J. Lanchares, and R. Hermida, "Partitioning and placement for multi-FPGA systems using geneticalgorithms," *Euromicro Conference, 2000. Proceedings of the 26th*, vol. 1, 2000.

[15] K. Roy and C. Sechen, "A timing driven N-way chip and multi-chip partitioner," *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pp. 240–247, 1993.

[16] J. de Vicente, J. Lanchares, and R. Hermida, "Placement optimization based on global routing updating for system partitioning onto multi-fpga mesh topologies," in *FPL '99: Proceedings of the 9th International Workshop*

on Field-Programmable Logic and Applications. London, UK: Springer-Verlag, 1999, pp. 91–100.

[17] B. W. Kernighan, S. Lin, "An efficient heuristic procedure for partitioning of electrical circuits," *Bell Systems Technical Journal*, vol. 49, no. 2, pp. 291–307, February 1970.

[18] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *DAC '82: Proceedings of the 19th conference on Design automation*. Piscataway, NJ, USA: IEEE Press, 1982, pp. 175–181.

[19] S. Dutt, "New faster kernighan-lin-type graph-partitioning algorithms," *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pp. 370–377, 1993.

[20] L. A. Sanchis, "Multiple-way network partitioning," *IEEE Trans. Comput.*, vol. 38, no. 1, pp. 62–81, 1989.

[21] S. Dutt and W. Deng, "A probability-based approach to VLSI circuit partitioning," in *Design Automation Conference*, 1996, pp. 100–105. [Online]. Available: citeseer.ist.psu.edu/dutt96probabilitybased.html

[22] S. M. Sait, A. H. El-Maleh, and R. H. Al-Abaji, "General iterative heuristics for vlsi multiobjective partitioning," in *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, 2003, pp. 497–500.

[23] ——, "Evolutionary algorithms for vlsi multi-objective netlist partitioning," *Engineering applications of artificial intelligence*, vol. 19, no. 3, pp. 257–268, April 2006.

[24] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[25] T. W. Manikas and J. T. Cain, "Genetic algorithms vs. simulated annealing: A comparison of approaches for solving the circuit partitioning problem," Department of Electrical Engineering, The University of Pittsburgh, Tech. Rep. 101, May 1996.

[26] J. Cong, "A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design," *Proceedings of the 30th international conference on Design automation*, pp. 755–760, 1993.

[27] C. J. Alpert, J.-H. Juang, and A. B. Kahng, "Multilevel circuit partitioning," in *DAC*, 1997, pp. 530–533.

[28] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, 1998.

[29] G. Karypis, V. Kumar, and kirk Shloegel, "A new algorithm for multi-objective graph partitioning," Department of Computer Science and Engineering, University of Minnesota, Tech. Rep. 003, September 1999.

[30] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *DAC*, 1999, pp. 343–348.

[31] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 7, no. 1, pp. 69–79, 1999.

[32] D. Behrens, K. Harbich, and E. Barke, "Hierarchical partitioning," in *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 470–477.

[33] S. Hauck and G. Borriello, "Logic partition orderings for multi-FPGA systems," in *FPGA*, 1995, pp. 32–38. [Online]. Available: citeseer.ist.psu.edu/hauck95logic.html

[34] C. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 346–365, 1961.

[35] R. Baraglia, R. Perego, J. I. Hidalgo, J. Lanchares, and F. Tirado, "A parallel compact genetic algorithm for multi-fpga partitioning," *pdp*, vol. 00, p. 113, 2001.

[36] A. Kahng, "Futures for Partitioning in Physical design," *Proc. IEEE/ACM International Symposium on Physical Design*, pp. 190–193, 1998.

[37] H. Krupnova, A. Abbara, and G. Saucier, "A Hierarchy-Driven FPGA Partitioning Method," *Proceedings of the 34th annual conference on Design automation conference*, pp. 522–525, 1997.

[38] W.-J. Fang and A. C. H. Wu, "Integrating hdl synthesis and partitioning for multi-fpga designs," *IEEE Des. Test*, vol. 15, no. 2, pp. 65–72, 1998.

[39] W.-J. Fang and A. C.-H. Wu, "Multiway fpga partitioning by fully exploiting design hierarchy," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 1, pp. 34–50, 2000.

[40] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," *IPPS/SPDP Workshops*, pp. 31–36, 1998.

[41] C. Niessen, "Hierarchical design methodologies and tools for VLSI chips," *Proceedings of the IEEE*, vol. 71, no. 1, pp. 66–75, 1983.

[42] S. M. Sait and H. Youssef, *VLSI Physical Design Automation*. World Scientific Publishing, 1999, vol. Lecture Notes Series on Computing, no. 6.

[43] R. Mehra and J. Rabaey, "Exploiting regularity for low-power design," *iccad*, vol. 00, p. 166, 1996.

[44] T. Kutzschebauch and L. Stok, "Regularity driven logic synthesis," in *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 2000, pp. 439–446.

[45] Xilinx, Inc, *XST User Guide*.

[46] ——, *Embedded System Tools Reference Manual*.

[47] IBM Corp., *The CoreConnect Bus Architecture*.

[48] M. Murgida, A. Panella, V. Rana, M. Santambrogio, and D. Sciuto, "Fast ip-core generation in a partial dynamic reconfiguration workflow," *Very Large Scale Integration, 2006 IFIP International Conference on*, pp. 74–79, Oct. 2006.

[49] M. Redaelli, "Task partitioning for the scheduling on partially dynamically reconfigurable fpgas," 2006.

[50] P. J. Ashenden, "The vhdl cookbook," 1990.

[51] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison Wesley Professional, 2001.

[52] C. Donnely and R. Stallman, *Bison. The YACC-compatible Parser Generator*, November 1995.

[53] V. Paxson, *Flex, version 2.5*, 2nd ed., March 1995.

[54] *GNU Linear Programming Kit Reference Manual, Version 4.28*, Free Software Foundation, Inc., March 2008.

[55] A. Makhorin, *Modeling Language GNU MathProg*, Free Software Foundation, Inc., May 2007.

[56] CoreTex Systems, LLC, *Triple-DES Encryption+Decryption Core*, November 2006.

[57] J. Daemen, M. Peeters, G. V. Assche, and V. Rijmen. Noekeon, http://gro.noekeon.org.

[58] C. Bolchini, D. Quarta, and M. D. Santambrogio, "Seu mitigation for sram-based fpgas through dynamic partial reconfiguration," in *GLSVLSI '07: Proceedings of the 17th Great lakes symposium on VLSI*. New York, NY, USA: ACM, 2007, pp. 55–60.

[59] E. Gansner, E. Koutsofios, and S. North, *Drawing graphs with dot*, January 2006.

[60] J. Ellson, E. R. Gansner, E. Koutsofios, and S. C. N. andGordon Woodhull, "Graphviz and dynagraph - static and dynamic graph drawing tools," AT&T Labs, Tech. Rep.

[61] (M)JPEG Decoder. [Online]. Available: http://www.opencores.org

[62] Xilinx, Inc, *Spartan-3E FPGA Family: Complete Data Sheet*, April 2008.

Printed the October 14, 2009 using LaTeX 2ε