THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

Spécialité : **INFORMATIQUE ET MICRO-ÉLECTRONIQUE**

Présentée par : **Zied Marrakchi**

Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PARIS VI

EXPLORATION AND OPTIMIZATION OF TREE-BASED FPGA ARCHITECTURES

Soutenue le : **25 Novembre 2008**

Devant le jury composé de :

M.	Lionnel Torres	LIRMM, Rapporteur
M.	Jean-Luc Danger	ENST, Rapporteur
M.	Michel Minoux	Paris 6
M.	Olivier Lepape	M2000
M.	Christian Masson	BULL
M.	Jean-Luc Rebourg	CEA-DAM
M.	Joachim Pistorius	Altera
M.	Habib Mehrez	Paris 6

Résumé

Les circuits FPGAs (Field Programmable Gate Arrays) sont devenus des acteurs importants dans le domaine du traitement numérique qui a été dominé auparavant par les microprocesseurs et les circuits intégrés spécifiques. Le plus grand défi pour les FPGAs aujourd'hui est de présenter un bon compromis entre une grande souplesse et de bonnes performances (vitesse, surface et consommation). La combinaison de trois facteurs définit les caractéristiques d'un circuit FPGA: la qualité de l'architecture, la qualité des outils CAO de configuration et la conception électrique du FPGA. L'objet de cette thèse est l'exploration de nouvelles architectures et de structures d'interconnexion qui pourront améliorer les performances de ces circuits. En effet, les ressources d'interconnexion occupent 90% de la surface totale et occasionnent 60% de la consommation électrique. Les architectures étudiées présentent des structures matricielles et arborescentes. Les principaux résultats sont les suivants:

• Au départ nous explorons différentes topologies arborescentes et nous comparons leurs surfaces à celles des architectures matricielles. Pour cela, nous développons une plateforme d'outils logiciels permettant d'implanter différents circuits logiques sur l'architecture cible. En se basant sur cette étude expérimentale, nous définissons une nouvelle architecture arborescente. Nous montrons, en nous appuyant sur un modèle d'estimation de surface, que cette architecture permet de réduire la surface totale de 56% par rapport à une architecture matricielle. Ceci est dû essentiellement à une meilleure utilisation des ressources d'interconnexion.

• Nous explorons les effets des différents paramètres de l'architecture proposée: le coefficient de Rent, la taille des groupes logiques et le nombre d'entrées par bloc logique. Ceci permet de régler l'architecture pour l'adapter à des domaines d'applications qui ont des contraintes spécifiques en terme de surface, vitesse et consommation.

• Enfin, nous proposons une architecture qui rassemble les avantages des structures arborescentes et matricielles. Nous unifions les deux structures en construisant des groupes de blocs logiques qui ont localement un réseau d'interconnexion arborescent et qui sont connectés entre eux via un réseau matriciel. Nous montrons que l'architecture obtenue présente un bon compromis entre l'évolutivité de la vue physique et la densité de la surface.

Mots-clés: FPGA, Réseau d'interconnexion, Loi de Rent, Structure arborescente, Structure matricielle, Optimisation, Partitionnement, Placement, Routage, Analyse de timing

Abstract

Today, FPGAs (Field Programmable Gate Arrays) become important actors in the computational devices domain that was originally dominated by microprocessors and ASICs. FPGA design big challenge is to find a good tradeoff between flexibility and performances. Three factors combine to determine the characteristics of an FPGA: quality of its architecture, quality of the CAD tools used to map circuits into the FPGA, and its electrical design. The subject of this dissertation is the exploration of new interconnect topologies and architectures that may play important roles in FPGA performances improvement. In fact interconnect is the dominant factor in terms of area (90%) and power dissipation (60%). The main architectures under exploration have Tree-based or Mesh-Based topology. The main results are the following:

• We first explore different Tree-based architectures and we compare them to Mesh-based architecture in terms of area. For this purpose we develop an exploration tools platform allowing to implement various benchmark circuits on the target architecture. Using experimental evaluation, we define a new Tree-based FPGA architecture and we show that it has good performances and density characteristics. We show, based on total cells area evaluation, that using the proposed topology we achieve a gain of 56% compared to the common Mesh-based FPGA architecture. This is due essentially to the high interconnect utilization achieved by this architecture.

• We explore the effect of different architecture parameters: Rent's ratio, cluster sizes, and LUTs sizes. We show how they interact and the way to tune them to satisfy different specific applicative constraints (density, performance and power).

• Finally, we propose an architecture that takes advantage of both Mesh and Tree strongest points. We unify both structures by building clusters with a Tree-based local interconnect and we connect these clusters by a Mesh-based interconnect. We show that the resulting architecture presents a good tradeoff between layout scalability and area density.

Keywords: FPGA, Interconnect, Rent's rule, Tree-based architecture, Mesh-based architecture, Optimization, Partitioning, Placement, Routing, Timing analysis.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Field-Programmable Gate Arrays are Integrated Circuits (ICs) whose functionality is programmed after manufacturing. They consist of configurable logic blocks and I/O blocks that are interconnected by a configurable routing. FPGAs are configured to implement user circuits by writing into the configuration memory that is embedded within the FPGA. Configuration memory is spread throughout the FPGA and defines the logical function of each configurable logic block and the connections within the configurable routing fabric. Although other methods exist, FPGA configuration memory is typically implemented using static RAM (SRAM). Other technologies used to implement configuration memory include antifuses [J.Greene et al., 1993] and floating gate transistors [S.Brown, 1994]. However, this thesis focuses on SRAM-based FPGA devices exclusively, since SRAM-based FPGAs are the most common commercial FPGAs.

Reconfigurability of FPGAs is fundamentally different from traditional general-purpose microprocessors. An application is implemented on a microprocessor by compiling the application to a stream of instructions that are sequentially decoded and executed by fixed, general-purpose logic resources. Unlike FPGAs, the functionality of a microprocessor's logic resources cannot be modified on a per-application basis. Instead, each application is compiled into a unique stream of instructions that are executed by the microprocessor. Since it is possible to express almost any application as a sequence of instructions, microprocessors are arguably the most flexible computational devices today. However, microprocessors often incur a performance penalty due to their very flexibility. To support flexibility, the fixed logic resources in a microprocessor are deliberately designed to execute certain basic computations efficiently. Consequently, applications that would benefit from customized, tailor-made logic resources often take a performance hit when executed on a general-purpose microprocessors.

While microprocessors are attractive for their flexibility, an Application Specific Integrated Circuit (ASIC) is a device that is customized to a specific application. Since the exact nature of the application is known beforehand, ASIC hardware resources are designed to provide the highest performance implementation for the application. The price paid by ASICs because of their superlative performance characteristics is flexibility. Once an ASIC has been manufactured, it is impossible to modify it to implement

another application, different from the one it was intended for. Further, since the Non-Recurring Engineering (NRE) costs involved in designing and manufacturing an ASIC are comparatively high, it is generally unfeasible to design and fabricate ASICs in low volumes.
Since their introduction in the mid eighties, FPGAs evolved from a simple, low-capacity gate array technology to devices [Stratix, II] [Virtex, 5] that provide a mix of coarse-grained data path units, microprocessor cores, on chip A/D conversion, and gate counts by millions. Today, FPGAs become important actors in the computational devices domain that was originally dominated by microprocessors and ASICs. Just like microprocessors, FPGA-based systems can be reprogrammed on a per-application basis. At the same time, FPGAs offer significant performance benefits over microprocessor implementations for a number of applications. Although these benefits are still generally an order of magnitude less than equivalent ASIC implementations, low NRE costs, fast time-to-market, and flexibility of FPGAs make them an attractive choice for low-to-medium volume applications.
In order to investigate the quality of different FPGA architectures, one needs CAD tools capable of implementing circuits automatically in each FPGA architecture of interest. Once a circuit has been implemented in an FPGA architecture, one needs next accurate area and delay models to evaluate the quality (speed achieved, area required) of the circuit implementation in the FPGA architecture under test.
Three factors combine to determine the characteristics of an FPGA: quality of the FPGA architecture, quality of the CAD tools used to map circuits into the FPGA, and electrical (i.e. transistor-level) design of the FPGA.

1 Research goals and contributions

Despite their design cost advantage, FPGAs impose large area overheads when compared to custom silicon alternatives (ASICs). To illustrate the magnitude of this problem we refer to the work presented in [I.Kuon and J.Rose, 2007] where authors measure the gap between FPGAs and ASICs in terms of logic density, circuit speed and power consumption. It is shown that for circuits containing only look-up table-based logic and flip-flops, the ratio of silicon area required to implement them in FPGAs and ASICs is on average 35. The dynamic power consumption ratio is approximately 14 times. This is due essentially to the FPGA programmable interconnect which is the dominant area factor. In fact FPGAs have 90% routing and 10% logic occupancy. In addition interconnect is the major factor behind power dissipation. According to results shown in [L.Shang et al., 2002], the power dissipation share of routing, logic and clocking resources are 60%, 16% and 14%, respectively.

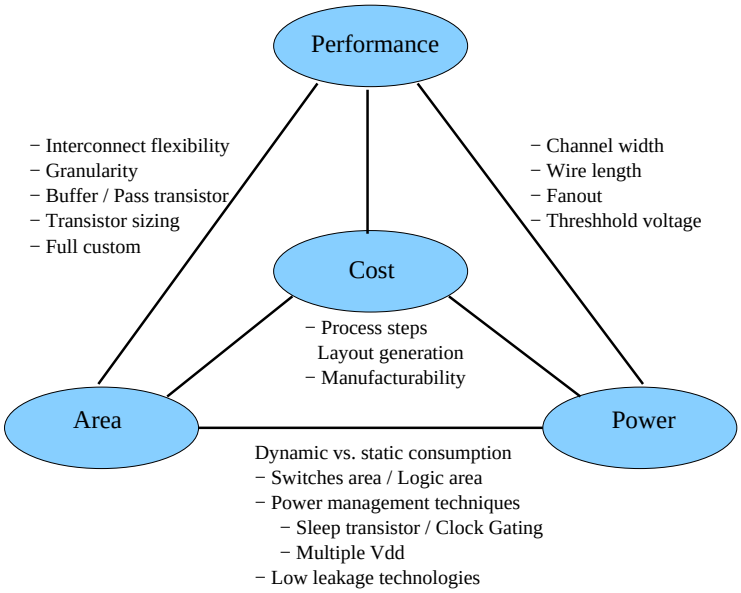


Figure 1: Today's FPGA design challenges [M.Hutton, 2005]

There is a considerable demand for FPGAs with less area, lower delay and power consumption. As illustrated in figure 1, the 4 features characterizing an FPGA are: Area, performance (achieved speed), power dissipation and cost (manufacturing). The tight interaction between these features can be explained by the following examples:

- Reducing total switches number may induce area and power dissipation reduction, however it can reduce flexibility and consequently increases path lengths (performance degradation).

- Using power management techniques reduces power at the cost of area increase, performances degradation and manufacturing cost increase.

Thus FPGA design big challenge is to find a good tradeoff between the 4 different features. A general method used to make FPGAs more efficient is to search for improvements to the numerous algorithmic steps which map a logic circuit into FPGA. Improvements to the logic synthesis step, for example, can reduce the amount and depth of needed logic. Also, improvements to the partitioning, placement, and clustering steps, such as those described in [V.Betz et al., 1999], can reduce interconnect use and delay by shortening connections. Similarly, improvements on the routing step can map critical delay paths to faster connections. Defining an FPGA architecture is a challenge of fixing logic and routing resources so that these algorithms produce the most efficient results possible. Since both algorithms and architectures can be simultaneously defined, there is a significant amount of interaction which can influence the final result.

The aim of this thesis is to present a new efficient way to design interconnection structures for programmable logic: the way in which the programmable wires are connected. We also propose a set of new CAD tools to map circuits on the proposed architecture and to explore its efficiency. The main characteristics of the proposed architecture topology are summarized in the following points:

- Tree-based topology: Most logic designs exhibit locality of connections, which implies a hierarchy in the placement and routing of the connections between logic blocks. We propose a Tree-based FPGA (hierarchical) architecture which aims at exploiting this feature to provide smaller routing delays and more predictable timing behavior. This architecture is created by connecting logic blocks into clusters. Theses clusters are connected recursively to form a hierarchical structure.

- Interconnect depopulation: The interconnect structure in common FPGA architectures is designed generally to maximize logic utilization. Our philosophy is to design architectures with depopulated interconnect. Our purpose is to increase interconnect utilization at the expense of logic utilization. The philosophy behind depopulated routing architecture is to increase silicon utilization through efficient use of the interconnect structure, which accounts for 80-90% of the total area in common Mesh-based FPGA devices.

- Interconnect predictability: We use an interconnect topology based on the Butterfly fat Tree distribution. This structure offers a predictability feature since paths from sources to destinations are limited and predictable. This property is very interesting and can be exploited in the placement phase to improve routability.

- Single driver interconnect: In early FPGA architectures, an interconnect wire was shared and could be driven by many possible resources. Although this made wires bidirectional, it required several large, tri-state buffers per wire and only one of which could be turned on for a specific configuration. Consequently numerous buffers were left unused, which added area, capacitance, power and delay. Modern Mesh FPGA architectures have shifted away from allowing multiple drivers to connect to each interconnect wire. It was shown in [G.Lemieux et al., 2004] that when single-driver wiring is used, area improves by 25% and delay improves by 9%. Common Tree-based FPGA architectures [A.DeHon, 1999] [Y.Lay and P.Wang, 1997] use bidirectional wires. In this thesis we propose a Tree-based interconnect having a single driver at starting point of each wire. Instead of tri-states, each driver has a multiplexer to select from many possible sources. This organization results in unidirectional wires. The benefit of using unidirectional wires is the elimination of bidirectional buffering and tri-states and consequently area reduction, performance improvement and power dissipation reduction.

2 Outline

This section gives a brief overview of the contents of the following chapters:
The two first chapters give an overview of the current state-of-the-art FPGA architectures and configuration CAD tools. We start with describing some academic and industrial architectures with different interconnect topologies. The two most typical FPGA architectures are Mesh-based and Tree-based. Next, we describe the different steps of FPGA configuration flow. A survey of the most commonly used algorithms for each of these steps is provided.
Since the purpose of the thesis is to evaluate different FPGA architectures experimentally, we propose in the third chapter a generic exploration platform. In this environment we describe the different implemented tools and we explain their interaction with the target architecture. To evaluate each architecture we propose models and metrics to measure efficiencies in terms of area and speed performances.
In the three following chapters we present a progressive optimization of a Tree-based architecture. First, in chapter 4 we propose a basic architecture with fully populated switch boxes and optimized signals bandwidth. We conclude, based on a comparison with a common Mesh architecture, that optimizing only signals bandwidth is not sufficient. In chapter 5, we propose to optimize only switch boxes and to use large signals bandwidth. We conclude, based on the same comparison, that area density is much improved at the expense of routability degradation. Thus, in chapter 6 we propose an architecture combining a moderate optimization in terms of switch box population and signals bandwidth. We show experimentally, that this architecture has a good routability and interesting area density.
In the last chapter, we propose an architecture unifying both Mesh and Tree advantages, which are respectively: layout scalability and area efficiency. Finally, conclusions are provided along with possible directions for future research and development.

1 FPGA Architectures

1.1 Introduction

A field Programmable Gate Array (FPGA) is a prefabricated silicon device that can be reconfigured to implement various applications. Reconfigurability of an FPGA is derived from reprogrammable Static Random Access Memory (SRAM) cells. By program-

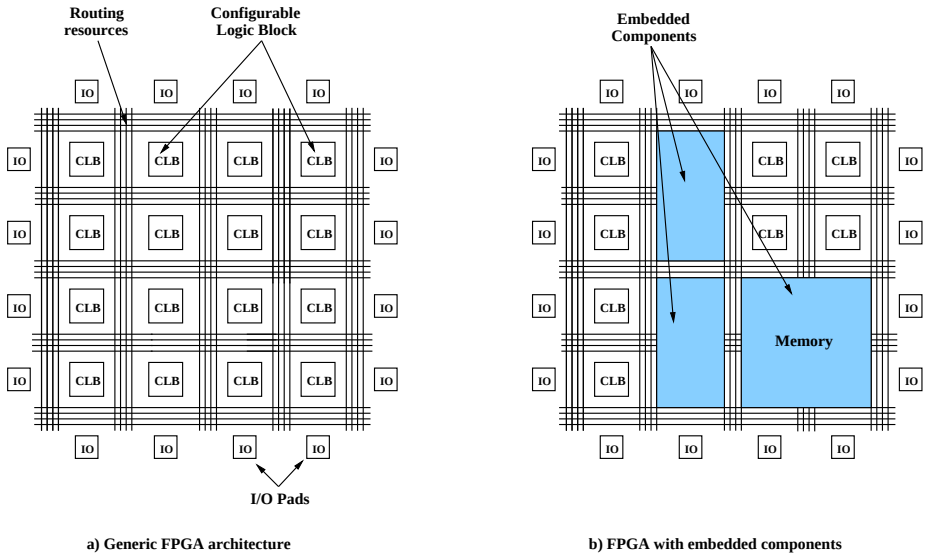


Figure 1.1: FPGA architecture model

ming SRAM cells, the functionality of FPGA logic units can be tailored to implement a particular computation. Logic Blocks and interconnections (figure 1.1 are established by programming SRAM cells to connect prefabricated routing wires together. Thus, any given application can be mapped into an FPGA by programming functionality and connectivity of logic Blocks based on the specific characteristics of the application. The big challenge of FPGA is to provide the maximum flexibility with the minimum area cost. FPGA designers propose different architectures topologies to achieve this tradeoff.
In this chapter we, first, describe the reconfigurable Logic Block which allows functional flexibility. Then, we describe different interconnect topologies which allow routing flexibility.

1.2 Configurable Logic Blocks

Logic blocks implement the logical component of a user circuit. Since FPGAs must be flexible enough to implement any user circuit, FPGA logic blocks must be capable of implementing a wide range of logical functions. To achieve this flexibility, most commercial FPGAs use lookup-table (LUT) based logic blocks. A LUT with k inputs (k-LUT) contains 2^k configuration bits and can implement any k-input function (or gate). Using LUTs with many inputs (large k) reduces the number of LUTs required to implement a user circuit and moreover reduces routing demands; however, it increases the area of the k-LUTs exponentially. By examining speed, area, and routability tradeoffs, previous works have shown that 4-input LUTs result in the fastest and the densest FPGAs [J.Rose et al., 1990] [E.Ahmed and J.Rose, 2000].

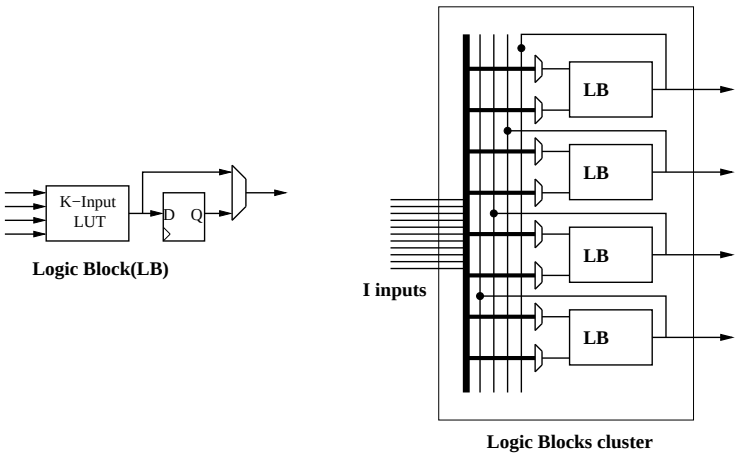


Figure 1.2: LB and Logic clusters

Early FPGAs had logic blocks that included a LUT, a flip-flop, and local interconnect.

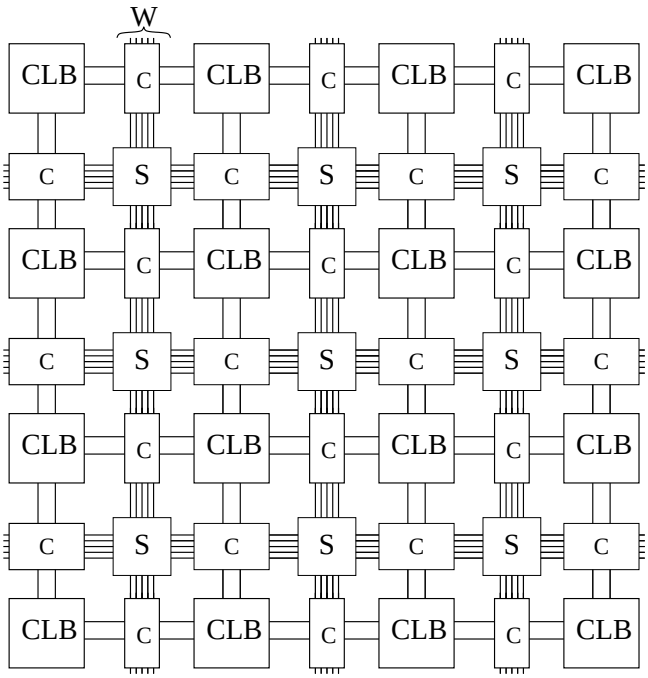
This simple structure, called a logic block (LB), is illustrated in figure 1.2. To enhance their functionality, multiple LBs are combined into logic block with additional local interconnect. This larger structure, called a cluster, is also illustrated in figure 1.2. The advantages of clusters are similar to those of large LUTs: fewer logic blocks, less global routing, and better performance. However, the area penalty incurred by a cluster is much smaller than that of a large LUT. Modern FPGAs contain typically between 4 and 10 logic elements per cluster.
Most commercial FPGAs contain an increasingly larger number of hard macro blocks. As shown in figure 1.1, these macro blocks can include embedded memories, multipliers, or high speed I/Os. In this thesis we are interested to improve architecture performances based on interconnect topologies exploration. The FPGA model used in the following consists of programmable Logic Blocks (LBs) and programmable routing elements.

1.3 Interconnect topologies

FPGA routing interconnect connects internal FPGA components, such as logic blocks and I/O blocks. The performance and the density of an FPGA is largely determined by its routing architecture since routing accounts for most of the area, delay, and power of the FPGA.

1.3.1 VPR-based Mesh interconnect

Mesh based FPGA are also called island-style FPGA, since, as illustrated in figure 1.3, logic blocks look like islands in a sea of configurable routing. Logic Blocks are typically arranged in a grid and are surrounded by horizontal and vertical routing channels. Mesh architectures are most common among academic and commercial FPGAs. The routing fabric consists of pre-fabricated wiring segments and programmable switches organized into rows and columns. The set of switches used to connect a logic block to an adjacent routing channel is called a connection block C. Similarly, the set of switches used to connect intersecting routing channels is called a switch block S. Every routing channel contains W parallel wire tracks, where W is called the channel width. The same width is used for all channels. Figure 1.3 illustrates these various routing structures. The structure of these individual routing components can be parametrized by routing channel width, segments distribution, connection block topology, and switch block topology. Segments distribution describes the lengths of the wire segments in the routing channels. Figure 1.4 shows an example of channel segmentation distribution. Longer wire segments span multiple blocks and require fewer switches, thereby reducing routing



S : Switch Box L : Logic Block C : Connection Box

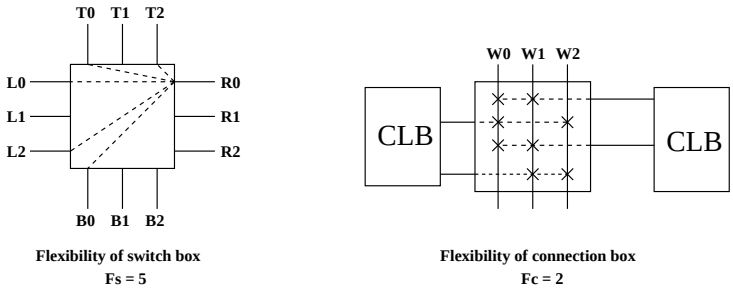


Figure 1.3: Mesh based FPGA architecture

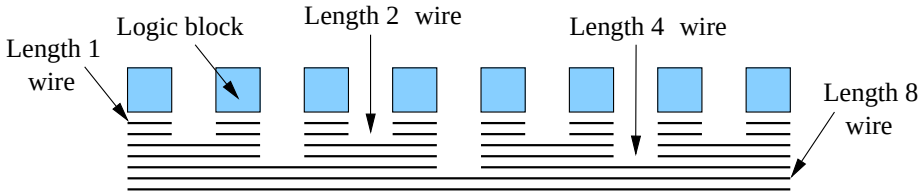


Figure 1.4: Channel segmentation distribution

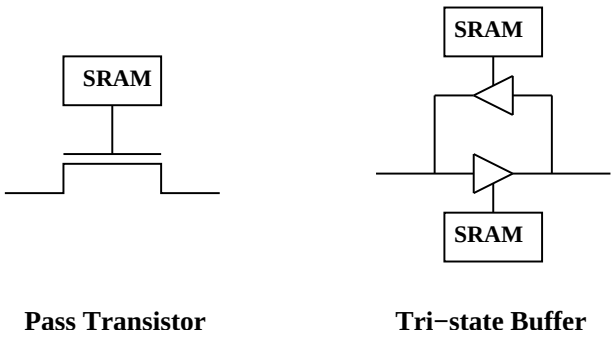


Figure 1.5: Two types of programmable switches used in SRAM-based FPGAs

area and delay. However, they also decrease routing flexibility, which reduces the probability that a user circuit can be routed successfully. Modern FPGAs commonly use a combination of long and short wires in order to balance this tradeoff.
Connection and switch block topologies describe the interconnection pattern within these blocks. In terms of routability, fully populated blocks (that is, blocks for which any incident pin can be connected to any other incident pin) would be optimal. However, in terms of area, the cost would be prohibitive. Previous work [J.Rose et al., 1990, G.Lemieux and D.Lewis, 2002] has shown that connection and switch blocks still provide good routability even when only sparsely populated. Connection block population is defined by $F_{c_{in}}$ and $F_{c_{out}}$ parameters, where $F_{c_{in}}$ is routing channel to cluster input switch density and $F_{c_{out}}$ is cluster output to the routing channel density. Programmable SRAM-based switches within connection blocks and switch blocks can be implemented using either pass-transistors or tri-state buffers, as illustrated in Figure 1.5. Pass-transistor switches require less area and dissipate less power than tri-state buffer switches. However, tri-state buffer switches are faster for connections that span many segments. It is well known by VLSI designers [V.Adler and E.G.Friedman, 1997] that propagation delay through one pass transistor is smaller than corresponding delay through one buffer. However, it is also known that placing many pass transistors in series is much slower than a similar chain of buffers because delay grows quadratically with the former, but linearly with the latter. Routing architectures commonly use a combination of tri-state buffer and pass-transistor switches to reduce area and delay. Global networks, such as

clock and reset networks, are implemented with dedicated routing tracks which are separated from the configurable routing. Like other integrated circuits, FPGA clock distribution networks are designed to minimize skew in order to maximize system performance.
FPGA vendors do not offer FPGAs with different amounts of interconnects, for a given logic capacity. This is surprising since interconnect consumes nearly 90% of the chip area. Some reasons for not offering a variety of interconnect sizes are inventory control, the impact of marketing and sales of inferior or unroutable devices, and the large amount of engineering effort required to develop a single device. The LUT size, the number of LBs in every cluster and the number of inputs per cluster vary with each vendor. For all experiments performed in the main chapters of this thesis, those parameters are chosen to be consistent with previous work [E.Ahmed and J.Rose, 2000]. Note that the channel width of the FPGA is left as a variable. The CAD tools used in this thesis attempt to find the minimum possible channel width required to route a specific circuit. The amount of interconnect is tailored for the circuit to be implemented. This technique allows us to compare different interconnect topologies in terms of routability targeting different applications domains.

1.3.2 Altera's Stratix II architecture

Altera's Stratix II [Stratix, II] architecture is an industrial example of an island-style FPGA (Figure 1.6). The logic structure consists of LABs (Logic Array Blocks), memory blocks, and digital signal processing (DSP) blocks. LABs are used to implement general-purpose logic, and are symmetrically distributed in rows and columns throughout the device fabric. The DSP blocks are custom designed to implement full-precision multipliers of different granularities, and are grouped into columns. Input- and output-only elements (IOEs) represent the external interface of the device. IOEs are located along the periphery of the device.
Each Stratix II LAB consists of eight Adaptive Logic Modules (ALMs). An ALM consists of 2 adaptive LUTs (ALUTs) with eight inputs altogether. Construction of an ALM allows implementation of 2 separate 4-input Boolean functions. Further, an ALM can also be used to implement any six-input Boolean function, and some seven-input functions. In addition to lookup tables, an ALM provides 2 programmable registers, 2 dedicated full-adders, a carry chain, and a register-chain. Full-adders and carry chain can be used to implement arithmetic operations, and the register-chain is used to build shift registers. Outputs of an ALM drive all types of interconnect provided by the Stratix II device. Figure 1.7 illustrates a LAB interconnect interface.
Interconnections between LABs, RAM blocks, DSP blocks and the IOEs are established using the Multi-track interconnect structure. This interconnect structure consists of wire

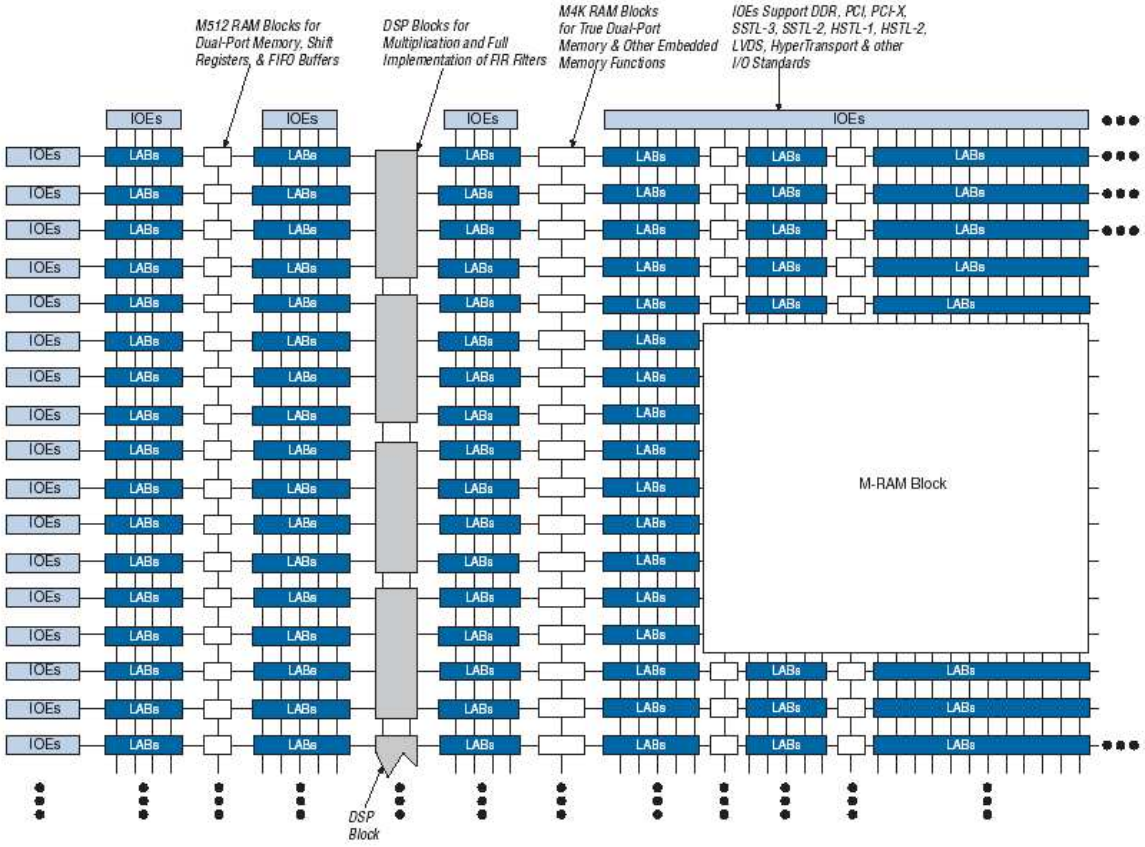


Figure 1.6: Altera's Stratix-II block diagram

segments of different lengths and speeds. The interconnect wire-segments span fixed distances, and run in the horizontal (row interconnects) and vertical (column interconnects) directions. The row interconnects (Figure 1.8) can be used to route signals between LABs, DSP blocks, and memory blocks in the same row. Row interconnect resources are of the following types:

- Direct connections between LABs and adjacent blocks.
- R4 resources that span 4 blocks to the left or right.
- R24 resources that provide high-speed access across 24 columns.

Each LAB owns its set of R4 interconnects. A LAB has approximately equal numbers of driven-left and driven-right R4 interconnects. An R4 interconnect that is driven to the left can be driven by either the primary LAB (Figure 1.8) or the adjacent LAB to the left. Similarly, a driven-right R4 interconnect may be driven by the primary LAB or the LAB immediately to its right. Multiple R4 resources can be connected to each other to

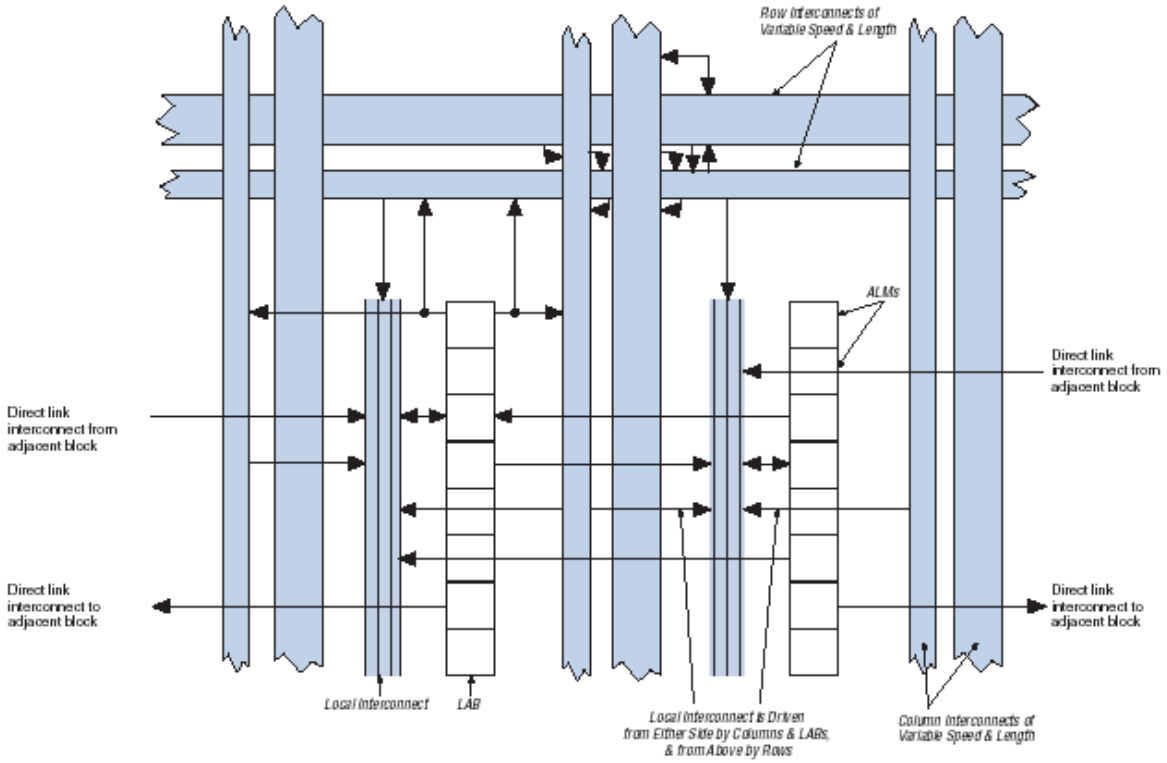


Figure 1.7: Stratix-II Logic Array Block (LAB) structure

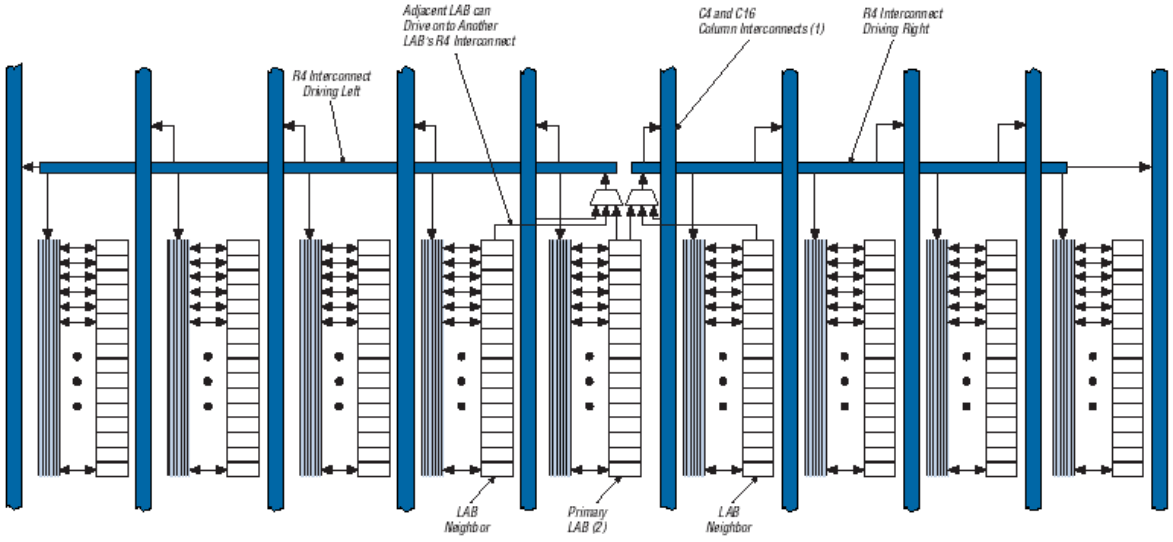


Figure 1.8: R4 interconnect connections

establish longer connections within the same row. R4 interconnects can also drive C4

and C16 column interconnects, and R24 high speed row resources.
Column interconnect structure is similar to row interconnect structure. Column interconnects include:

- Carry chain interconnects within a LAB, and from LAB to LAB in the same column.
- Register chain interconnects.
- C4 resources that span 4 blocks in the up and down directions
- C16 resources for high-speed vertical routing across 16 rows.

Carry chain and register chain interconnects are separated from local interconnect (Figure 1.7) in a LAB. Each LAB has its own set of driven-up and driven-down C4 interconnects. C4 interconnects can also be driven by the LABs that are immediately adjacent to the primary LAB. Multiple C4 resources can be connected to each other to form longer connections within a column, and C4 interconnects can also drive row interconnects to establish column-to-column interconnections. C16 interconnects are high-speed vertical resources that span 16 LABs. A C16 interconnect can drive row and column interconnects at every fourth LAB. A LAB local interconnect structure cannot be directly driven by a C16 interconnect; only C4 and R4 interconnects can drive a LAB local interconnect structure. Figure 1.9 shows the C4 interconnect structure in the Stratix II device.

1.3.3 Multilevel Hierarchical Interconnect

Most logic designs exhibit locality of connections implying a hierarchy in placement and routing of connections between logic blocks. The Hierarchical FPGA architecture attempts to exploit this feature to provide smaller routing delays and more predictable timing behavior. Multilevel hierarchical architecture is created by connecting logic blocks into clusters. These clusters are recursively connected to form a hierarchical structure.
The speed of a net is determined by the number of routing switches it has to pass through and the length of wires. The relationship between switch delay and wire delay is explained in section 3.3.3. In a Mesh structure, the number of segments in series increases linearly with manhattan distance d, between the logic blocks to be connected. An advantage of a Tree connectivity is that the number of switches in series in a route connecting 2 logic blocks increases as a logarithmic function of the manhattan distance. This is illustrated on figure 1.10.
We assume that Multilevel hierarchical interconnect regroups architectures with more than 2 levels of hierarchy and Tree-based ones. For example VPR and APEX architectures are not included in this category since they have only 2 levels of hierarchy.

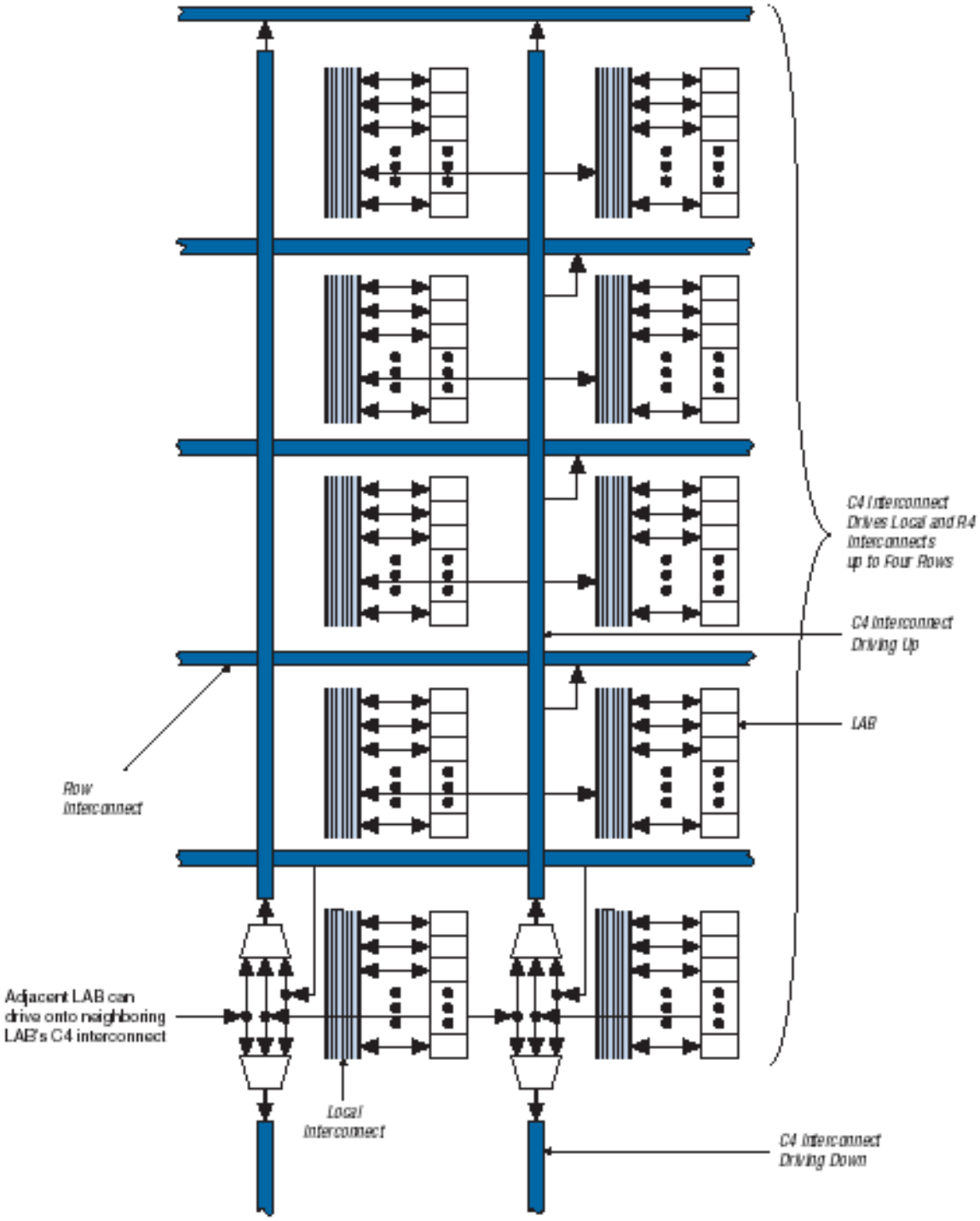
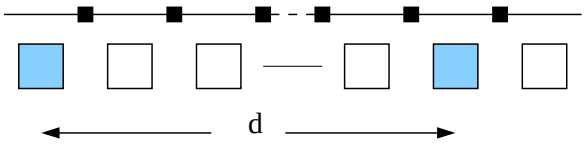
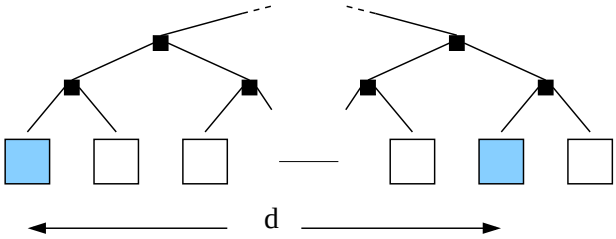


Figure 1.9: C4 interconnect connections



(a) Number of Series Switches in a Mesh Structure



(b) Number of Series Switches in a Tree Structure

Figure 1.10: Mesh vs. Tree structure

HFPGA: Hierarchical FPGA

In the hierarchical FPGA called HFPGA, LBs are grouped into clusters. Clusters are then grouped

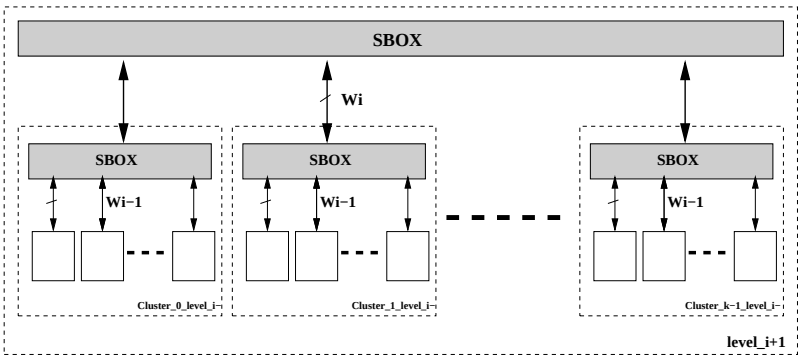


Figure 1.11: Hierarchical FPGA topology

recursively together (see figure 1.11). The clustered VPR mesh architecture has a Hierarchical topology with only two levels. Here we consider multilevel hierarchical architectures with more than 2 levels. In [A.Aggarwal and D.M.Lewis, 1994] and [Y.Lay and P.Wang, 1997] various hierarchical structures were discussed. The HFPGA routability depends on switch boxes topologies. HFPGAs comprising fully populated switch boxes ensure 100% routability but are very

penalizing in terms of area. In [Y.Lay and P.Wang, 1997] authors explored the HFPGA architecture, investigating how the switch pattern can be partly depopulated while maintaining a good routability.

HSRA: Hierarchical Synchronous Reconfigurable Array

A well-known academic hierarchical FPGA is the Hierarchical Synchronous Reconfigurable

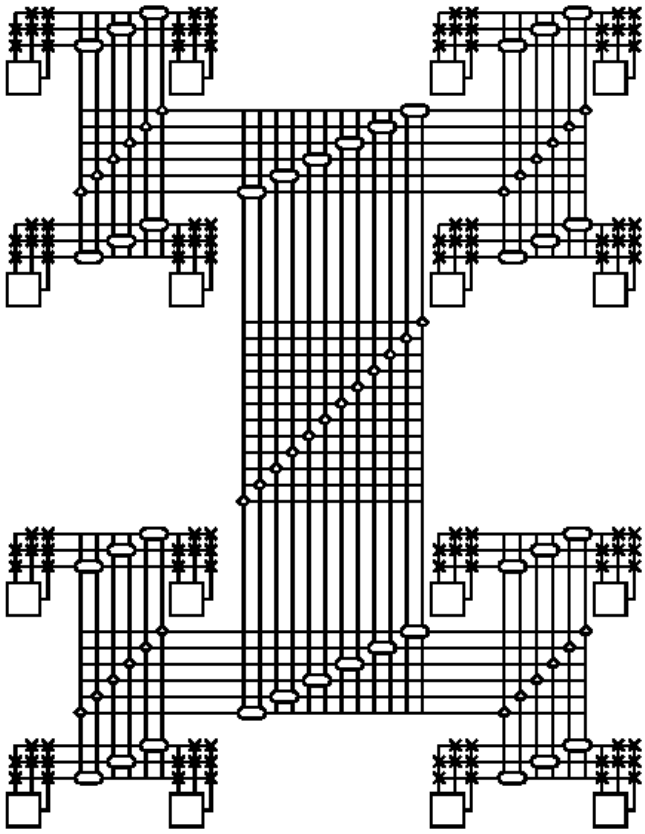


Figure 1.12: HSRA interconnect structure

Array (HSRA) [A.DeHon, 1999]. HSRA has a strictly hierarchical, Tree-based interconnect structure (Figure 2-6). Consequently, HSRA logic and interconnect structures are not as closely coupled as the logic and interconnect structures of island-style FPGAs. Recall that every LAB in Altera's Stratix II device owns R4 and C4 interconnects. In HSRA, the only wire-segments that directly connect to the logic units are located at the leaves of the interconnect tree. All other wire-segments are decoupled from the logic structure. A HSRA logic unit consists of a single 4-LUT / D-FF pair. The input-pin connectivity is based on a choose- k strategy [A.DeHon, 1999], and the output pins are fully connected. The richness of HSRA interconnect structure is defined by its base channel width and interconnect growth rate. The base channel width c is the number

of tracks at the leaves of the interconnect Tree (in figure 1.12, $c = 3$). Growth rate p is the rate at which the interconnect grows towards the root (in figure 1.12, $p = 0.5$). The growth rate is realized using the following types of switch-blocks:

- Non-compressing (2:1) switch blocks - The number of root-going tracks is equal to the sum of the number of root-going tracks of the two children.
- Compressing (1:1) switch blocks The number of root-going tracks is equal to the number of root-going tracks of either child.

A repeating combination of non-compressing and compressing switch blocks can be used to realize any value of p less than one. For example, a repeating pattern of (2:1 1:1) switch blocks realizes $p = 0.5$, while the pattern (2:1 2:1 1:1) realizes $p = 0.67$. A HSRA that has only 2:1 switch blocks provides maximum interconnection bandwidth (i.e. a value of $p = 1$).

APEX Altera

$APEX$ architecture is a commercial product from Altera Corporation which includes 3 lev-

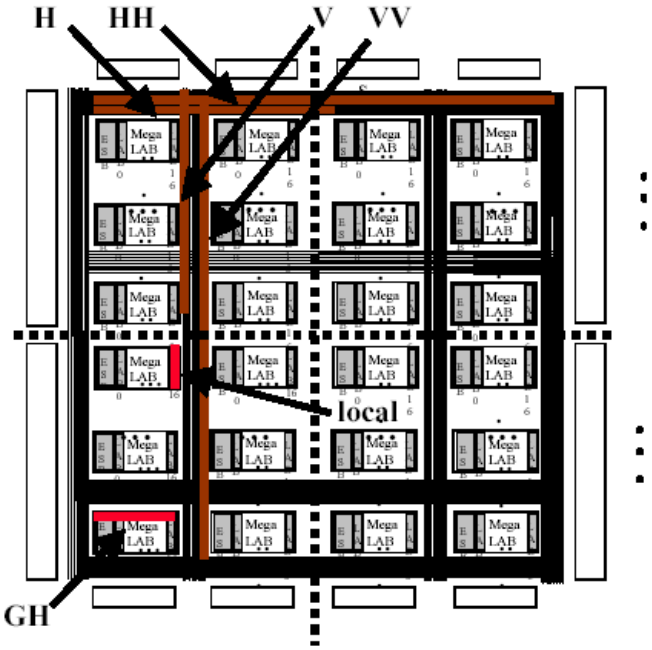


Figure 1.13: The APEX programmable logic devices [M.Hutton et al., 2001]

els of interconnect hierarchy. Figure 1.13 shows a diagram of the APEX 20K400 programmable logic device. The basic logic-element (LE) is a 4-input LUT and DFF pair. Groups of 10 LEs are grouped into a logic-array-block or LAB. Interconnect within a LAB is complete, meaning that a connection from the output of any LE to the input of another LE in its LAB always exists, and any signal entering the input region can reach every LE.

Groups of 16 LABs form a MegaLab. Interconnect within a MegaLab requires an LE to drive a GH (MegaLab global H) line, a horizontal line, which switches into the input region of any other LAB in the same MegaLab. Adjacent LABs have the ability to interleave their input regions, so an LE in LAB_i can usually drive LAB_{i+1} without using a GH line. A 20K400 MegaLab contains 279 GH lines.
The top-level architecture is a 4 by 26 array of MegaLabs. Communication between MegaLabs is accomplished by global H (horizontal) and V (vertical) wires, that switch at their intersection points. The H and V lines are segmented by a bidirectional segmentation buffer at the horizontal and vertical centers of the chip. In figure 1.13, We denote the use of a single (half-chip) line as H or V and a double or full-chip line through the segmentation buffer as HH or VV. The 20K400 contains 100 H lines per MegaLab row, and 80 V lines per LAB-column.

1.4 Conclusion

The interconnect structure of a Mesh-based FPGA is generally designed to maximize logic utilization. Hierarchical FPGAs belong to the class of routing-poor FPGA architectures that are designed to increase interconnect utilization at the expense of logic utilization. The philosophy behind routing-poor architectures is increased silicon utilization through efficient use of the interconnect structure (which may account for $\sim 80-90\%$ of the total area in island-style FPGAs). The most used and studied architecture is the Mesh. In the following chapters we will focus on the Tree-based topology interconnect and we will try to combine it with the Mesh to take advantage of both architectures merits.

2 FPGA Configuration CAD Flow

FPGA architectures have been intensely investigated over the past two decades. A major aspect of FPGA architecture research is the development of Computer Aided Design (CAD) tools for mapping applications to FPGAs. It is well established that the quality of an FPGA-based implementation is largely determined by the effectiveness of accompanying suite of CAD tools. Benefits of an otherwise well designed, feature rich FPGA architecture might be impaired if the CAD tools cannot take advantage of the features that the FPGA provides. Thus, CAD algorithm research is essential to the necessary architectural advancement to narrow the performance gaps between FPGAs and other computational devices like ASICs.
The process of converting a circuit description into a format that can be loaded into an FPGA can be roughly divided into five distinct steps, namely: synthesis, technology mapping, clustering, placement and routing. The final output of FPGA CAD tools is a bitstream that configures the state of the memory bits in an FPGA. The state of these bits determines the logical function that the FPGA implements. Figure 2.1 shows a flowchart of the FPGA CAD flow. In the following sections, we describe the typical algorithms used in each step of the CAD flow.

2.1 Synthesis

Synthesis involves translating a circuit description, traditionally written in a hardware description language (HDL) (e.g. VHDL or Verilog), into a gate-level representation. The gate-level representation is a network consisting of Boolean logic gates and flip-flops. There are no FPGA-specific optimizations performed during synthesis since this is normally a technology independent step. Further details concerning synthesis are omitted because they are beyond the scope

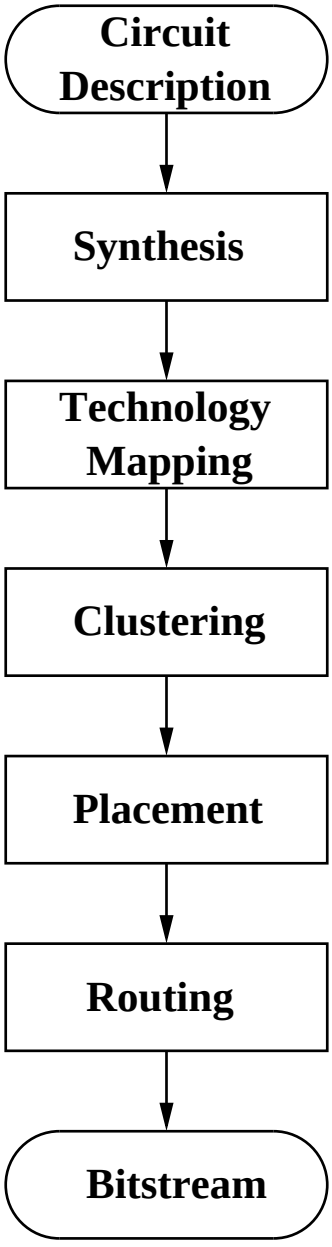


Figure 2.1: FPGA CAD flow

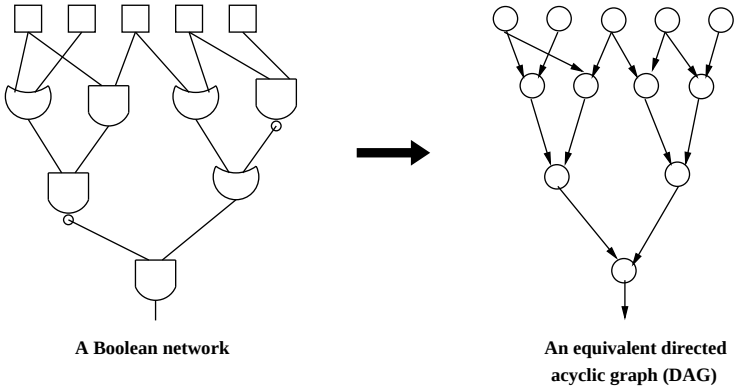


Figure 2.2: Directed Acyclic Graph representation of a circuit

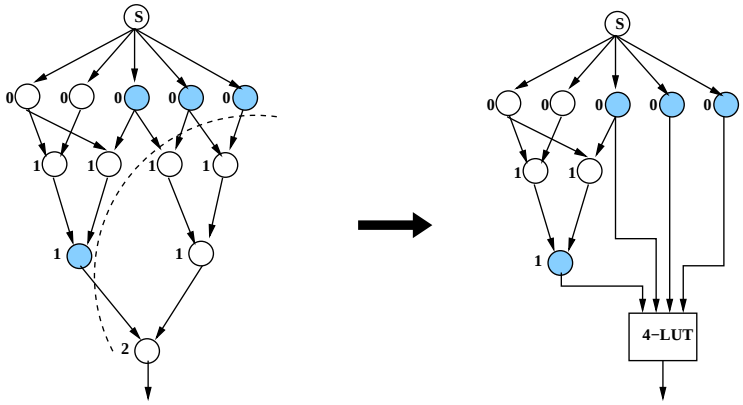


Figure 2.3: Example of Technology Mapping

of this thesis.

2.2 Technology Mapping

The output from synthesis tools is a circuit description of Boolean logic gates, flip-flops and wiring connections between these elements. The circuit can also be represented by a Directed Acyclic Graph (DAG). Each node in the graph represents a gate, flip-flop, primary input or primary output. Each edge in the graph represents a connection between two circuit elements. Figure 2.2 shows an example of a DAG representation of a circuit. Given a library of cells, the technology mapping problem can be expressed as finding a network of cells that implements the Boolean network. In the FPGA technology mapping problem, the library of cells is composed of k-input LUTs and flip-flops. Therefore, FPGA technology mapping involves transforming the Boolean network into k-bounded cells. Each cell can then be implemented as an independent k-LUT. Figure 2.3 shows an example of transforming a Boolean network into k-bounded cells. Technology mapping algorithms can optimize a design for a set of objectives including depth,

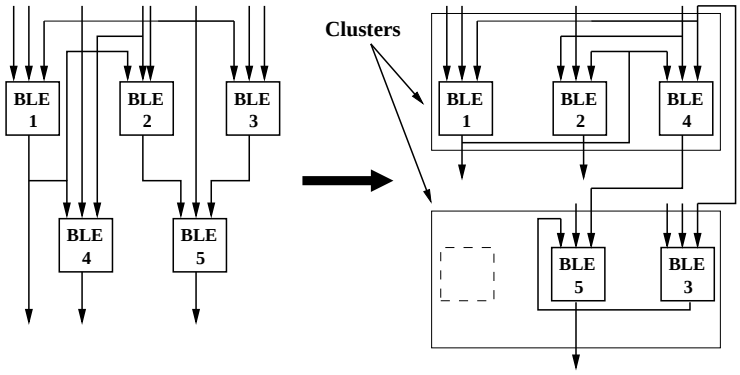


Figure 2.4: Example of clustering

area or power. The FlowMap algorithm [J.Cong and Y.Ding, 1994a] is the most widely used academic tool for FPGA technology mapping. FlowMap is a breakthrough in FPGA technology mapping because it is able to find a depth-optimal solution in polynomial time. FlowMap guarantees depth optimality at the expense of logic duplication. Since the introduction of FlowMap, numerous technology mappers have been designed that optimize for area and run-time while still maintaining the depth-optimality of the circuit [J.Cong and Y.Ding, 1994b] [J.Cong and Y.Hwang, 1995] [J.Cong and Y.Ding, 2000]. The result of the technology mapping step generates a network of k-bounded LUTs and flip-flops.

2.3 Clustering

The logic elements in a Mesh-based FPGA are typically arranged in two levels of hierarchy. The first level consists of logic blocks (LBs) which are k-input LUT and flip-flop pairs. The second level hierarchy groups k LBs together to form logic blocks clusters. The clustering phase of the FPGA CAD flow is the process of forming groups of k LBs. These clusters can then be mapped directly to a logic element on an FPGA. Figure 2.4 shows an example of the clustering process. Clustering algorithms can be broadly categorized into three general approaches, namely top-down [D.Huang and A.Kahng, 1995] [L.Hagen and A.Kahng, 1997], depth-optimal [R.Murgai et al., 1991] [M.Dehkordi and S.Brown, 2002] and bottom-up [A.Marquart et al., 1999] [E.Bozorgzadeh and al, 2004] [A.Singh and M.Marek-Sadowska, 2002]. Top-down approaches partition the LBs into clusters by successively subdividing the network or by iteratively moving LBs between parts. Depth-optimal solutions attempt to minimize delay at the expense of logic duplication. Bottom-up approaches are generally preferred for FPGA CAD tools due to their fast run times and reasonable timing delays. They only consider local connectivity information and can easily satisfy clusters pin constraints. Top-down approaches offer the best solutions; however, their computational complexity can be prohibitive.

2.3.1 Bottom-up approaches

Bottom-up approaches build clusters sequentially one at a time. The process starts by choosing an LB which acts as a cluster seed. LBs are then greedily selected and added to the cluster, applying various attraction functions. The VPack [A.Marquart et al., 1999] attraction function is based on the number of shared nets between a candidate LB and the LBs that are already in the cluster. For each cluster, the attraction function is used to select a seed LB from the set of all LBs that have not already been packed. After packing a seed LB into the new cluster, a second attraction function selects new LBs to pack into the cluster. LBs are packed into the cluster until the cluster reaches full capacity or all cluster inputs have been used. If all cluster inputs become occupied before this cluster reaches full capacity, a hill-climbing technique is applied, searching for LBs that do not increase the number of inputs used by the cluster. The VPack pseudo-code is outlined in algorithm 2.1.
T-VPack [V.Betz et al., 1999] is a timing-driven version of VPack which gives added weight to grouping LBs on the critical path together. The algorithm is identical to VPack, however, the attraction functions which select the LBs to be packed into the clusters are different. The VPack seed function chooses LBs with the most used inputs, whereas the T-VPack seed function chooses LBs that are on the most critical path. VPack's second attraction function chooses LBs with the largest number of connections with the LBs already packed into the cluster. T-VPack's second attraction function has two components for a LB B being considered for cluster C:

$$Attraction(B, C) = \alpha.Crit(B) + (1 - \alpha)\frac{\mid Nets(B) \cap Nets(C) \mid}{G} \tag{2.1}$$

where $Crit(B)$ is a measure of how close LB B is to being on the critical path, $Nets(B)$ is the set of nets connected to LB B, $Nets(C)$ is the set of nets connected to the LBs already selected for cluster C, α is a user-defined constant which determines the relative importance of the attraction components, and G is a normalizing factor. The first component of T-VPack's second attraction function chooses critical-path LBs, and the second chooses LBs that share many connections with the LBs already packed into the cluster. By initializing and then packing clusters with critical-path LBs, the algorithm is able to absorb long sequences of critical-path LBs into clusters. This minimizes circuit delay since the local interconnect within the cluster is significantly faster than the global interconnect of the FPGA.
RPack [E.Bozorgzadeh and al, 2004] improves routability of a circuit by introducing a new set of routability metrics. RPack significantly reduced the channel widths required by circuits compared to VPack. T-RPack [E.Bozorgzadeh and al, 2004] is a timing driven version of RPack which is similar to T-VPack by giving added weight to grouping LBs on the critical path.
iRAC [A.Singh and M.Marek-Sadowska, 2002] improves the routability of circuits even further by using an attraction function that attempts to encapsulate as many low fanout nets as possible within a cluster. If a net can be completely encapsulated within a cluster, there is no need to route that net in the external routing network. By encapsulating as many nets as possible within clusters, routability is improved because there are less external nets to route in total.

```
UnclusteredLBs = PatternMatchToLBs(LUTs,Registers);
LogicClusters = NULL;
while UnclusteredLBs != NULL do
    C = GetLBwithMostUsedInputs(UnclusteredLBs);
    while | C |< k do
        /*cluster is not full*/
        BestLB = MaxAttractionLegalLB(C,UnclusteredLBs);
        if BestLB == NULL then
            /*No LB can be added to this cluster*/
            break;
        endif
        UnclusteredLBs = UnclusteredLB − BestLB;
        C = C ∪ BestLB;
    endw
    if | C |< k then
        /*Cluster is not full - try hill climbing*/
        while | C |< k do
            BestLB = MinClusterInputIncreaseLB(C,UnclusteredLBs);
            C = C ∪ BestLB;
            UnclusteredLBs = UnclusteredLB − BestLB;
        endw
        if ClusterIsIllegal(C) then
            RestoreToLastLegalState(C,UnclusteredLBs);
        endif
    endif
    LogicClusters = LogicClusters ∪ C;
endw
```

Algorithm 2.1: Pseudo-code of the VPack algorithm [V.Betz et al., 1999]

2.3.2 Top-down approaches

The K-way partitioning problem seeks to minimize a given cost function of such an assignment. A standard cost function is net cut, which is the number of hyperedges that span more than one partition, or more generally, the sum of weights of such hyperedges. Constraints are typically imposed on the solution, and make the problem difficult. For example some vertices can be fixed in their parts or the total vertex weight in each part must be limited (balance constraint and FPGA clusters size). With balance constraints, the problem of partitioning optimally a hypergraph is known to be NP-hard [M.Garey and D.Johnson, 1979]. However, since partitioning is critical in several practical applications, heuristic algorithms were developed with near-linear runtime. Such move-based heuristics for k-way hypergraph partitioning appear in [B.Kernighan

and S.Lin, 1970] [C.M.Fiduccia and R.M.Mattheyeses, 1982] [T.Bui et al., 1987].

Fiduccia-Mattheyses algorithm

The Fiduccia-Mattheyses (FM) heuristics [C.M.Fiduccia and R.M.Mattheyeses, 1982] work by prioritizing moves by gain. A move changes to which partition a particular vertex belongs, and the gain is the corresponding change of the cost function. After each vertex is moved, gains for connected modules are updated.
The Fiduccia-Mattheyses (FM) heuristic for partitioning hypergraphs is an iterative improve-

```
partitioning = initial_solution;
while solution quality improves do
    Initialize gain_container from partitioning;
    solution_cost = partitioning.get_cost();
    while not all vertices locked do
        move = choose_move();
        solution_cost += gain_container.get_gain(move);
        gain_container.lock_vertex(move.vertex());
        gain_update(move);
        partitioning.apply(move);
    endw
    roll back partitioning to best seen solution;
    gain_container.unlock_all();
endw
```

Algorithm 2.2: Pseudo-code for FM heuristic [D.A.Papa and I.L.Markov,]

ment algorithm. FM starts with a possibly random solution and changes the solution by a sequence of moves which are organized as passes. At the beginning of a pass, all vertices are free to move (unlocked), and each possible move is labeled with the immediate change to the cost it would cause; this is called the gain of the move (positive gains reduce solution cost, while negative gains increase it). Iteratively, a move with highest gain is selected and executed, and the moving vertex is locked, i.e., is not allowed to move again during that pass. Since moving a vertex can change gains of adjacent vertices, after a move is executed all affected gains are updated. Selection and execution of a best-gain move, followed by gain update, are repeated until every vertex is locked. Then, the best solution seen during the pass is adopted as the starting solution of the next pass. The algorithm terminates when a pass fails to improve solution quality. Pseudo-code for the FM heuristic is given in algorithm 2.2.
The FM algorithm has 3 main components (1) computation of initial gain values at the beginning of a pass; (2) the retrieval of the best-gain (feasible) move; and (3) the update of all affected gain values after a move is made. One contribution of Fiduccia and Mattheyses lies in observing that circuit hypergraphs are sparse, and any move's gain is bounded between plus and minus

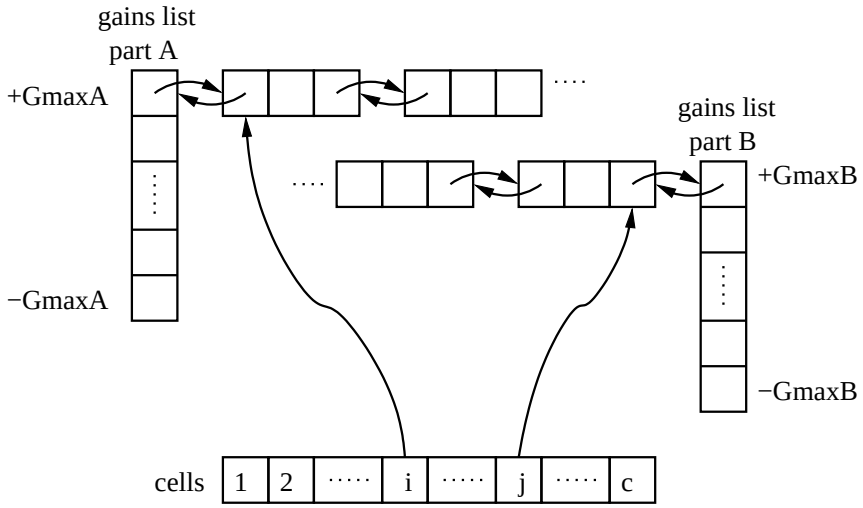


Figure 2.5: The gain bucket structure as illustrated in [C.M.Fiduccia and R.M.Mattheyeses, 1982]

the maximal vertex degree G_{max} in the hypergraph (times the maximal hyperedge weight, if weights are used). This allows prioritizing moves by their gains. All affected gains can be updated in amortized-constant time, giving overall linear complexity per pass [C.M.Fiduccia and R.M.Mattheyeses, 1982]. All moves with the same gain are stored in a linked list representing a "gain bucket". Figure 2.5 presents the gain bucket list structure. It is important to note that some gains G may be negative, and as such, FM performs hill-climbing and is not strictly greedy.

Multilevel Partitioning

The multilevel hypergraph partitioning framework was successfully verified by [C.J.Alpert et al., 1997a] [G.Karypis et al., 1997] [G.Karypis and V.Kumar, 1999] and leads to the best known partitioning results ever since. The main advantage of multilevel partitioning over flat partitioners is its ability to search the solution space more effectively by spending comparatively more effort on smaller coarsened hypergraphs. Good coarsening algorithms allow for high correlation between good partitioning for coarsened hypergraphs and good partitioning for the initial hypergraph. Therefore, a thorough search at the top of the multilevel hierarchy is worthwhile because it is relatively inexpensive when compared to flat partitioning of the original hypergraph, but can still preserve most of the possible improvement. The result is an algorithmic framework with both improved runtime and solution quality over a completely flat approach. Pseudo-code for an implementation of the multilevel partitioning framework is given in algorithm 2.3. As illustrated in figure 2.6, multilevel partitioning consists of 3 main components: clustering, top-level partitioning and refinement or "uncoarsening". During clustering, hypergraph vertices are combined into clusters based on connectivity, leading to a smaller, clustered hypergraph. This step is

```
level = 0;
hierarchy[level] = hypergraph;
min_vertices = 200;
while hierarchy[level].vertex_count() > min_vertices do
    next_level = cluster(hierarchy[level]);
    level = level + 1;
    hierarchy[level] = next_level;
endw
partitioning[level] = a random initial solution for top-level hypergraph;
FM(hierarchy[level], partitioning[level]);
while level>0 do
    level = level - 1;
    partitioning[level] = project(partitioning[level+1], hierarchy[level]);
    FM(hierarchy[level], partitioning[level]);
endw
```

Algorithm 2.3: Pseudo-code for the Multilevel Partitioning algorithm [D.A.Papa and I.L.Markov,]

repeated until obtaining only several hundred clusters and a hierarchy of clustered hypergraphs. We describe this hierarchy, as shown in figure 2.6, with the smaller hypergraphs being "higher" and the larger hypergraphs being "lower". The smallest (top-level) hypergraph is partitioned with a very fast initial solution generator and improved iteratively, for example, using the FM algorithm. The resulting partitioning is then interpreted as a solution for the next hypergraph in the hierarchy. During the refinement stage, solutions are projected from one level to the next and improved iteratively. Additionally, the hMETIS partitioning program [G.Karypis and V.Kumar, 1999] introduced several new heuristics that are incorporated into their multilevel partitioning implementation and are reportedly performance critical.

2.4 Placement

Placement algorithms determine which logic block within an FPGA should implement the corresponding logic block (instance) required by the circuit. The optimization goals consist in placing connected logic blocks close together to minimize the required wiring (wire length-driven placement), and sometimes to place blocks to balance the wiring density across the FPGA (routability-driven placement) or to maximize circuit speed (timing-driven placement). The 3 major classes of placers in use today are min-cut (Partitioning-based) [A.Dunlop and B.Kernighan, 1985] [D.Huang and A.Kahng, 1997], analytic [G.Sigl et al., 1991] [C.J.Alpert et al., 1997b] which are often followed by local iterative improvement, and simulated annealing based placers [S.Kirkpatrick et al., 1983] [C.Sechen and A.Sangiovanni-Vincentelli, 1985]. To investigate architectures fairly

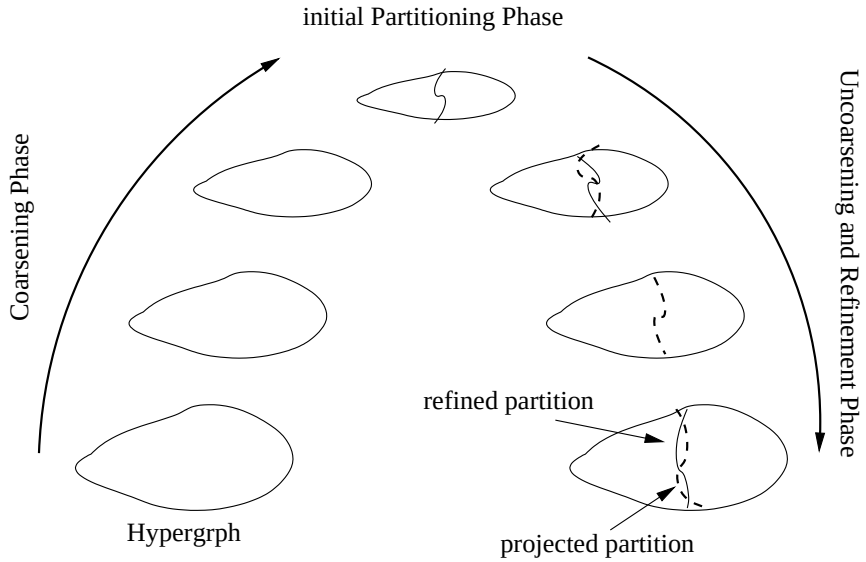


Figure 2.6: Multilevel Hypergraph Bisection

we must make sure that our CAD tools are attempting to use every FPGA's feature. This means that the optimization approach and goals of the placer may change from architecture to architecture. Partitioning and simulated annealing approaches are the most common and used in FPGA CAD tools. Thus we focus on both techniques in the sequel.

2.4.1 Simulated annealing based approach

Simulated annealing mimics the annealing process used to cool gradually molten metal to produce high-quality metal objects [S.Kirkpatrick et al., 1983]. Pseudo-code for a generic simulated annealing-based placer is shown in algorithm 2.4. A cost function is used to evaluate the quality of a given placement of logic blocks. For example, a common cost function in wirelength-driven placement is the sum over all nets of the half perimeter of their bounding boxes. An initial placement is created by assigning logic blocks randomly to the available locations in the FPGA. A large number of moves, or local improvements are then made to gradually improve the placement. A logic block is selected at random, and a new location for it is also selected randomly. The change in cost function that results from moving the selected logic block to the proposed new location is computed. If the cost decreases, the move is always accepted and the block is moved. If the cost increases, there is still a chance to accept the move, even though it makes the placement worse. This probability of acceptance is given by $e^{-\frac{\Delta C}{T}}$, where ΔC is the change in cost function, and T is a parameter called temperature that controls probability of accepting moves that worsen the placement. Initially, T is high enough so almost all moves are accepted; it is gradually decreased as the placement improves, in such a way that eventually the probability of accepting a worsening move is very low. This ability to accept hill-climbing moves that make a placement worse allows simulated annealing to escape local minima of the cost function.

```
S = RandomPlacement();
T = InitialTemperature();
R_limit = InitialR_limit;
while ExitCriterion() == false do
    while InnerLoopCriterion() == false do
        S_new = GenerateViaMove(S, R_limit);
        ΔC = Cost(S_new) − Cost(S);
        r = random(0,1);
        if r < e^(−ΔC/T) then
            S = S_new;
        endif
    endw
    T = UpdateTemp();
    R_limit = UpdateR_limit();
endw
```

Algorithm 2.4: Generic simulated annealing-based placer [V.Betz et al., 1999]

The R_{limit} parameter in algorithm 2.4 controls how close are together blocks must be to be considered for swapping. Initially, R_{limit} is fairly large, and swaps of blocks far apart on a chip are more likely. Throughout the annealing process, R_{limit} is adjusted to try to keep the fraction of accepted moves at any temperature close to 0.44. If the fraction of moves accepted, α, is less than 0.44, R_{limit} is reduced, while if α is greater than 0.44, R_{limit} is increased.
In [V.Betz et al., 1999], the objective cost function is a function of the total wirelength of the current placement. The wirelength is an estimate of the routing resources needed to completely route all nets in the netlist. Reductions in wirelength mean fewer routing wires and switches are required to route nets. This point is important because routing resources in an FPGA are limited. Fewer routing wires and switches typically are also translated into reductions of the delay incurred in routing nets between logic blocks. The total wirelength of a placement is estimated using a semi-perimeter metric, and is given by Equation 2.2. N is the total number of nets in the netlist, $bbx(i)$ is the horizontal span of net i, $bby(i)$ is its vertical span, and $q(i)$ is a correction factor. Figure 2.7 illustrates the calculation of the horizontal and vertical spans of a hypothetical net that has 6 terminals.

$$WireCost = \sum_{i=1}^{N} q(i) \times (bb_x(i) + bb_y(i)) \tag{2.2}$$

The temperature decrease rate, the exit criterion for terminating the anneal, the number of moves attempted at each temperature (InnerLoopCriterion), and the method by which potential moves are generated are defined by the annealing schedule. An efficient annealing schedule is crucial to obtain good results in a reasonable amount of CPU time. Many proposed annealing schedules are "fixed" schedules with no ability to adapt to different problems. Such schedules can

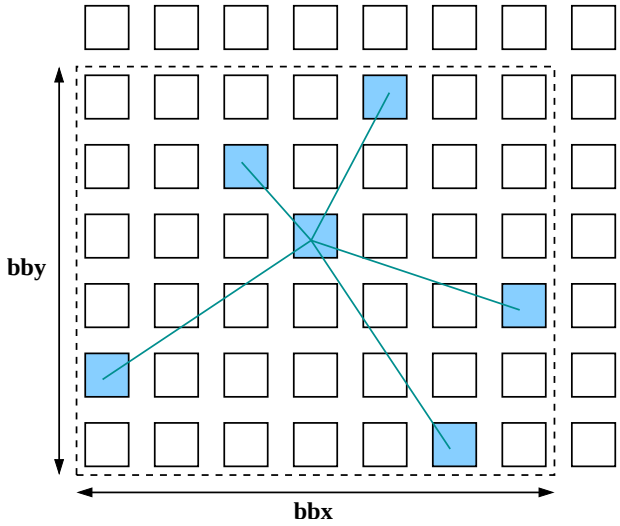


Figure 2.7: Bounding Box of an hypothetical 6-terminals net [V.Betz et al., 1999]

work well within the narrow application range for which they are developed, but their lack of adaptability means they are not very general. In [M.Huang et al., 1986] authors propose an "adaptive" annealing schedule based on statistics computed during the anneal itself. Adaptive schedules are widely used to solve large scale optimization problems with many variables.

2.4.2 Partitioning based approach

Partitioning-based placement methods, are based on graph partitioning algorithms such as the Fiduccia-Mattheyses (FM) algorithm [C.M.Fiduccia and R.M.Mattheyeses, 1982], and Kernighan Lin (KL) algorithm [A.Dunlop and B.Kernighan, 1985]. Partitioning-based placement are suitable to Tree-based FPGA architectures. The partitioner is applied recursively to each hierarchical level to distribute netlist cells between clusters. The aim is to reduce external communications and to collect highly connected cells into the same cluster.
The partitioning-based placement is also used in the case of Mesh-based FPGA. The device is divided into two parts, and a circuit partitioning algorithm is applied to determine the adequate part where a given logic block must be placed to minimize the number of cuts in the nets that connect the blocks between partitions, while leaving highly-connected blocks in one partition.
A divide-and-conquer strategy is used in these heuristics. By partitioning the problem into subparts, a drastic reduction in search space can be achieved. On the whole, these algorithms perform in the top-down manner, placing blocks in the general regions which they should belong to.
In the Mesh FPGA case, partitioning-based placement algorithms are good from a "global" perspective, but they do not actually attempt to minimize wirelength. Therefore, the solutions obtained are sub-optimal in terms of wirelength. However, these classes of algorithms run very fast. They are normally used in conjunction with other search techniques for further quality im-

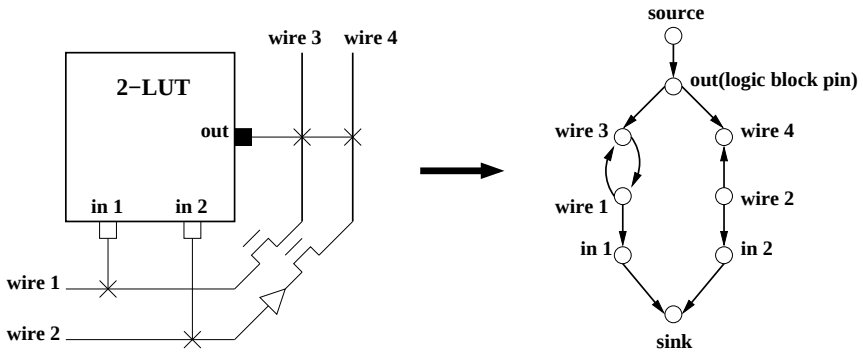


Figure 2.8: Modelling FPGA routing architecture as a directed graph [V.Betz et al., 1999]

provement. Some algorithms [Y.Sanker and J.Rose, 1999] and [P.Du et al., 2004] combine multi-level clustering and hierarchical simulated annealing to obtain ultra-fast placement with good quality.
In the following chapters, the partitioning-based placement approach will be used only for Tree-based FPGA architectures.

2.5 Routing

The FPGA routing problem consists in assigning nets to routing resources such that no routing resource is shared by more than one net. $Pathfinder$ [L.McMurchie and C.Ebeling, 1995] is the current, state-of-the-art FPGA routing algorithm. $Pathfinder$ operates on a directed graph abstraction $G(V, E)$ of the routing resources in an FPGA. The set of vertices V in the graph represents the IO terminals of logic blocks and the routing wires in the interconnect structure. An edge between two vertices represents a potential connection between them. Figure 2.8 presents a part of a routing graph in a Mesh-based interconnect.
Given this graph abstraction, the routing problem for a given net is to find a directed tree embedded in G that connects the source terminal of the net to each of its sink terminals. Since the number of routing resources in an FPGA is limited, the goal of finding unique, non-intersecting trees for all the nets in a netlist is a difficult problem.
$Pathfinder$ uses an iterative, negotiation-based approach to successfully route all the nets in a netlist. During the first routing iteration, nets are freely routed without paying attention to resource sharing. Individual nets are routed using $Dijkstra$'s shortest path algorithm [T.Cormen et al., 1990]. At the end of the first iteration, resources may be congested because multiple nets have used them. During subsequent iterations, the cost of using a resource is increased, based on the number of nets that share the resource, and the history of congestion on that resource. Thus, nets are made to negotiate for routing resources. If a resource is highly congested, nets which can use lower congestion alternatives are forced to do so. On the other hand, if the alternatives are more congested than the resource, then a net may still use that resource. The cost of using a

routing resource n during a routing iteration is given by Equation 2.3.

$$c_n = (b_n + h_n) \times p_n \tag{2.3}$$

b_n is the base cost of using the resource n, h_n is related to the history of congestion during previous iterations, and p_n is proportional to the number of nets sharing the resource in the current iteration. The p_n term represents the cost of using a shared resource n, and the h_n term represents the cost of using a resource that has been shared during earlier routing iterations. The latter term is based on the intuition that a historically congested node should appear expensive, even if it is slightly shared currently. Cost functions and routing schedule were described in details in [V.Betz et al., 1999]. The Pseudo-code of the $Pathfinder$ routing algorithm is presented in algorithm 2.5.

```
Let: RT_i be the set of nodes in the current routing of net i
while shared resources exist do
    /*Illegal routing*/
    foreach net, i do
        rip-up routing tree RT_i;
        RT(i) = s_i;
        foreach sink t_ij do
            Initialize priority queue PQ to RT_i at cost 0;
            while sink t_ij not found do
                Remove lowest cost node m from PQ;
                foreach fanout node n of node m do
                    Add n to PQ at PathCost(n) = c_n + PathCost(m);
                endfch
            endw
            foreach node n in path t_ij to s_i do
                /*backtrace*/
                Update c_n;
                Add n to RT_i;
            endfch
        endfch
    endfch
    update h_n for all n;
endw
```

Algorithm 2.5: Pseudo-code of the $Pathfinder$ routing algorithm [L.McMurchie and C.Ebeling, 1995]

An important measure of routing quality produced by an FPGA routing algorithm is the critical path delay. The critical path delay of a routed netlist is the maximum delay of any combinational

path in the netlist. The maximum frequency at which a netlist can be clocked has an inverse relationship with critical path delay. Thus, larger critical path delays slow down the operation of netlist. Delay information is incorporated into $Pathfinder$ by redefining the cost of using a resource n (Equation 2.4).

$$c_n = A_{ij} \times d_n + (1 - A_{ij}) \times (b_n + h_n) \times p_n \tag{2.4}$$

The c_n term is from Equation 2.3, d_n is the delay incurred in using the resource, and A_{ij} is the criticality given by Equation 2.5.

$$A_{ij} = \frac{D_{ij}}{D_{max}} \tag{2.5}$$

D_{ij} is the maximum delay of any combinational path going through the source and sink terminals of the net being routed, and D_{max} is the critical path delay of the netlist. Equation 2.4 is formulated as a sum of two cost terms. The first term in the equation represents the delay cost of using resource n, while the second term represents the congestion cost. When a net is routed, the value of A_{ij} determines whether the delay or the congestion cost of a resource dominates. If a net is near critical (i.e. its A_{ij} is close to 1), then congestion is largely ignored and the cost of using a resource is primarily determined by the delay term. If the criticality of a net is low, the congestion term in Equation 2.4 dominates, and the route found for the net avoids congestion while potentially incurring delay.
$Pathfinder$ has proved to be one of the most powerful FPGA routing algorithms to date. The negotiation-based framework that trades off delay for congestion is an extremely effective technique for routing signals on FPGAs. More importantly, $Pathfinder$ is a truly architecture-adaptive routing algorithm. The algorithm operates on a directed graph abstraction of an FPGA's routing structure, and can thus be used to route netlists on any FPGA that can be represented as a directed routing graph.

2.6 Timing Analysis

Timing analysis [R.Hitchcock et al., 1983] is used for two basic purposes:

- To determine the speed of circuits which have been completely placed and routed,
- To estimate the slack [J.Frankle, 1992] of each source-sink connection during routing (placement and other parts of the CAD flow) in order to decide which connections must be made via fast paths to avoid slowing down the circuit.

First the circuit under consideration is presented as a directed graph. Nodes in the graph represent input and output pins of circuit elements such as LUTs, registers, and I/O pads. Connections between these nodes are modeled with edges in the graph. Edges are added between the inputs of combinational logic Blocks (LUTs) and their outputs. These edges are annotated with a delay corresponding to the physical delay between the nodes. Register input pins are not joined to register output pins. To determine the delay of the circuit, a breadth first traversal is

performed on the graph starting at sources (input pads, and register outputs). Then the arrival time, $T_{arrival}$, at all nodes in the circuit is computed with the following equation:

$$T_{arrival}(i) = \max_{j \in fanin(i)} \{T_{arrival}(j) + delay(j,i)\}$$

Where node i is the node currently being computed, and $delay(j,i)$ is the delay value of the edge joining node j to node i. The delay of the circuit is then the maximum arrival time, D_{max}, of all nodes in the circuit.
To guide a placement or routing algorithm, it is useful to know how much delay may be added to a connection before the path that the connection is on becomes critical. The amount of delay that may be added to a connection before it becomes critical is called the slack of that connection. To compute the slack of a connection, one must compute the required arrival time, $T_{required}$, at every node in the circuit. We first set the $T_{required}$ at all sinks (output pads and register inputs) to be D_{max}. Required arrival time is then propagated backwards starting from the sinks with the following equation:

$$T_{required}(i) = \min_{j \in fanout(i)} \{T_{required}(j) - delay(j,i)\}$$

Finally, the slack of a connection (i,j) driving node, j, is defined as:

$$Slack(i,j) = T_{required}(j) - T_{arrival}(i) - delay(i,j)$$

2.7 Conclusion

The most important architectural feature of an FPGA is the interconnect structure. During architectures exploration, the effectiveness of an FPGA interconnect structure is evaluated using placement and routing tools. Fortunately, some classes of the used algorithms are architecture-adaptive and can be used to evaluate different structures. In the next section we will present an exploration tools platform that can be adapted to different architectures topologies.

3

Architectures Exploration Environment

To compare different architectures we have first to ensure that they have the same flexibility and the capability to implement netlists with equivalent congestion. Thus, architecture evaluation must be based on benchmark circuits implementation. In this chapter we, first, propose an experimentation platform enclosing a set of architecture adaptive tools. Then, we present different metrics and models to evaluate interconnect structures efficiency in terms of area and speed. Finally, we present an example of an industrial architecture and how we have adapted our tools to target it.

3.1 Exploration Methodologies

The aim of our work is to propose efficient interconnect topologies for FPGA. Efficiency is measured in terms of area and performance (clock frequency). In order to compare and evaluate various architectures, we rely on two different procedures:

- Analytical method based on Rent's rule modeling.
- Experimental method based on benchmarks circuits implementation.

3.1.1 Analytical comparison

The best characterization to date which empirically estimates interconnect requirements is Rent's Rule [B.Landman and R.Russo, 1971]. It states that the number of external in/out signals of a

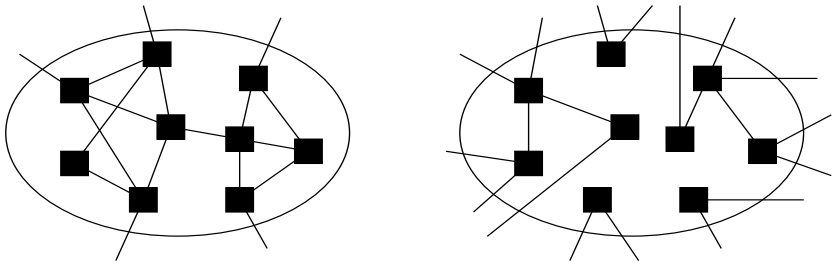


Figure 3.1: Implication of local Rent exponent

partition is proportional to a power of LBs number included in the partition.

$$N_{io} = c.N^p \tag{3.1}$$

N_{io} is the number of in/out interconnections of a region containing N gates, c is a constant corresponding to in/out pins number per gate, and p is an empirical constant defining the growth rate of Rent's rule. Intuitively, as shown in figure 3.1, p presents the locality in interconnect requirement. If most connections are exclusively local and only few of them come from the exterior of a local region, p is small. On the other hand, a region with large p implies that it has relatively more connections with outside cells. It was shown in [J.Pistorius and M.Hutton, 2003], for real logic circuits, that p is typically between 0.5 and 0.6.
Rent's rule is adapted to different interconnect topologies. Using this rule and the work in [W.E.Donath, 1979], Dehon [A.DeHon, 1996] relates the channel width parameter W of a Mesh arranged in a $\sqrt{N} \times \sqrt{N}$ array to Rent's parameters. Therefore he provides a lower bound for W to support a design characterized by Rent's parameters (c, p).

$$W \geq \left(\frac{c}{2^p}\right) N^{p-0.5} \tag{3.2}$$

Thus, the total number of switches per logic block in a Mesh is:

$$N_{switch}(LB) = O(W) = O(N^{p-0.5}) \tag{3.3}$$

Rent's rule can be easily associated to Tree-based topology. In fact a cluster located at level ℓ of the Tree can be considered as a partition, with N_{io} external signals and k^ℓ LBs (leaves).

3.1.2 Experimental comparison

Rent's rule provides an empirical estimation of switching and wiring requirements. Nevertheless this is not sufficient since it does not give accurate information about interconnect routability. FPGA interconnect flexibility is a very important feature since it reflects the architecture potential to route different highly congested benchmark netlists. The best way to verify this point is to implement different benchmark circuits and to evaluate the required area and minimal clock frequency. The proposed configuration flow to implement netlists is presented in the next section.

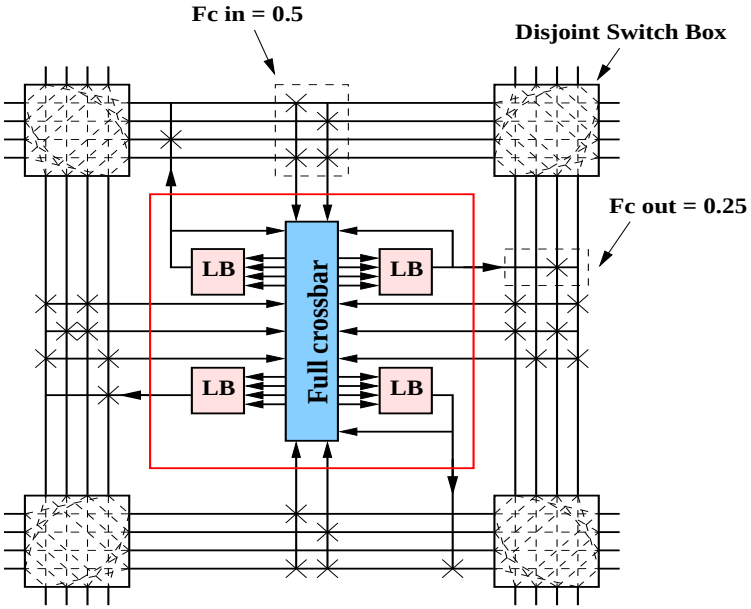


Figure 3.2: RFPGA cluster containing 4 LBs, 10 inputs and 4 outputs

3.1.3 Reference architecture

To evaluate our proposed interconnect topology we compare it to the well known VPR clustered architecture [V.Betz et al., 1999]. This Reference FPGA (RFPGA) uses an uniform routing with single-length segments and a disjoint switch block. Each cluster logic block contains four 4-LUTs, 10 inputs and 4 outputs which are distributed over the cluster sides. LUTs pins are connected to cluster pins using a full local crossbar. Connection block population is defined by $F_{c_{in}}$ and $F_{c_{out}}$ parameters, where $F_{c_{in}}$ is routing channel to cluster input switch density and $F_{c_{out}}$ is cluster output to routing channel density. $F_{c_{in}} = 0.5$ and $F_{c_{out}} = 0.25$ are chosen to be consistent with previous work [E.Ahmed and J.Rose, 2000]. In figure 3.2, we show an RFPGA cluster and its surrounding interconnect.

RFPGA has a basic interconnect topology and present architectures consist in improved versions of it. RFPGA is considered as a reference to evaluate improved architectures performance. For example in [V.Betz et al., 1999], authors state that increasing wire segment length from 1 to 2 logic blocks increases the speed of long connections by 61% and reduces area by about 20%. Work in [G.Lemieux et al., 2004], shows that using directional and single-driver wires instead of bidirectional one improves area efficiency by about 25%. Thus, when we compare our proposed architecture to the basic RFPGA we can get an idea about our architecture efficiency compared to different recent FPGAs. To implement circuits on RFPGA we use the VPR place and route toolset [V.Betz et al., 1999] which provides optimized packing, placement and routing tools.

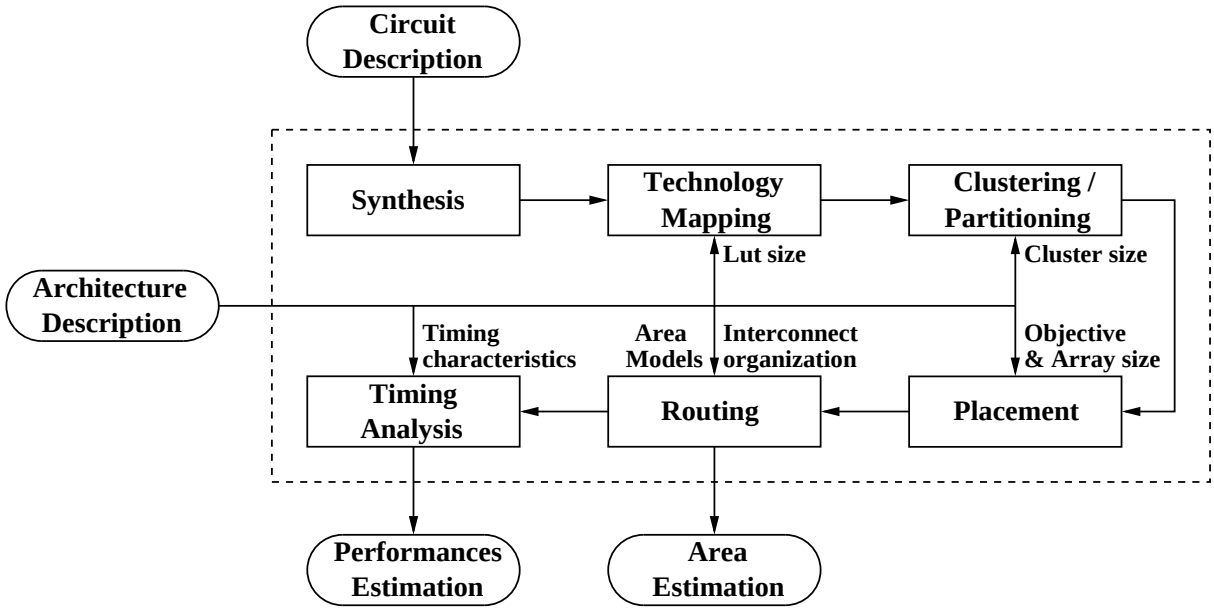


Figure 3.3: Architectures exploration platform

3.2 Exploration Platform

Since we are exploring different architectures topologies, we need as generic as possible CAD tools which can deal with different types of architectures. We propose a set of generic tools requiring a minimum effort to be adapted to a specific architecture topology. In figure 3.3 we present the dependency between each phase in the CAD flow and the target architecture.

3.2.1 Synthesis and Mapping

Synthesis consists in translating a circuit description into a gate-level representation. As illustrated on figure 3.3 this operation is architecture independent. In our flow we use SIS [E.M.Sentovich et al., 1992] synthesis tool. It can be replaced by any other commercial synthesis tool.
As explained in chapter 2, mapping consists in translating the description based on boolean logic gates into a description with k-input LUTs and flip-flops. The only required architecture parameter is k, the LUT inputs number. In our flow we use FlowMap algorithm [J.Cong and Y.Ding, 1994a], which is included in SIS package. As presented in figure 3.3, this tool depends only on LUTs size and can target any interconnect topology. It can be driven by different objectives like timing (depth optimization) and area (LUTs number).
Notice that today, commercial mapping tools can target specific architectures interconnects. Thus in this early stage they can alleviate routing congestion and improve performance.

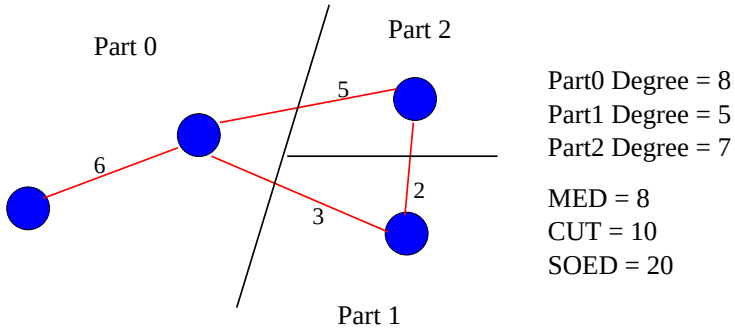


Figure 3.4: Evaluation of several partitioning metrics

3.2.2 Clustering and Partitioning

In recent FPGA architectures, interconnect is organized in multiple hierarchical levels. Hierarchy becomes an interesting feature to improve density, to reduce run time effort (divide and conquer) and to consider local communication. For example in the case of VPR Mesh architecture, interconnect is organized in two levels of hierarchy: 1) Mesh level where clusters are surrounded by depopulated interconnect organized in row and columns and 2) cluster level where LBs are connected using a full crossbar. Stratix [D.Lewis and al, 2003] and [D.Lewis and al, 2005] architecture also has the same number of hierarchical levels but with a more optimized interconnect topology.
In the case of a Tree-based interconnect we get multiple hierarchical levels. Levels number depends on the LBs total number and clusters size (arity). Basically if 2 signals are within the same hierarchy level, it does not really matter where they are within that hierarchy. Similarly, geometrically close cells incur greater delay to get to other locations outside their hierarchical boundary than to distant cells within their hierarchical boundary. Thus, unlike flat or island style device, a hierarchical architecture uses a natural placement algorithm based on recursive partitioning.
Multilevel hierarchical organization is considered in our CAD flow and netlists instances are partitioned between architecture clusters in the best possible way, reducing the desired objectives. We implemented 3 different partitioning objectives:

- CUT: Corresponds to the total number of nets crossing parts boundaries.
- SOED (Sum of External Degree): External part degree corresponds to the number of nets crossing a part boundary.
- MED (Max of External Degree): Corresponds to the maximum degree over all parts.

These objectives can be combined or considered separately. In figure 3.4, we present an example of partitioning and an evaluation of the 3 different objectives.
As presented in chapter 2, there are two main partitioning approaches: bottom-up (clustering) and top-down. The choice between both approaches depends on levels number, clusters size, clusters number at each level and problem constraints. For example t-vpack [A.Marquart et al., 1999] a bottom-up clustering tool is used to construct clusters in the case of VPR Mesh architecture. In [Z.Marrakchi et al., 2005], we proposed to replace t-vpack with hMetis [G.Karypis

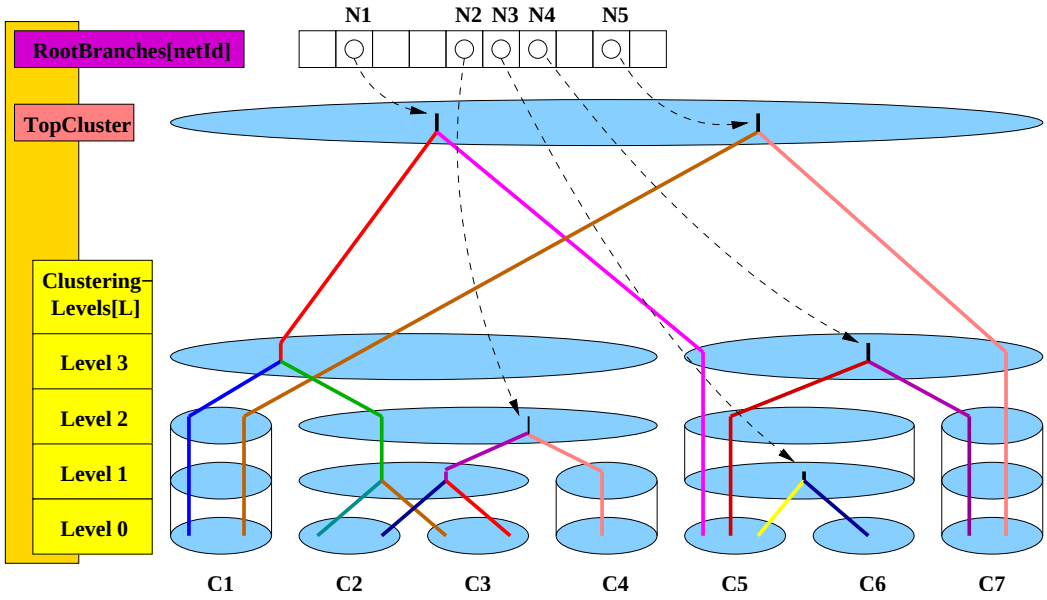


Figure 3.5: Mangrove data structure: Multilevel clustered hypergraph

and V.Kumar, 1999] a top-down partitioner. We showed that we can improve results slightly in terms of external nets reduction, at the cost of an important run time increase. In fact top-down approaches based on FM refinement heuristics are efficient when we target a small number of clusters (parts) of important size (balance constraint). Conversely in the case of VPR Mesh architecture clusters size is small (between 4 and 16) and clusters number is generally important. To investigate partitioning approaches, we used a multilevel hypergraph data structure called $Mangrove$. It provides a development framework for efficient modeling of hypergraph nested partitions. It offers a compact C++ data structure and a high level API. As illustrated in figure 3.5, this structure is organized as follows:

- $ClusteringHierarchy$: holds a vector of nested partitions called $ClusteringLevel$, and refers to a unique enclosing cluster $TopLevelCluster$,
- $ClusteringLevel$: corresponds to the set of clusters at the partitioning at a given level. A $clusteringLevel$ corresponds to an hypergraph where nodes are clusters located at this level,
- $Cluster$: Aggregates sub-clusters belonging to a lower $ClusteringLevel$ (unless leaf one). A $Cluster$ may cross multiple levels and has $UpperLevel$ and $LowerLevel$ identifiers,
- Net: presents a tree of branches,
- $Branch$: represents the net (signal) crossing point of a cluster boundary. Branch bifurcates within a cluster if the net crosses at least 2 sub-clusters.

Since in Mangrove a $clusteringLevel$ can be added at any level, this structure can be used in different partitioning approaches: Bottom-up and top-down. The combination of both approaches

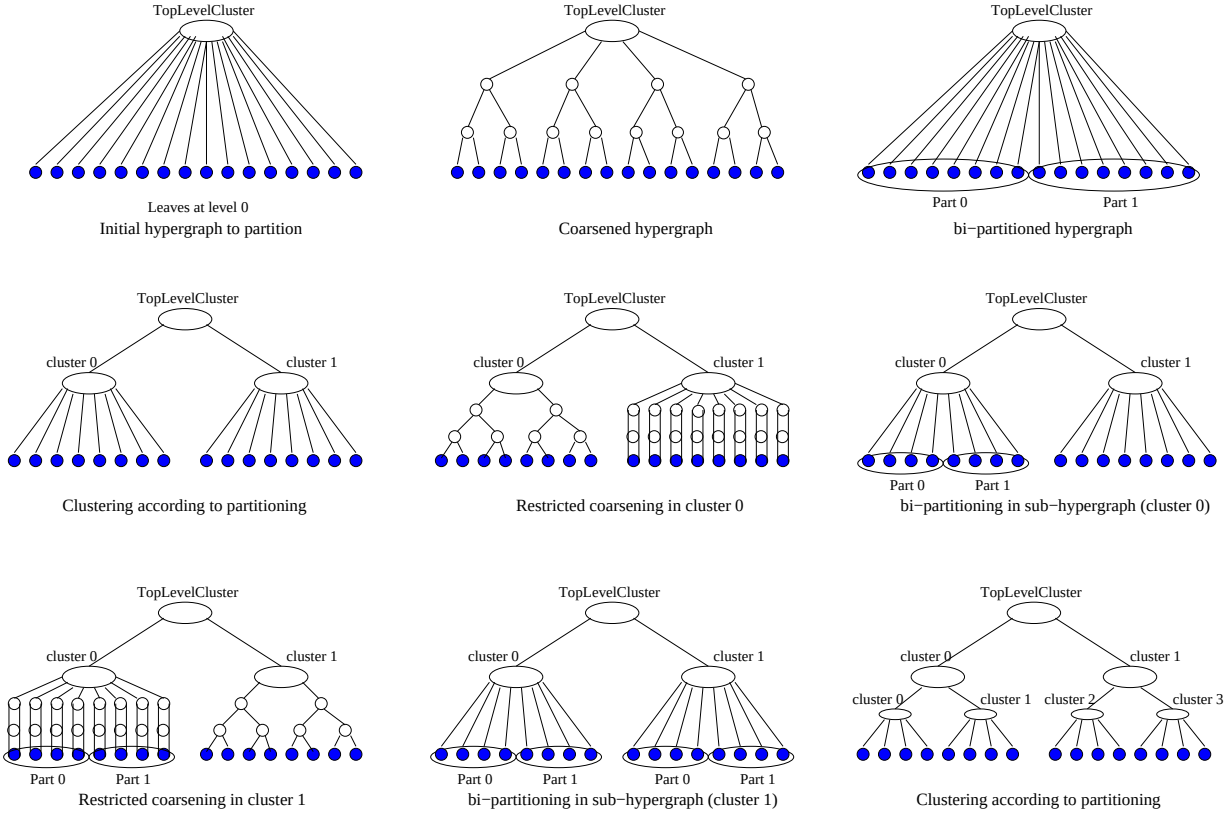


Figure 3.6: 2-levels recursive bi-partitioning steps

leads to an efficient multilevel partitioner where first multilevel bottom-up coarsening is run and then top-down multilevel refinement is applied. In figure 3.6, we show the different steps of recursive netlist partitioning based on a multilevel approach. The netlist is first partitioned into 2 parts (first level) and then instances inside each part are partitioned into 2 fractions. In each partitioning phase we apply a multilevel coarsening followed by a multilevel refinement. Finally, we obtain the partitioning result corresponding to each level. The final result describes how instances are distributed between clusters of the Tree-based topology. Recursive partitioning is also interesting to reduce run time since it allows to avoid applying FM heuristics directly on a large number of parts, which can dramatically increase partitioning run time according to [L.A.Sanchis, 1989].

3.2.3 Placement

Placement algorithms determine which logic block within an FPGA should implement the corresponding instance required by the circuit. In a cluster-based architecture (VPR clustered Mesh or multilevel Tree), depending on interconnect structure, we identify two different placement problems:

- Mesh clusters placement: This placement problem consists in determining netlist logic clusters positions on a 2D grid of clusters. Mesh interconnect is closely approximated by geometric proximity. We implemented a simulated annealing based algorithm where the objective is to reduce the total wirelength. A logic cluster is selected at random, and a new location in the 2D grid for it, is also selected randomly.

- Intra-clusters placement: This placement problem consists in re-arranging logic blocks or sub-clusters inside an owner cluster. In fact depending on the cluster local interconnect, logic blocks ordering can have an important impact on routability. In the case of VPR Mesh architecture, clusters local interconnect corresponds to a full crossbar. Thus, LBs positions are equivalent and no local arrangement is needed. However, using a full crossbar is very penalizing and leads to cluster LBs number limitation. Local interconnect depopulation and especially the way logic blocks outputs are connected, may induce important constraints on LBs ordering. To solve such a problem, simulated annealing can easily be adapted by choosing a suitable objective function and allowing LBs to move only inside their owner clusters.

To solve both placement problems we use an "adaptive" annealing schedule [M.Huang et al., 1986] based on statistics computed during anneal itself. The only elements to tune depending on the target problem are the objective functions to optimize and the movement generator component.

3.2.4 Routing

As stated in chapter 2, FPGA routing consists in assigning netlist signals to routing resources such that no routing resource is shared by more than one net. Thus routing is interconnect dependent. Fortunately, $Pathfinder$ [L.McMurchie and C.Ebeling, 1995] is a truly architecture-adaptive routing algorithm, since it can deal with any graph presenting the interconnect routing resources. In this way the only element depending on architecture interconnect is the routing graph. Our implementation is organized into 2 parts: routing graph generator (architecture dependent) and routing algorithm (architecture independent).

3.2.5 Timing Analysis

Timing analysis evaluates performances of a circuit implemented on a FPGA in terms of functional speed. Thus, once an application is completely placed and routed we estimate the minimum feasible clock to run it. To achieve timing analysis we need 2 different graphs:

- Routed graph: Describes the way netlist instances are routed using architecture resources. This graph allows to evaluate routing delays between netlist instances connections. A path connecting two instances crosses several wires and switches. The connection delay is equal to the sum of resources delays.

- Timing graph: It is a direct acyclic graph generated from the netlist hypergraph. Nodes correspond to instances pins and edges to connections. Based on the resulting routed graph, each edge is labeled with the corresponding routed connection delay. The minimum required clock period is determined via a breadth-first traversal applied on this graph.

Only the routed graph is architecture dependent. Timing graph generation and critical path extraction depend only on netlist to implement.

3.3 Area and Delay Models

This section describes the area and delay models used to compute performance metrics for FPGA architectures under investigation. Based on these metrics and models, we can compare architectures efficiencies and achieve different tradeoffs.

3.3.1 Switches requirement

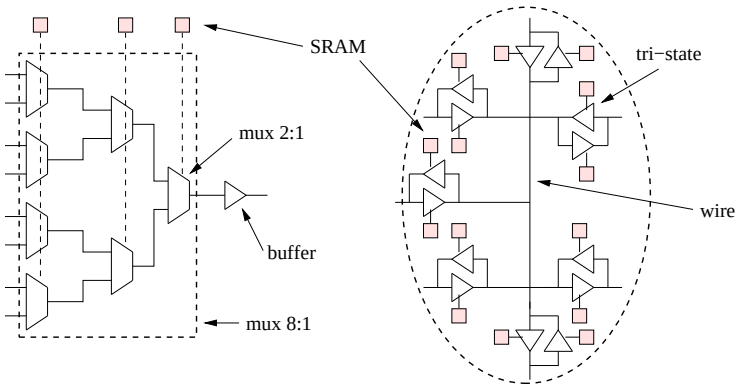


Figure 3.7: Unidirectional vs. bidirectional wires

As mentioned in [V.Betz et al., 1999], discussions with FPGA vendors have revealed that transistor area, and not wiring density, is the area limiting factor. The use of directional wires in Virtex I also suggests that routing area is transistor-dominant and must be reduced. As it was explained by DeHon in [A.DeHon, 2001], the large area of switches compared to wires is one of the key reasons why we have to care about the number of switches required by a network. If the wire pitch is 5 to 8λ, the area of a wire crossing is 25-64λ^2. The area of static memory cell used to configure a switch is roughly 1200λ^2. A switch transistor size is 2500λ^2. In this case the ratio switches area/wires area can reach the value of 40. This ratio increases if we want more than just a pass gate for the switch. We may want to rebuffer the switch or even add a register to it. Such switch can easily be 5-10Kλ^2. The large area ratio means that we definitely need to take much care about switch count in the interconnect.

Cell	Area λ^2
sram	30×50
tri-state	35×50
buffer	20×50
flip-flop	90×50
mux 2:1	35×50

Table 3.1: Standard cells characteristics

In this work we consider 2 area estimating models:

- Switches count: Since interconnect takes until 90% of the total FPGA area, we are interested in evaluating the required interconnect switches number.

- Area evaluation: We estimate the layout area as the sum of areas required for all logic cells in an FPGA. As presented in figure 3.7, switching cells depend on the interconnect structure and especially on wires directions (unidirectional / bidirectional). We use symbolic standard cells library [A.Greiner and F.Pecheux, 1992] to estimate the FPGA required area. Different cells areas are presented in table 3.1.

3.3.2 Wiring requirement

To estimate the required wiring area in a two-dimensional layout, we refer to Thompson's argument about bisection width [C.Thompson, 1979]. He defines the minimal bisection width of a graph as the number of cuts needed to slice it in half. In figure 3.8 we show the smallest number of edges whose removal disconnects one half of the vertices from the other. Let the minimum

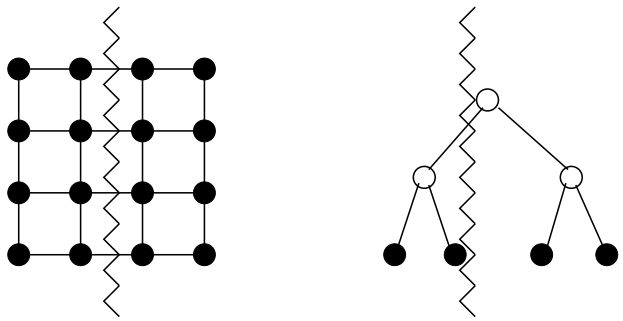


Figure 3.8: The minimal bisection width of a Mesh and a binary Tree

bisection width of a network be $\propto (W)$ (proportional to W), meaning that in any layout, $\propto (W)$ wires are necessarily crossing between the two halves of the layout. When we are limited to 2D-VLSI, this means $\propto (W)$ wires must cross the 1D-line that bisects the chip. This fact gives a lower bound of $\propto (W)$ on the width of the chip, if we assume a fixed number of wire layers. Since this

property holds recursively, we can establish that both the width and height of the layout must be $\propto (W)$, making the entire chip area $\propto (W^2)$.
This property (relation between bisection width and wiring area) is an interesting feature to estimate whether FPGA area is dominated by wires or switches.

3.3.3 Delay Model

The delay through the routing network may easily be dominant in a programmable technology. Care is required to minimize interconnect delays. The 2 following factors are significant in this respect:

- Wires delay: Delay on a wire is proportional to distance and capacitive loading (fanout). This makes interconnect delay roughly proportional to distance run. Consequently short signals runs are faster than long signals runs.
- Switches delay: Each programmable switches in a path (crossbar, multiplexer) adds delay. This delay is generally much larger than the propagation or fanout delay. Consequently, one generally wants to minimize the number of switch elements in a path, even if this means using some longer signals runs.

Wire length and switches delays depend respectively on physical layout and cells library. The SPICE circuit simulator is used to obtain highly accurate delay estimation in each sub-path. A sub-path can be a wire, a switch or a set of connected wires and switches.

3.4 Benchmark circuits

In order to experiment and quantify the benefit of diverse architectures, we use Microelectronics Center of North Carolina (MCNC) designs [S.Yang, 1991]. As presented in table 3.2, these circuits cover various application types with several sizes (<10K 4-Luts), In/Out Pads number and congestion levels. We used also ava circuit [R.Tessier, 2005] [J.Pistorius et al., 2007] which is the largest circuit (∼15K 4-LUTs) containing only lookup-tables and flip-flops.
To get any benchmarks over 15K 4-LUTs, we need to be able to support black-boxes for hard blocks, e.g. memory. This is because designers cannot build a design larger than that, with no memories, DSP blocks, arithmetic, etc. In this work we are interested only in interconnect topology effects. Target architectures and tools do not support heterogeneous blocks; we use only LUTs with 4 inputs (4-LUT).
When we evaluate a specific interconnect topology, we tailor different architectures to each benchmark. For the same interconnect topology we select an architecture with an appropriate level of routability based on the benchmark congestion level (estimated by Rent parameter). For example in the case of Mesh architecture, VPR tool executes a binary search to determine the smallest architecture with the minimal channel width that can route a specific benchmark circuit. Architectures tailoring is interesting to explore different topologies and to check if they

can deal with different applications families. In the case of commercial FPGAs, the architecture is fixed independently of the targeted applications. Nevertheless, we can find different FPGA families proposed by one vendor to better fit specific applications and constraints. In our experimentation approach, we consider each circuit benchmark as representing a design family.

3.5 Architecture Example

Our tools platform was used to place and route circuits on a specific architecture CFPGA which where proposed and designed by CEA (Commisariat a l'Energie Atomique in France). In figure 3.9, we present CFPGA architecture having the following features:

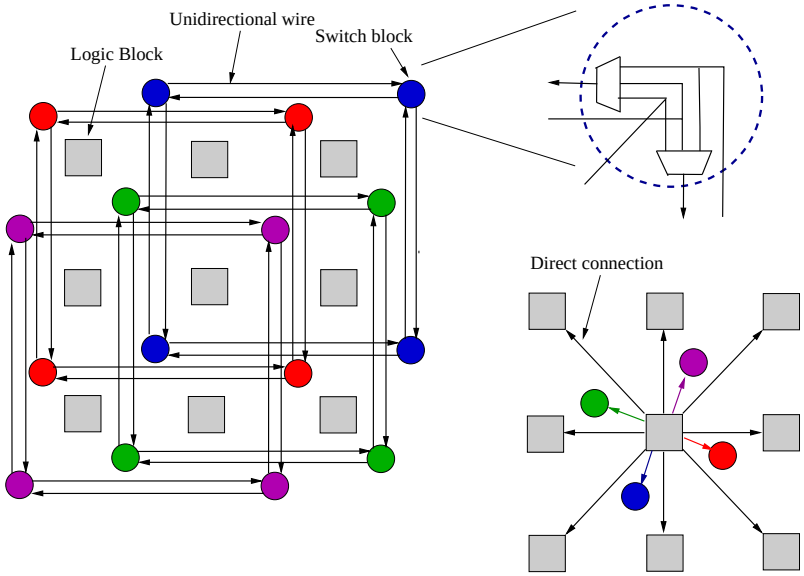


Figure 3.9: Single driver-based Mesh interconnect

- Mesh topology: Logic Blocks (LBs) and switch boxes are organized into rows and columns.
- Single driver routing architecture: There is only unidirectional wires driven by multiplexers,
- Wire length equal to 2: A wire in the global interconnect spans 2 LBs,
- Disjoint switch boxes: Each LB is surrounded by $4 \times W$ disjoint switch boxes, where W is the channel width. Each switch box corresponds to a full cross bar. A switch box has input signals coming from the adjacent switch boxes and LBs outputs. We can state that interconnect is composed of $4 \times W$ directional networks.

Design Name	4-LUTs	In Pads	Out Pads	Function
alu4	584	14	8	ALU
apex2	1878	39	3	
apex4	1262	9	19	
ava	14964	9	74	AVA Decoder
b9	61	41	21	Logic
bigkey	1707	263	179	Key Encryption
c2678	363	233	140	ALU and Control
c5315	725	178	123	ALU and Selector
c7552	881	207	108	ALU and Control
cc	33	21	20	Logic
clma	8383	61	82	Bus Interface
count	37	35	16	Counter
decod	32	5	16	Decoder
des	3235	256	245	Data Encryption
diffeq	1497	64	39	
dsip	1370	229	197	Encryption Circuit
elliptic	3604	131	114	
ex1010	4589	10	10	
ex5p	1064	8	63	
frisc	3556	20	116	
i4	110	192	6	Logic
i9	471	63	522	Logic
misex3	1397	14	14	
pcle	29	19	9	Logic
pcler8	40	25	19	Logic
pdc	4575	16	40	
s298	1931	4	6	PLD
s38417	6406	29	106	Logic
s38584	6447	39	304	Logic
seq	1750	41	35	
spla	3690	16	46	
tseng	1047	52	122	

Table 3.2: Benchmarks characteristics

- Local interconnect: In addition to the global interconnect, there are direct connections between each LB and its 8 adjacent LBs.

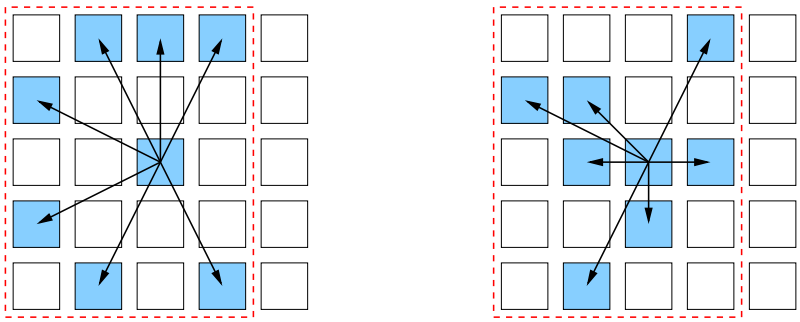


Figure 3.10: Placements cost evaluation based on two different objective functions

Since this architecture does not contain any hierarchy (no LBs clusters), we adapted the placement and routing tools only. We developed also an interactive user interface allowing to control different algorithms parameters and to display CAD flow steps. User can also modify placement manually.
To consider local interconnect in the placement phase, we use simulated annealing to optimize a specific objective function. In fact, with signals bounding box evaluation, we cannot consider direct connections between adjacent LBs. In figure 3.10, we show two different placements of the same signal terminals. Despite that in the second case we can take advantage of direct connections to route adjacent LBs, in both cases we obtain the same bounding box (bbx) cost value. This shows the inefficiency of signal bounding box objective to take into account direct connections. We propose an incremental objective function called $STAR$ which considers geometric proximity (wire length reduction) and direct connections. This objective is easy to compute and correlates well with routability and delay metrics. Algorithm 3.1 presents pseudo-code to compute cost function variation after a single instance move. We consider that 2 instances are adjacent if they are located at 2 neighboring LBs (see direct connections presented in figure 3.9). If we use $STAR$ function to evaluate placements presented in figure 3.10, we obtain a cost equal to 20 for the first placement and 9 for the second one. This is obvious since the second placement allows the router to take advantage of interconnect direct connections.
To route netlists, we model CFPGA routing resources with a direct graph where nodes correspond to LBs pins and wires and edges correspond to switches. We use the $Pathfinder$ algorithm to route signals using graph nodes and edges.
Placement and routing steps on CFPGA are illustrated by the following figures:

- Figure 3.11 shows a random placement of $pcler8$ circuit (LBs , In pads and Out Pads) on CFPGA matrix. LBs colored in gray are occupied by instances. White squares correspond to empty LBs. In/out Pads are located at the circuit boundary. Black lines correspond to nets.

- Figure 3.12 shows instances placement after simulated annealing process. The objective

$old\_cost = 0$;
$new\_cost = 0$;
foreach *instance* ins_i *connected to the instance to move* ins_m **do**
  if ins_i *is not adjacent to* ins_m **then**
    $old\_cost = old\_cost + \mid X(ins_m) - X(ins_i) \mid + \mid Y(ins_m) - Y(ins_i) \mid$;
  endif
endfch
Move instance ins;
foreach *instance* ins_i *connected to the instance moved* ins_m **do**
  if ins_i *is not adjacent to* ins **then**
    $new\_cost = new\_cost + \mid X(ins_m) - X(ins_i) \mid + \mid Y(ins_m) - Y(ins_i) \mid$;
  endif
endfch
$\Delta cost = old\_cost - new\_cost$;

Algorithm 3.1: pseudo-code to compute cost function variation after a single instance moving

function $STAR$ were improved by 60%.

- Figure 3.13 shows the routing resources of CFPGA architecture. In this case channels width is equal to 2 and each directional connecting network is plotted with a different color.

- Figure 3.14 shows routed signals using CFPGA resources.

- Figure 3.15 shows some routed signals connected to a specific LB (colored in black). Corresponding LBs drivers and receivers are colored respectively in red and blue. An LB output signal starts at the square center and an input signal stops at the square edge.

3.6 Conclusion

With this background, we can formulate design requirements for FPGA programmable interconnect as follows:

- Adequate flexibility: The network must be capable of implementing the interconnection topology required by the programmed logic design with acceptable delays.

- Interconnect depopulation: Interconnect is the major factor concerning area, delay and power consumption inefficiencies of FPGA compared to ASICs. It is a big challenge to provide an interconnect architecture with high flexibility and reduced routing resources. In [F.Li et al., 2005], authors show that utilization rate of interconnect switches is extremely low (about 12%). The remaining 88% of resources are necessary for flexibility and to deal with a large number of various designs.

- Efficient configuration memory utilization: Space required for configuration memory can account for a reasonable fraction of the total area. Configuration encodings can be tight and do not have to take up substantial area, compared to the wires and switches area.
- Balanced bisection bandwidth: Interconnect wiring takes space and, in some topologies may dominate the array size. The wiring topology should be chosen to balance interconnect bandwidth with array size and expected design interconnect requirement.
- Delays reduction: The delay through the routing network is the dominant delay in FPGA. Switching can be used to reduce fanout on a line by segmenting tracks, and large fanout can be used to reduce switching by making a signal always available in several places. Minimizing the interconnect delay, therefore, always requires technology dependent tradeoffs between the amount of switching and the length of wire runs.

In the following chapters we propose architectures taking advantages of both Mesh and Tree merits. The evolution of the architecture is driven progressively to be consistent with points formulated above.

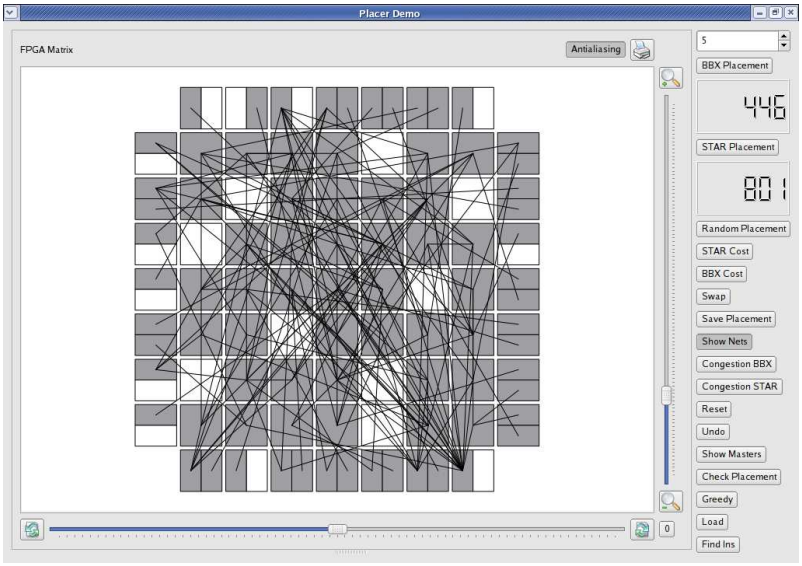


Figure 3.11: Random netlist placement on CFPGA architecture

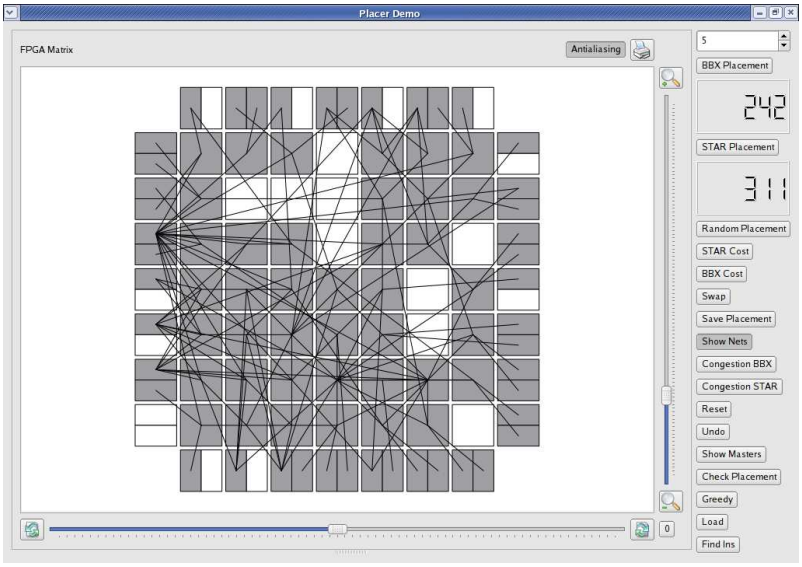


Figure 3.12: Optimized netlist placement on CFPGA architecture

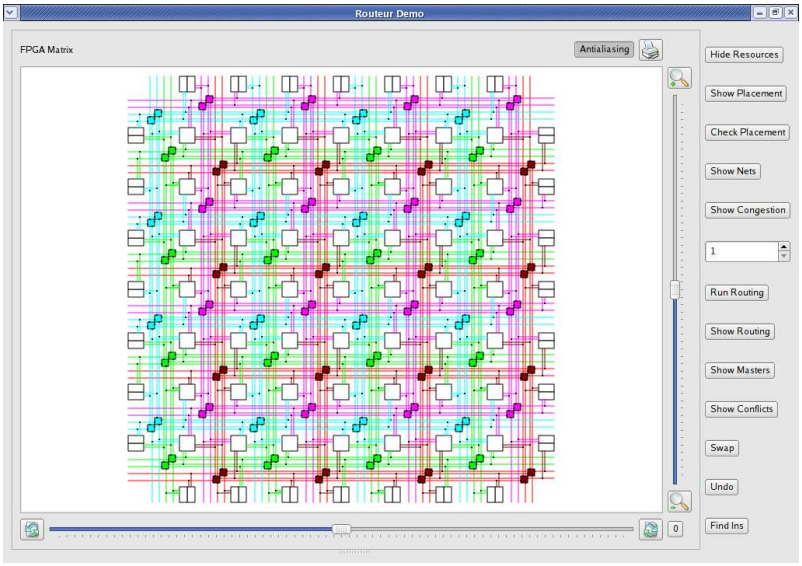


Figure 3.13: CFPGA architecture routing resources

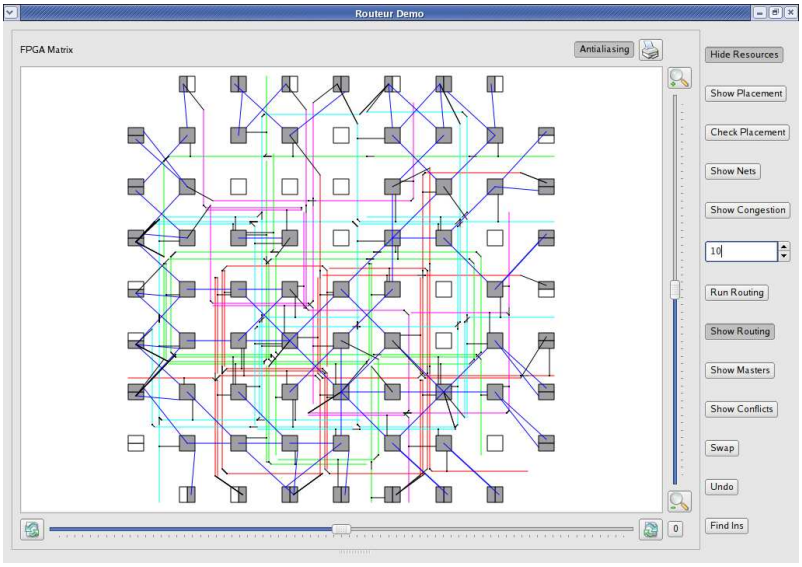


Figure 3.14: Routed netlist on CFPGA architecture

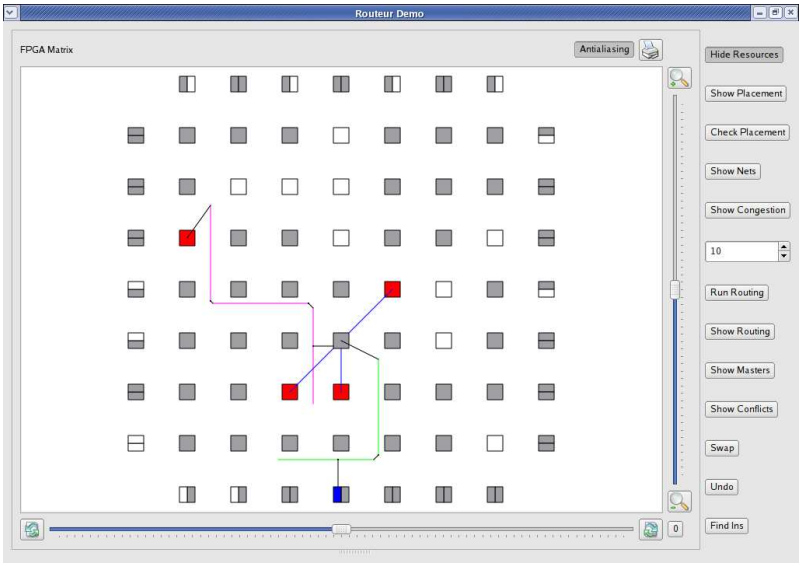


Figure 3.15: Routed signals connected to an LB

4

Tree-based FPGA with optimized cluster signals bandwidth

Design of large devices implies fundamental and efficient innovation in architecture to improve speed, density and software mapping time. Relying on industry experience with standard ASICs, we believe that partitioning and hierarchy become unavoidable for hardware and software developments. In fact most logic designs exhibit local connections, which implies a hierarchy in placement and routing of connections between logic blocks.
We propose a Tree-based FPGA architecture TFPGA which takes advantage of this feature to provide smaller routing delays and more predictable timing behavior. Routability and interconnect area depend on switch boxes topology and signals bandwidth (in/out signals per cluster). In TFPGA we use full crossbar switch boxes and we aim at exploiting the available flexibility to reduce signals bandwidth based on suitable partitioning approaches.
To reduce clusters signals bandwidth, we tested different partitioning strategies. We compare the resulting Tree-based architecture to the common Mesh-based architecture in terms of switches requirement.

4.1 Proposed Architecture

As illustrated in figure 4.1, in TFPGA (Tree-based FPGA), Logic blocks and routing resources are partitioned into a multilevel clustered structure. Each cluster contains sub-clusters and a switch box allowing to connect external signals to sub-clusters. In this first study of hierarchical topology, we consider the problem with a different stand point. All Tree based networks presented

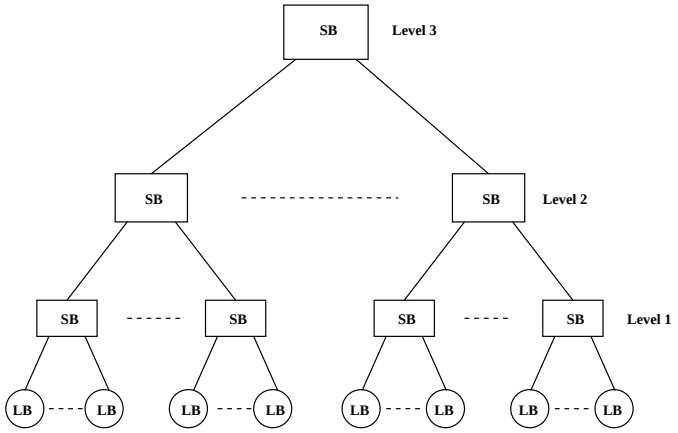


Figure 4.1: General Tree-based architecture

in the chapter 1 use bidirectional switches and wire tracks. This introduces considerable complication in both hardware network design and increases the load on routing tools [G.Lemieux et al., 2004]. In this architecture we use only single-driver unidirectional wires. As proposed in [A.DeHon, 1996], we build a fully hierarchical interconnect with inter-level signal bandwidth, growing according to Rent's Rule. We consider only unidirectional signal wires. Logic Blocks represent the Tree leaves. Let N the number of LBs in the architecture. Gates are recursively partitioned into k equally sized sets at each level of the hierarchy. The principal interconnect occurs at each node of convergence in the hierarchy (see figure 4.2). At level ℓ in the hierarchy, each node has a fan-in from the lower level equal to $k * n_{out}(\ell - 1)$ signals and a fan-in from the upper level equal to $n_{in}(\ell)$. Similarly, it has a fan-out of $k * n_{in}(\ell - 1)$ toward the leaves and $n_{out}(\ell)$ towards the root. At each level ℓ, we have $n_{LB}(\ell)$ LBs, $n_{in}(\ell)$ external inputs and $n_{out}(\ell)$ external outputs. We are interested to evaluate wiring and switching growth. According to the architecture hierarchy and the Rent's rule growth we have:

$$N_{LB}(\ell) = n^{\ell}$$

$$n_{in}(\ell) = c_{in}.k^{\ell.p}$$

$$n_{out}(\ell) = c_{out}.k^{\ell.p} \tag{4.1}$$

4.1.1 Wire growth model

First we consider how wiring resources grow in this structure. At each level ℓ of the hierarchy, each switching node has $n_{in}(\ell)$ inputs and $n_{out}(\ell)$ outputs. This makes the bisection width equal to $(c_{in} + c_{out})k^{\ell.p}$. Since $\forall \ell \in \{1, \ldots, \log_k(N)\} \quad k^{\ell.p} \leq N$, the bisection width is $O(N^p)$. For a 2-dimensional network layout this bisection width must cross out of the subarray through the perimeter. Thus the perimeter of each subarray is $O(N^p)$. The area of the subarrays will be proportional to the square of its perimeter, making: $A_{subarray} \propto N^{2p}$. The required area per logic

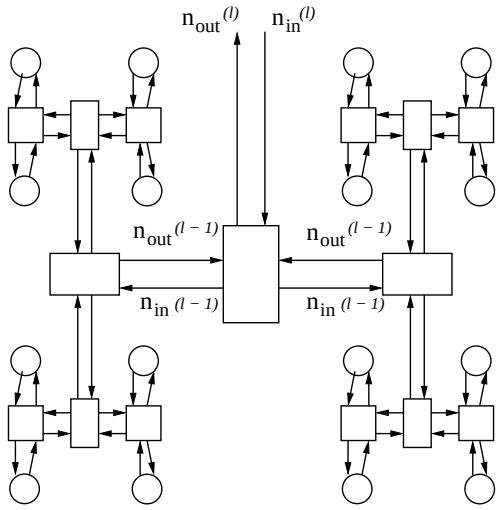


Figure 4.2: TFPGA: Tree based FPGA interconnect

block (LB) based on wiring constraints, then goes as:

$$A_{wiring}(LB) \propto N^{2p-1}$$

4.1.2 Switch growth model

We have $k+1$ distinct output directions from each node of convergence in the interconnect: k for the k leaves, plus one for the root. Allowing full connectivity within each Tree node, each one of the k leaves picks its n_{in} inputs from the $(k-1)*n_{out}$ outputs from its siblings and from the n_{in} inputs from the parent node. The n_{out} outputs of this node are selected from the $k*n_{out}$ outputs from all k subtrees converging to this point. Figure 4.3 shows this basic arrangement for $k=2$. Each one of the logical switching units is a fully-populated crossbar. At each level ℓ, the total switch number is:

$$\begin{aligned} N_{switch}(\ell) &= [k*((k-1)n_{out_{\ell-1}} + n_{in_\ell})*n_{in_{\ell-1}}] + [(k*n_{out_{\ell-1}})*n_{out_\ell})] && (4.2)\\ &= [k^p(c_{in}+c_{out}) + (k-1)c_{out}]kc_{in}k^{2p(l-1)} && (4.3)\end{aligned}$$

Dividing by the number of LBs supported at level ℓ, we can count the number of switches per LB at each level:

$$N_{switch}(\ell) = \frac{[k^p(c_{in}+c_{out}) + (k-1)c_{out}]kc_{in}k^{2p(\ell-1)}}{k^\ell}$$

Summing across all levels we obtain:

$$N_{switch}(LB) = [k^p(c_{in}+c_{out}) + (k-1)c_{out}] \times c_{in} \sum_{\ell=1}^{\log_k(N)} k^{(2p-1)(\ell-1)} \quad (4.4)$$

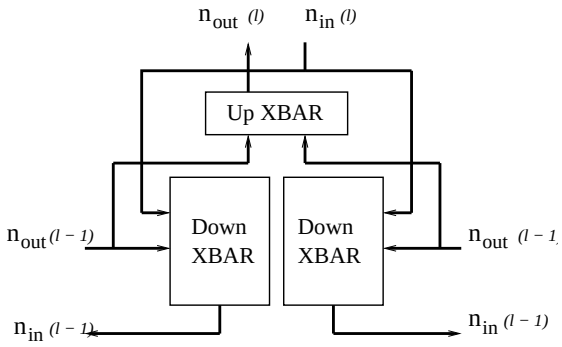


Figure 4.3: Switching node in 2-arity Hierarchical Interconnect [A.DeHon, 1996]

$$N_{switch}(LB) = \begin{cases} O(1) & \text{if } p < 0.5 \\ O(\log_k(N)) & \text{if } p = 0.5 \\ O\left(N^{2p-1}\right) & \text{if } p > 0.5 \end{cases} \tag{4.5}$$

Equation 4.5 shows that we have switching area per LB grows as $O(1)$, for $p < 0.5$, and $O\left(N^{2p-1}\right)$ for $p > 0.5$.

The large area of switches relative to wires is one of the reasons that we care about the number of switches required by the network. To reduce switches requirement we aim at reducing interconnect Rent's parameter. Architecture Rent's parameter is the minimum possible Rent's parameter of the architecture allowing routability achievement of a given netlist. In the proposed architecture, we use fully populated switch boxes (crossbar). There is exactly one switch associated with every possible input to output connection, so routing is trivial and guaranteed. Architecture Rent's parameter corresponds exactly to the partitioned netlist Rent's parameter. In [L.Hagen et al., 1994], authors showed that the resulting Rent's parameter is subject to the algorithm which generates the partitioning Tree. Thus in a TFPGA architecture switches requirement depends on the partitioning methodology.

To determine Rent's parameter of a netlist design we run a multilevel partitioning. In each hierarchical level, we determine the maximum number of inputs and outputs in all parts. Numbers of inputs and outputs of a cluster located at level ℓ of the Tree architecture are given by:

$$N_{in}(\ell) = \max_{part \in P(\ell)} N_{in}(part)$$

$$N_{out}(\ell) = \max_{part \in P(\ell)} N_{out}(part)$$

$P(\ell)$ is the set of parts at level ℓ.

4.2 Partitioning methodologies

Rent's parameter is an accurate indicator of wiring and switching requirements for a given partitioning hierarchy. In particular, in the case of two partitioning tools, the one with lower Rent's

parameter requires less switching and wirelength and corresponds to a denser final layout. Thus, in one part of this work various partitioning methods are compared, to identify the partitioning strategy leading to the optimal hierarchy. This yields a new methodology for comparing multilevel partitioning techniques efficiencies.

4.2.1 Top-down partitioning

Top-down approaches split a given netlist into smaller subclusters. This technique is based on global connectivity informations and leads to a good partitioning solution. The objective is to group logic blocks in order to reduce external communication. Since we have to reduce the maximum Input/output signals crossing each part, we use a multi-objective function which considers Maximum External Degree (MED) and the cut (see figure 3.4). We implement a solution similar to the direct multi-phase refinement presented in [N.Selvakkumaran and G.Karypis, 2006]. Thus, we first generate a partitioning solution with the cut as objective, then we apply a multi-phase multi-objective refinement with MED as the highest priority objective. We also add a constraint to respect clusters arity imposed by the architecture. Our approach is top down; first we construct clusters of the top level and then each cluster is partitioned into subclusters. This is done until the bottom of the hierarchy is reached.

4.2.2 Bottom-up partitioning

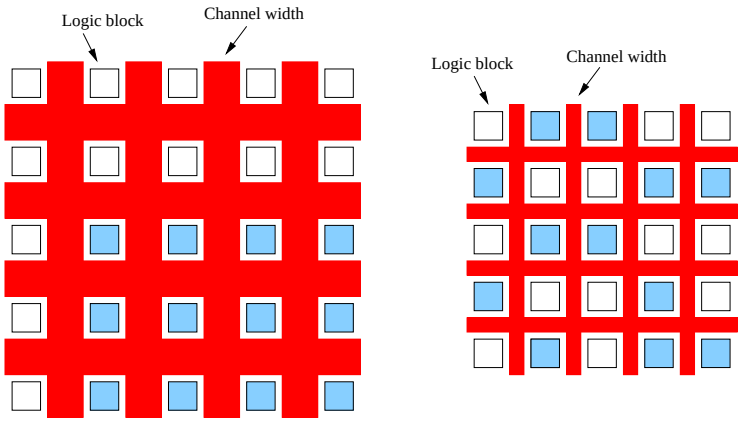


Figure 4.4: Congestion-aware placement

The size of the smallest TFPGA is penalized by the coarse granularity of the architecture. This means that in most cases the number of logic blocks slots present in the architecture is greater than the number of instances in the netlist to implement. This may have a good effect on congestion alleviating. In fact, DeHon [A.DeHon, 1999] showed that for hierarchical FPGAs, 100% logic use is not necessarily beneficial for overall device area minimization. The philosophy behind Logic and interconnect balancing is increasing logic utilization through efficient use of

the interconnect structure (which often accounts for $\sim 90\%$ of the total area in current FPGA families). The efficiency of this philosophy was shown in 2 different works:

- In [A.Sharma et al., 2005], authors proposed an efficient placement technique, called $Independence$, for routing-poor architectures. Routing-poor architectures attempt to increase interconnect utilization at the expense of logic utilization. As presented in figure 4.4, they modified the VPR's semi-perimeter based formulation (figure 4.4-a) and integrated an approch trying to spread congestion over all the area (figure 4.4-a). Specifically, they used $Pathfinder$ in the simulated annealing inner loop to maintain a fully routed solution at all times. In this way the required routing resources (channel width) are reduced considerably. Nevertheless, to create white spaces, instances are moved away and this might increase delays to connect logic blocks and consequently reduce speed performances. Authors [A.Sharma et al., 2005] do not give an accurate estimation of delay increase since they are only interested in area reduction.

- In [A.Singh and M.Marek-Sadowska, 2002], authors present a routability-driven clustering technique (iRac) for area and power reduction. The idea is to get a good device utilization by reducing clusters external signals at the cost of using more clusters. As illustrated in figure 4.5, when clusters are sparsely populated highly congested regions are eliminated and the required channel width is reduced.

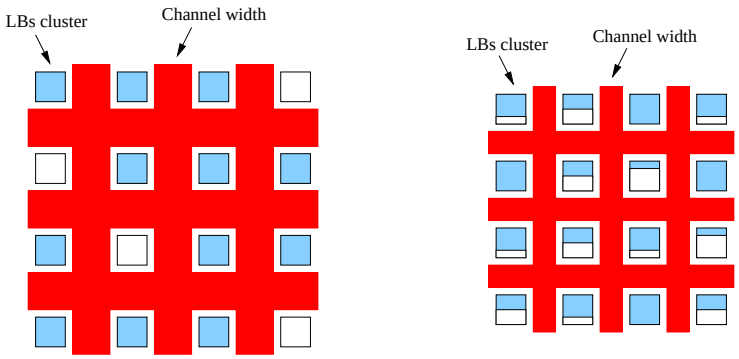


Figure 4.5: Congestion-aware clustering

In the following we present two different techniques to distribute instances over clusters. In both cases we use the same objective function proposed in iRac [A.Singh and M.Marek-Sadowska, 2002] to compute the attraction of each block to a cluster. iRac introduces a connectivity metric to the seed selection process of VPack [A.Marquart et al., 1999]. An unclustered block with most used inputs, and minimum connectivity is chosen as the cluster seed. Connectivity, defined in equation 4.6 measures the number of blocks appearing in the neighborhood of a given block. Gain computation for candidate blocks is based on common nets with the cluster under construction. High priority is given to absorbing 2-terminal nets, and edges of multi-terminal nets

with low terminals count. For 2 nets with same number of terminals, the net with more terminals already absorbed within the cluster is given higher weight.

$$Connectivity = \frac{Separation}{Degree^2} \tag{4.6}$$

$$Separation = \sum_{i \in Nets(B)} TerminalCount(i)$$

$$Degree = Number\ of\ Connected\ Nets$$

iRac achieves a large reduction in number of external nets, as it absorbs as many 2-terminal nets as possible, and tries to bring multi-terminal nets to an absorbable state.

LBs-limit strategy

Tessier [R.Tessier and H.Giza, 2000] showed that depopulation of clusters can result in reduced channels width in the case of Mesh architecture. The presented algorithm depopulates each cluster equally in order to reach an uniform distribution of empty LBs across the chip. In this approach, an attempt is made to spread the logic evenly across all clusters on the device. In this clustering algorithm, the possible number of LBs to be included in each cluster (N_{high}, N_{low}) are determined first. These numbers reflect the overall LB utilization of the device and are such that $N_{high} = N_{low} + 1$. After this step, the numbers C_{high} and C_{low} clusters that include respectively N_{high} and N_{low} LBs are determined. Clustering is then performed for both types of clusters.

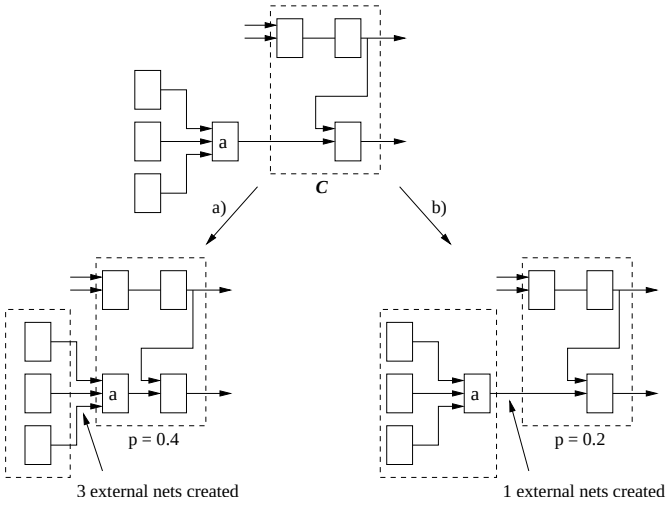


Figure 4.6: Reducing external routing demand [A.Singh and M.Marek-Sadowska, 2002]

Pins-limit strategy

iRac [A.Singh and M.Marek-Sadowska, 2002] clustering tool is very effective at reducing channel

width in the case of Mesh architecture. iRAC limits the number of inputs to each cluster using Rent's parameter, resulting in solutions that exhibit some depopulation. The aim of this technique is to alleviate routing congestion by absorbing as many nets into clusters as possible, and depopulating clusters according to Rent's rule in order to achieve spatial uniformity in the clustered netlist. In figure 4.6, we present an example of constructing clusters with size constraint k equal to 4 and pin constraint equal to 10 ($2k + 2$). Cluster C can absorb an additional LB. LB a has the highest gain and adding it to C does not violate either the architecture pin or cluster size constraint. Nevertheless, inserting a into cluster C creates 3 external nets and increases the cluster Rent's parameter p from 0.2 to 0.4. The interconnect-resource-aware clustering constraint adopted in [A.Singh and M.Marek-Sadowska, 2002], identifies this and ensures that the situation in figure 4.6-a) does not occur. Instead, LB a is chosen as seed for a new cluster, therefore adding a single external net as shown in figure 4.6-b).
We notice that pins-limit strategy is inefficient when applied in high levels. This is due essentially to the bottom-up and the greedy aspect to construct clusters with this technique. In fact clustering in high levels has a limited freedom and is penalized by choices made in lower levels. To deal with such a problem, we propose to create clusters in high levels without pins-limit enforcing. As presented in figure 4.7, once the multilevel clustering is achieved, we run a multilevel top-down refinement.

4.2.3 Multilevel refinement

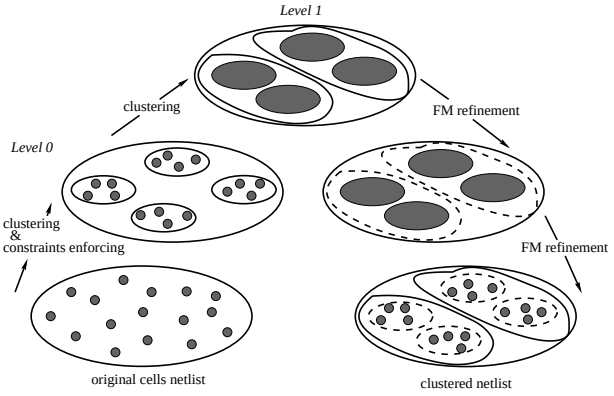


Figure 4.7: Multilevel clustering & refinement

After completing the clustering phase (with Pins-limit strategy), we obtain a tree of clusters each one containing k sub-clusters. During the refinement phase, cells are moved between clusters (parts) to optimize an objective function without violating the constraints imposed by the cluster size. In a level ℓ, cells are not allowed to move between all clusters, because this can decrease the quality of the solution obtained in the higher level. To prevent such unwanted effect, cells can only move between neighboring clusters. We call neighboring clusters, all clusters in a level belonging to the same super-cluster. Thus in every level, neighboring clusters are

isolated and form a subgraph. In figure 4.7 those subgraphs are represented by the continuous lines and partition by the dashed ones. A cell is allowed only to move across dashed lines. The objective function is specific to each subgraph and corresponds to the Maximum External Degree (MED) of all parts belonging to the same subgraph. An FM algorithm [C.M.Fiduccia and R.M.Mattheyeses, 1982] is applied to each subgraph to optimize the local objective function.
The complexity of our k-way refinement is reduced since we apply it successively for each subgraph (in each subgraph there are small number of parts: Arity of the architecture) and only for the highest levels (where bottom-up pins-limit strategy fails).

4.3 Experimental Results

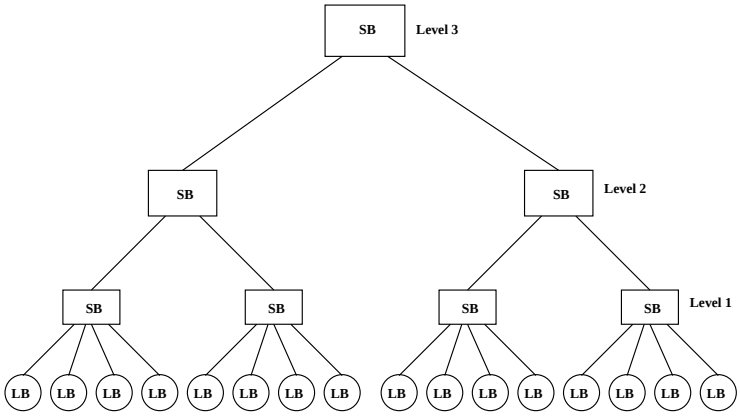


Figure 4.8: 4x2x2 Tree architecture: clusters arity definition at every level

The aim of this section is to compare different partitioning techniques. Our comparison metric is reducing clusters signals bandwidth. Once we identify the approach requiring the smallest architecture, we compare the TFPGA required switches to the Mesh-based FPGA.

4.3.1 Partitioning methodologies comparison

In the preceding approaches, the main classes of partitioning are top-down and the bottom-up. The following algorithms were used in our experiments:

- TD: Top-Down partitioning.
- BU-B : Bottom-Up clustering with LB-limit.
- BU-P : Bottom-Up clustering with Pins-limit.
- BU-P-R: A BU-P followed by the refinement phase.

Benchmark				TD		BU-P		BU-B		BU-P-R	
Name	LBs	Arch	level	Rent	T(s)	Rent	T(s)	Rent	T(s)	Rent	T(s)
seq	1750	8x8x8x8	1	0.828	27.3	0.773	20	0.845	12	0.773	27
			2	0.686		0.727		0.743		0.712	
			3	0.605		0.667		0.651		0.626	
apex2	1878	8x8x8x8	1	0.877	30	0.712	20	0.877	15	0.712	37
			2	0.722		0.773		0.770		0.722	
			3	0.602		0.674		0.684		0.626	
s298	1931	8x8x8x8	1	0.828	43	0.733	16.3	0.828	13	0.733	50
			2	0.517		0.627		0.631		0.597	
			3	0.341		0.588		0.579		0.480	
frisc	3556	8x8x8x8	1	0.877	65.5	0.733	44	0.810	38	0.733	75
			2	0.669		0.732		0.725		0.681	
			3	0.605		0.674		0.690		0.626	
elliptic	3604	8x8x8x8	1	0.828	71	0.712	42	0.828	35	0.712	80
			2	0.624		0.702		0.741		0.687	
			3	0.626		0.732		0.669		0.712	
spla	3690	8x8x8x8	1	0.861	75.4	0.733	55	0.845	40	0.733	84
			2	0.745		0.790		0.783		0.747	
			3	0.629		0.667		0.709		0.667	

Table 4.1: Rent parameter partitioning results

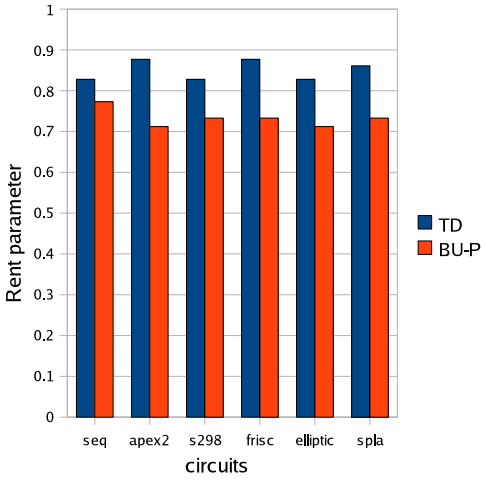


Figure 4.9: Results for partitioning Rent's parameters at level 1

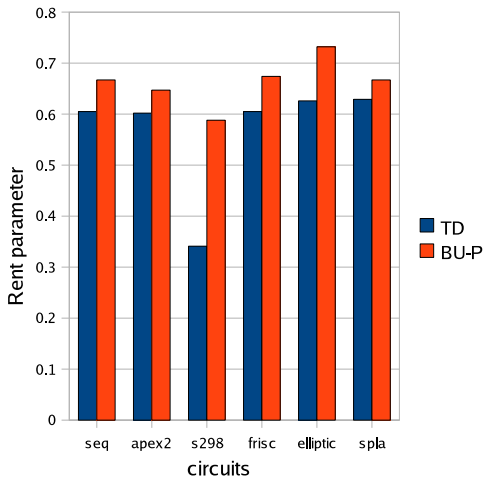


Figure 4.10: Results for partitioning Rent's parameters at level 3

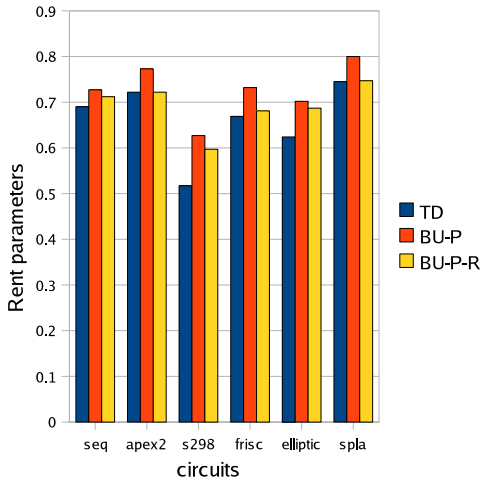


Figure 4.11: Results for partitioning Rent's parameters at level 2

We consider an architecture with cluster size equal to 8 in each level. The number of levels is determined to match the number of instances in the netlist. As shown in table 4.1, the third column presents the number of levels and clusters arity. For example in figure 4.8, we show an architecture 4x2x2. This means that it has 3 levels of hierarchy; clusters in the first one have a 4-arity, clusters in the second one have a 2-arity and cluster in the third one has a 2-arity.
The experiments were performed as follows. Each partitioning algorithm was used to construct a partitioning hierarchy for the circuit via recursive partitioning of this circuit and its sub-partitions. We have used some of the MCNC benchmark circuits with various logic sizes. At each partitioning step, we noted the number of external pins for each sub-partition as explained previously. In order to correlate the experimental data to Rent's rule, we reformulate the relationship $w_\ell = c(k^\ell)^p$ as:

$$p = \frac{log(w_\ell) - log(c)}{log(k^\ell)}$$

(ℓ is the corresponding level). As shown in table 4.1, for each level we have an associated Rent's parameter.
As presented in table 4.1, for the bottom-up clustering approaches we notice that pins-limit strategy BU-P is more efficient than LBs-limit strategy BU-B. This seems obvious since the aim of the first strategy is to reduce the number of external pins of every constructed cluster.
Figure 4.9 and figure 4.10 show that the TD method is the most efficient to reduce Rent's parameter in the highest level. This is due to the fact that the top down approach starts by partitioning instances between the highest level parts. Conversely, BU-P approach leads to better results in reducing the lowest level parts degrees. Thus BU-P approach is more efficient to reduce Rent's parameter in the lowest level; but with this technique, we obtain a poor solution when we construct the highest levels (level 1 and level 2). This is due to the inefficiency of pins-limiting in this stage. As shown in figure 4.11 the solution can be improved if we run a top-down refinement phase. With BU-P-R we obtained a good Rent's parameter in all levels (lowest and highest ones). Nevertheless, results obtained in level 2 by TD method are better than the ones obtained by BU-P-R approach. Thus, we conclude that TD approach is the best partitioning technique since it provides a good tradeoff between high and low levels Rent's parameters reduction.

4.3.2 Architectures comparison

We compared the switches requirement in TFPGA to a Mesh-based architecture. Mesh architecture is similar to the RFPGA architecture described in chapter 3. It is composed of clusters and has an uniform routing network with single-length segments and a subset switch box. Each cluster contains 8 4-LUTs. The number of inputs in each cluster is 18 and the number of outputs is 8. We use T-Vpack to construct clusters and the channel minimizing VPR 4.3 to place and route the obtained netlists. We vary the IO_{ratio} to achieve the optimal array size. VPR determines the optimal size as well as the optimal channel width to place and route each benchmark.
From table 4.2, we notice that the average number of needed switches in hierarchical TFPGA is about 3 times greater than in Mesh architecture. This is due essentially to the fully populated

Benchmark		Mesh				TFPGA		
Name	LBs	CLBs	W	IO ratio	Switches x10^3	Arch	Occup-ancy%	Switches x10^3
seq	1750	242	41	2	410	8x8x8x8	42	1627
apex2	1878	263	40	1	445	8x8x8x8	45	1854
s298	1931	305	30	1	374	8x8x8x8	47	699
frisc	3556	737	54	2	1633	8x8x8x8	86	3643
elliptic	3604	527	46	3	950	8x8x8x8	87	3343
spla	3690	750	56	2	1956	8x8x8x8	90	4269
average	2734	470	44	2	961		66	2573

Table 4.2: Switches comparison between Mesh-based FPGA architecture and TFPGA

crossbar.

4.4 Conclusion

We introduced the notion of architecture Rent's parameter defined as the lowest Rent's parameter achievable by any partitioning method (since we considered fully populated switch boxes). Our results indicate that a combination between a multilevel bottom-up clustering and a top-down refinement generates partitioning hierarchies with reduced Rent's parameters. Nevertheless, a top-down partitioning approach combining cut and MED objectives offers the best trade-off between high and low levels signals bandwidths optimization. The aim of this study was to find the best partitioning method to obtain the smallest TFPGA area. Despite our effort we found that with a fully-populated crossbar, TFPGA cannot be denser than Mesh-based FPGAs. Thus to make TFPGA more competitive, we must use depopulated routing interconnect. The question is how to depopulate the hierarchical interconnect and keep a good routability? In the following chapters we will try to give an answer and to present the effect of interconnect depopulation on the architecture Rent's parameter.

5

Tree-based interconnect with depopulated switch boxes

To improve Tree-based interconnect density we must sparsely populate switch boxes (Tree nodes). In the sequel we propose a Butterfly Fat-Tree interconnect topology called MFPGA (Multilevel FPGA). Switch boxes depopulation is compensated by routing predictability and a large signals bandwidth (Rent's growth parameter $p = 1$). We start by presenting architecture interconnect topology. Next, based on Rent's parameter we evaluate analytically switches and wiring requirement. Then, we present placement and routing approaches considering interconnect predictability. Finally, we compare area and performance efficiency of MFPGA to Mesh-based architecture.

5.1 MFPGA routing interconnect

As illustrated in figure 5.1, MFPGA contains N LUT-based logic blocks (LBs) and two unidirectional connecting networks:

- The downward network is inspired from SPIN [P.Guerrier and A.Greiner, 2000]. It is based on the Butterfly Fat-Tree (BFT) style interconnect [C.Leiserson, 1985] with linear populated switch boxes and unidirectional wires. Tree leaves correspond to logic blocks.
- The upward network connects logic blocks outputs and input pads to the various levels of the Tree.

Each logic block contains one Look-Up-Table (with c_{in} inputs and $c_{out} = 1$ output), followed by a bypass Flip-Flop. Like TFPGA architecture Logic Blocks (LBs) are grouped into k sized

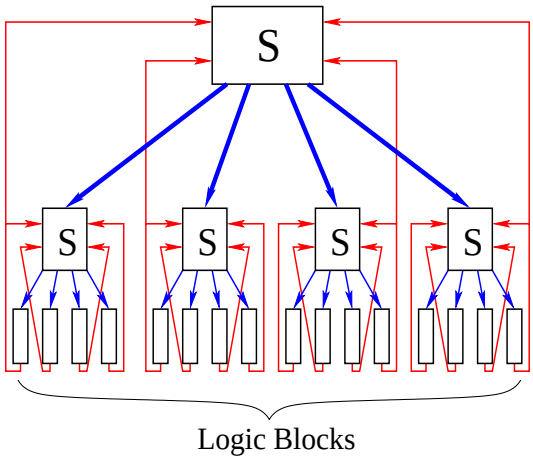


Figure 5.1: MFPGA interconnect

clusters and interconnect is organized into levels. Let $nb\ell$ denote the number of levels of a given Tree containing N leaves ($nb\ell = log_k(N)$). In each level ℓ we have $\frac{N}{k^\ell}$ clusters; C is the set of clusters in all levels. A cluster with index c belonging to level ℓ is noted by $cluster(\ell, c)$. A cluster switch block is divided into separated Mini Switch Boxes (MSBs). Each MSB corresponds to a full crossbar. Each $cluster(\ell, c)$ where $\ell \geq 1$ contains a set of inputs $N_{in}(\ell)$, a set of outputs $N_{out}(\ell)$, a set of MSBs and k sub-clusters. Sub-clusters of $cluster(\ell, c)$ are $cluster(\ell - 1, k.c + i)$ where $i \in \{0, 1, 2, .., k - 1\}$. k is called $cluster(\ell, c)$ arity. Let $nbMSB(\ell)$ the MSBs number in a cluster in level ℓ. MSB with index m belonging to $cluster(\ell, c)$ is denoted $MSB(\ell, c, m)$. Each MSB contains k inputs driven by the upper level and 1 feedback coming from a leaf output pin. Each cluster in level 0 is denoted $cluster(0, c)$ or $leafcluster(c)$ and corresponds to the Logic Block (LB) and contains c_{in} inputs, 1 output, no MSBs and no sub-cluster. Each $cluster(\ell, c)$ where $\ell < nb\ell - 1$ has an owner in level ℓ', where $\ell' > \ell$, denoted $cluster(\ell', c \div k^{(\ell' - \ell)})$. We define for each $cluster(\ell, c)$ a position inside its owner in level $\ell + 1$ (direct owner) by the following function:

$$\begin{array}{rccl} pos: & C & \longrightarrow & \{0, 1, 2, .., k - 1\} \\ & cluster(\ell, c) & \longmapsto & c \mod k \end{array}$$

2 clusters belonging to level ℓ and with the same owner at level $\ell + 1$ have 2 different positions. To get the cluster owner in level ℓ' of $cluster(\ell, c)$ $(\ell < \ell' \leq nb\ell - 1)$ we define the function:

$$\begin{array}{rccl} owner: & C \times \mathbb{N} & \longrightarrow & C \\ & (cluster(\ell, c), \ell') & \longmapsto & cluster(\ell', c \div k^{\ell' - \ell}) \end{array}$$

5.1.1 Downward Network

Figure 5.2 shows a sparse downward network based on unidirectional MSBs. The downward interconnect topology is similar to the butterfly fat tree. Each MSB of a $cluster(\ell, c)$ where $\ell > 1$

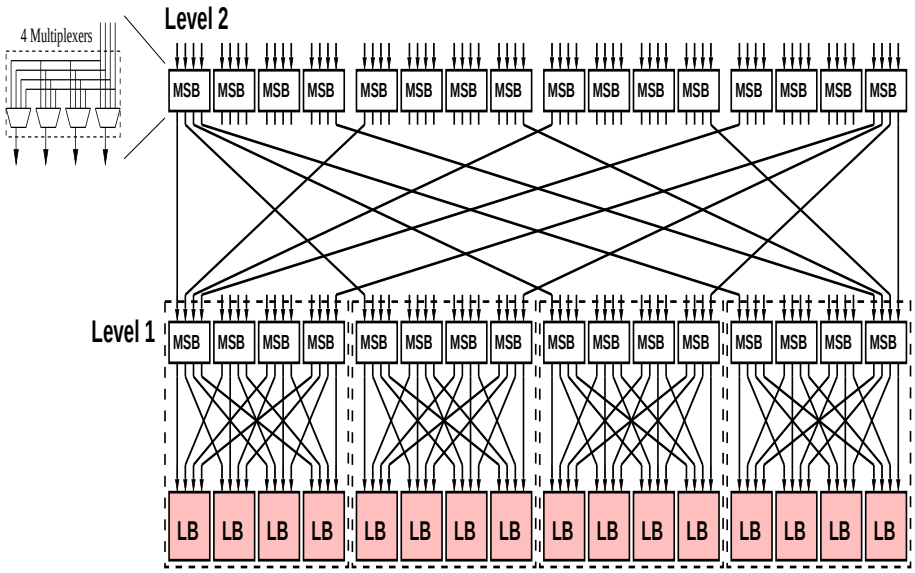


Figure 5.2: Downward Network

is connected to each sub-cluster through one and only one input pin. Thus, the MSBs number in a cluster situated in level ℓ is equal to the input number of a cluster situated in level $\ell - 1$: $nbMSB(\ell) = N_{in}(l-1)$.
We name $MSB(\ell', c', m')$ as the successor of an $MSB(\ell, c, m)$ where $0 < \ell' < \ell$ if there is a downward directed path from $MSB(\ell, c, m)$ to $MSB(\ell', c', m')$. The path between an MSB and its successor is unique. We define the function:

$$\begin{array}{rccl} Mod_\ell : \mathbb{N} & \longrightarrow & \mathbb{N} & \\ m & \longmapsto & m & \mod nbMSB(\ell) \end{array}$$

Thus each $MSB(\ell, c, m)$ has a successor in each sub-cluster belonging to level ℓ' $MSB(\ell', c', m')$ where $0 < \ell' < \ell$, with:

$$m' = Mod_{\ell'} \circ \cdots \circ Mod_{\ell-1}(m) \tag{5.1}$$

5.1.2 Upward Network

We propose to connect the output signals of leaf clusters to specific MSBs of upper levels. Thus for each logic block (LB) output, we define a list of feedbacks. Each one enables the LB output to reach one and only one MSB in a particular level. Each MSB is reached by one and only one Logic Block output. This means that the number of MSBs in each level is equal to the number of the Tree leaves (LBs). Since in each level we have $\frac{N}{k^\ell}$ clusters we obtain:

$$\frac{N}{k^\ell} \times NbMSB(\ell) = N$$

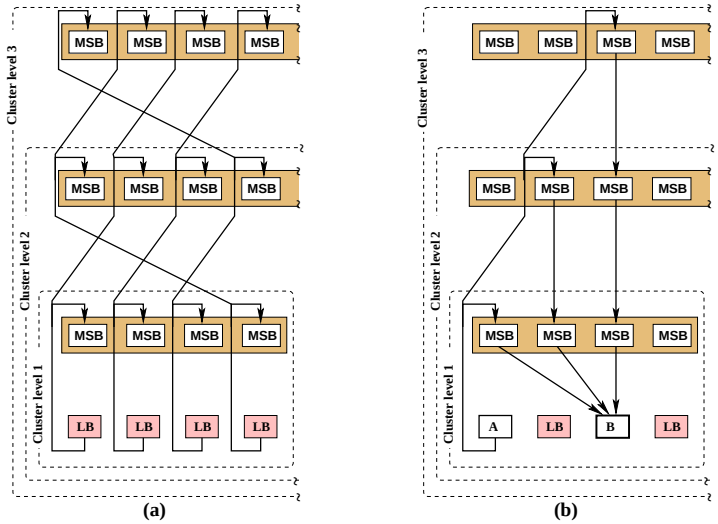


Figure 5.3: Upward Network

Since $NbMSB(\ell) = N_{in}(l-1)$, we obtain, according to Rent's rule:

$$c_{in}.k^{(\ell-1)p} = k^{\ell} \tag{5.2}$$

Equation 5.2 is verified if $k = c_{in}$ and $p = 1$, we call this the predictability condition. This condition is penalizing in terms of area but ensures interconnect predictability construction which is exploited by the placement and routing tools. Routing predictability is an interesting property that can be exploited in the placement phase to enhance congested netlists routing.
The way feedbacks are distributed has an important impact on the structure routability. Connecting an output of a leaf cluster to MSBs with different indexes increases the number of paths from a source to a destination. This specific distribution is described in figure 5.3-(a). Figure 5.3-(b) shows how the cluster leaf 'A' output can reach the cluster leaf 'B' inputs using different paths. Each leaf cluster $cluster(0,c)$ is connected to one and only one $MSB(\ell, c', m)$ in level $\ell > 0$. c' is the index of the owner of cluster(0,c) in level ℓ: $c' = c \div k^{\ell}$; m is given by:

$$m = (pos(cluster(0,c)) + \ell - 1)\ modulo\ (k) + \sum_{\jmath=1}^{\ell-1} pos(owner(cluster(0,c),\jmath)) \times nbMSB(\jmath) \tag{5.3}$$

5.1.3 Connection with outside

As shown in figure 5.4, output pads are clustered with the logic blocks at $level0$. The number of output pads per cluster can be varied to obtain the best design fit. We use a local interconnect between the logic block outputs and the output pads. Input pads are connected directly to MSBs of the highest level. In this way each input pad can reach all logic blocks. As presented in figure 5.4, input pads feedbacks distribution is similar to LB outputs one.

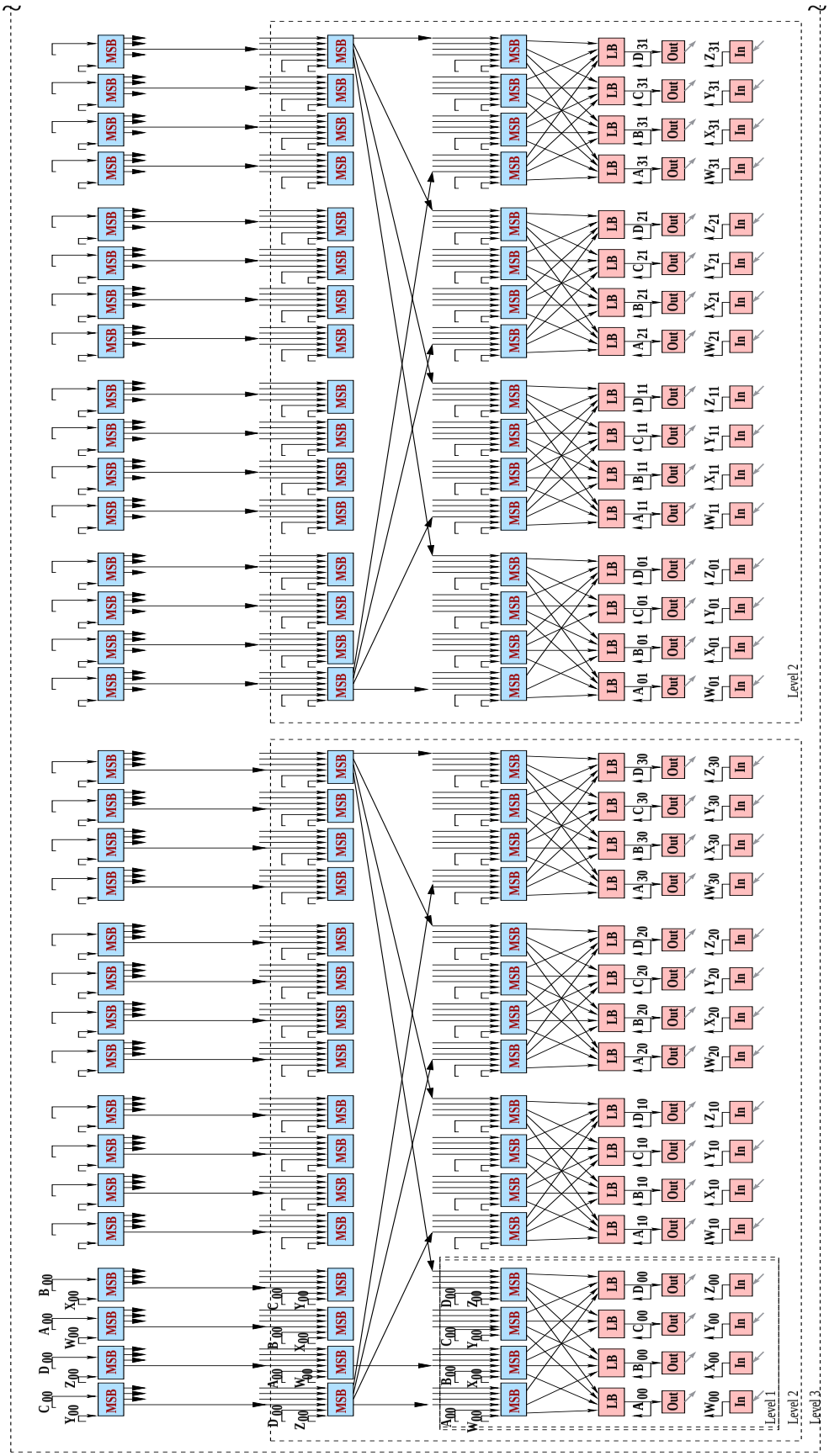


Figure 5.4: 3-level MFPGA structure with $k = 4$ and $p = 1$

5.1.4 Rent's Rule based MFPGA model

As we parametrize MFPGA architecture we can now evaluate wires and switches requirement. Consider a k-arity MFPGA as depicted in figure 5.4 with N Logic Blocks and whose wire growth follows Rent's Rule. Each Logic Block has c_{in} inputs and c_{out} outputs.

Switches requirement:

A cluster located at level ℓ contains $N_{in}(\ell-1)$ MSBs with k outputs and $\frac{N_{in}(\ell)+kN_{out}(\ell-1)}{N_{in}(\ell-1)}$ inputs. Since MSBs are full crossbar devices, we have $k(N_{in}(\ell)+kN_{out}(\ell-1))$ switches in the switch box of a level ℓ cluster. As we have $\frac{N}{k^\ell}$ clusters at level ℓ, we have a total number of switches equal to:

$$\sum_{\ell=1}^{\log_k(N)} kN\frac{N_{in}(\ell)+kN_{out}(\ell-1)}{k^\ell}$$

Based on Rent's rule, we have $N_{in}(\ell)=c_{in}k^{\ell.p}$ and $N_{out}(\ell)=c_{out}k^{\ell.p}$. If we take into account the predictability condition (section 5.1.2 ($c_{in}=k$ and $p=1$), we get $N_{in}(\ell)=k^{\ell+1}$ and $N_{out}(\ell)=k^\ell$. The number of switches per Logic Block is :

$$N_{switch} = \sum_{\ell=1}^{\log_k(N)} (k^2+k)$$

$$N_{switch} = (k^2+k)\log_k(N)$$

$$N_{switch} = O\left(\log_k(N)\right) \tag{5.4}$$

It was established in [A.DeHon, 1999] that in the Mesh architecture, switches per logic block grow as:

$$N_{switch}(LB) = O(N^{p-0.5}) \tag{5.5}$$

Equations (5.4) and (5.5) show that in MFPGA switches requirements grow more slowly than in Mesh architecture. Mesh switches requirement depends largely on Rent's parameter p value:

- for $p=0.65$, if $N > 15\times10^6$, Mesh becomes more penalizing than Tree in terms of switches requirement.

- for $p=0.7$, if $N > 17\times10^3$, Mesh becomes more penalizing than Tree in terms of switches requirement.

- for $p=0.78$, if $N > 2$, Mesh becomes more penalizing than Tree in terms of switches requirement. This is the case of APEX architecture discussed in [J.Pistorius and M.Hutton, 2003].

These results are encouraging for the construction of very broad MFPGA structures. But this does not mean that MFPGA topology is more efficient than Mesh-based architecture since they do not have the same routability. The best way to check this is to launch experimental work and compare the area results using MFPGA and the Mesh-based FPGA.

Wires requirement:

At each level ℓ of the hierarchy, each switching node (cluster) has $n_{in}(\ell)$ inputs and $n_{out}(\ell)$ outputs. This makes the bisection width equal to $(c_{in}+c_{out})k^{\ell}$. Since $\forall \ell \in \{1,\ldots,\log_k(N)\} \quad k^{\ell} \leq N$, the bisection width is $O(N)$. For a 2-dimensional network layout this bisection width must cross out the subarray through the perimeter. Thus the perimeter of each subarray is $O(N)$. The area of the subarrays will be proportional to the square of their perimeter, making: $A_{subarray} \propto N^2$. The required area per logic block (LB) based on wiring constraints is given by:

$$A_{LB} \propto N \tag{5.6}$$

Equations 5.6 and 5.4 show that wiring is the dominant resource constraining LB area in the proposed MFPGA architecture.

5.2 MFPGA placement

The MFPGA placement problem can be stated as assigning to each netlist cell a logic block (leaf) in the MFPGA architecture. The way how cells are distributed has an important impact on routability. In fact after cells placement, the router tries to find a path to connect a source LB (cluster leaf) to its destinations LBs (cluster leaf) using architecture resources. Thanks to the interconnect predictability provided by the MFPGA architecture we can introduce, in the placement phase, some conditions to limit later conflicts in the routing phase.

5.2.1 Conflict conditions

Definition 1. There is a resource conflict in level ℓ if 2 leaf clusters (or more), such as $cluster(0,c)$ and $cluster(0,c')$ reach a $cluster(\ell,c'')$ on the same pin pi.

Property 1. The owner in level $\ell+1$ of $cluster(\ell,c'')$ has one and only one $MSB(\ell+1, c'' \div k, m)$ which can reach this $cluster(\ell,c'')$ on pin pi.

Definition 2. Referring to the previous property, the definition 1 can be stated as:
There is a resource conflict problem in level ℓ if 2 leaf clusters (or more) such as $cluster(0,c)$ and $cluster(0,c')$ attempt to reach a $cluster(\ell,c'')$ and have both already reached its owner $cluster(\ell+1, c'' \div k)$ at the same $MSB(\ell+1, c'' \div k, m)$.

From definition 2, we can detect a resource conflict by finding 2 leaf clusters reaching the owner cluster of a common destination in the same MSB. We consider that $cluster(0,c)$ reaches $cluster(\ell,c'')$ in $MSB(\ell,c'',m)$ using the level ℓ_{up}, and that $cluster(0,c')$ reaches the same cluster destination in $MSB(\ell,c'',m')$ using level ℓ'_{up}. From equation (5.3) and (5.1) in this order we get:

$$\begin{cases} m = (pos(cluster(0,c)) + \ell_{up} - 1)\ modulo\ (k) + \sum_{\jmath=1}^{\ell-1} pos(owner(cluster(0,c),\jmath)) \times nbMSB(\jmath) \\ m' = (pos(cluster(0,c')) + \ell'_{up} - 1)\ modulo\ (k) + \sum_{\jmath=1}^{\ell-1} pos(owner(cluster(0,c'),\jmath)) \times nbMSB(\jmath) \end{cases} \tag{5.7}$$

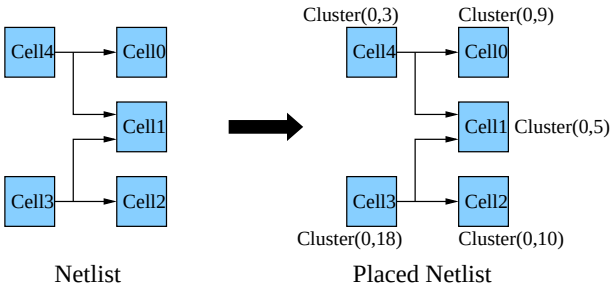


Figure 5.5: Netlist to route

thus,

$$\begin{cases} m = m' \\ \Updownarrow \\ (pos(cluster(0,c)) + \ell_{up})\ modulo\ (k) = (pos(cluster(0,c')) + \ell'_{up})\ modulo\ (k) \\ pos(owner(cluster(0,c),\jmath)) = pos(owner(cluster(0,c'),\jmath)) \\ \forall \jmath \in \{1,\ldots,\ell-1\} \end{cases} \tag{5.8}$$

$Proof$: Equations 5.7 correspond to the decomposition of m and m' in the base k because:
- $0 < (pos(cluster(0,c)) + \ell_{up})\ modulo\ (k) < k$
- $0 < pos(owner(cluster(0,c),\jmath)) < k\ \forall j, c$
- $nbMSB(\jmath) = k^{\jmath}$

Therefore we obtain results presented in equation 5.8.

Lemma 1. We say that 2 leaves $cluster(0,c)$ and $cluster(0,c')$ are in conflict to drive a common destination $cluster(\ell,c'')$ **if and only if:**

$$\begin{cases} pos(cluster(0,c)) - pos(cluster(0,c')) = (\ell'_{up} - \ell_{up})\ modulo\ (k) \\ pos(owner(cluster(0,c),\jmath)) = pos(owner(cluster(0,c'),\jmath)) \\ \forall \jmath \in \{1,\ldots,\ell\} \end{cases}$$

Where, ℓ_{up} (ℓ'_{up}) is the level allowing $cluster(0,c)$ ($cluster(0,c')$) to reach $cluster(\ell,c'')$.

5.2.2 Placement example

We propose to apply the obtained conflict condition on a practical example. We consider specific architecture with the following parameters: clusters arity $k = 4$, Rent's parameter $p = 1$ and Logic Block inputs number $c_{in} = 4$. We refer to the netlist presented in figure 5.5. We propose to place cells as shown in figure 5.6. In this example $cell0$, $cell1$, $cell2$, $cell3$ and $cell4$ are placed respectively in $cluster(0,9)$, $cluster(0,5)$, $cluster(0,10)$, $cluster(0,18)$, and $cluster(0,3)$.
Referring to the clustered netlist, $cluster(0,18)$ and $cluster(0,3)$ have 3 common destinations: $cluster(0,5)$ at level 0 and $cluster(1,2)$ and $cluster(1,1)$ at level 1. Referring to $lemma1$, we

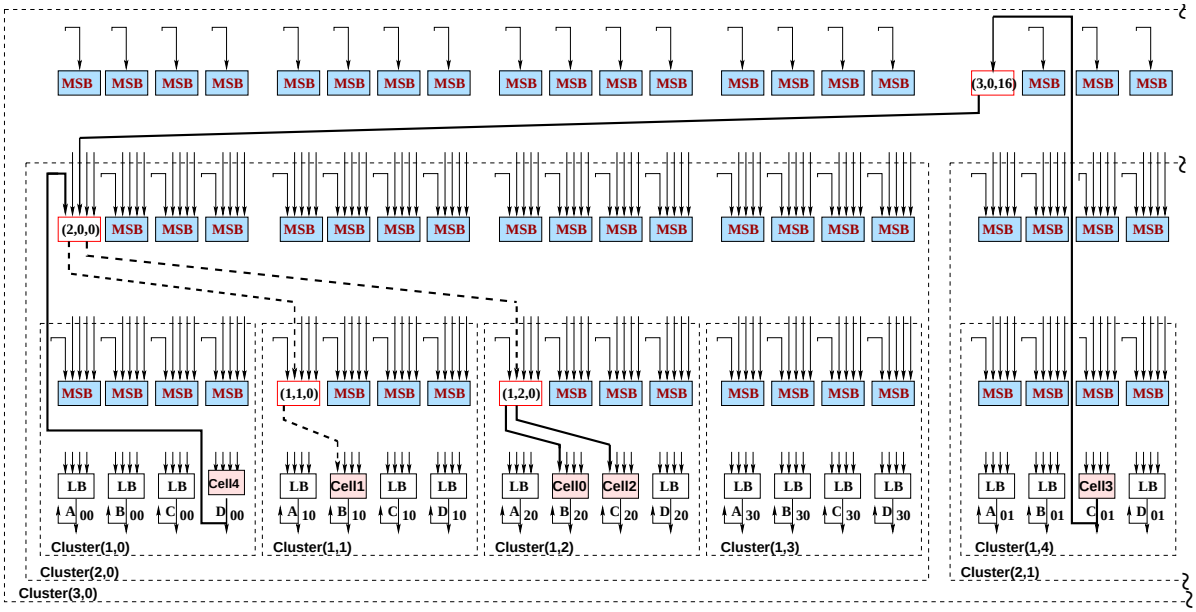


Figure 5.6: Detailed placement example

check whether there is a resource conflict to connect both sources to the 3 destinations. We propose to use the lowest possible level to connect a source to its destinations. To reach $cluster(0,5)$, $cluster(0,3)$ must go up to level 2 ($\ell_{up} = 2$) and $cluster(0,18)$ to level 3 ($\ell'_{up} = 3$).
Since $pos(cluster(0,3)) = 3$ and $pos(cluster(0,18)) = 2$,
we get $pos(cluster(0,3)) - pos(cluster(0,18)) = \ell'_{up} - \ell_{up}$. Thus the condition of $lemma1$ is satisfied and there is a resource conflict in level 0 to reach $cluster(0,5)$. The second common destination is $cluster(1,2)$. To reach this destination, $cluster(0,3)$ must be connected up to level 2 ($\ell_{up} = 2$) and $cluster(0,18)$ to level 3 ($\ell'_{up} = 3$). Since $pos(cluster(0,3)) = 3$ and $pos(cluster(0,18)) = 2$, we get $pos(cluster(0,3)) - pos(cluster(0,18)) = \ell'_{up} - \ell_{up}$. Thus the first condition in $lemma1$ is satisfied. We check now the second condition of $lemma1$ since destination $cluster(1,2)$ belongs to level 1 ($\ell > 0$). We have $owner(cluster(0,3),1) = cluster(1,0)$ and $owner(cluster(0,18),1) = cluster(1,4)$. Since $pos(cluster(1,0)) = pos(cluster(1,4)) = 0$ the second condition of $lemma1$ is verified too. Thus there is a resource conflict at level 1 to connect $cluster(0,3)$ and $cluster(0,18)$ to $cluster(1,2)$.
We have the same problem with the third common destination $cluster(1,1)$. The routing solution of the placed netlist using the lowest levels is presented in figure 5.6. The dashed arrows present the resource conflicts. To prevent resource conflicts we propose the following:

- To change positions of the leaf cluster sources.
- To change positions of the sources owners in level 1.

When we try to settle a congestion problem to reach a destination, we can introduce unexpected problems to reach other destinations. The aim of the following sections is to develop a method to model all placement constraints and to perform the optimal positions assignment.

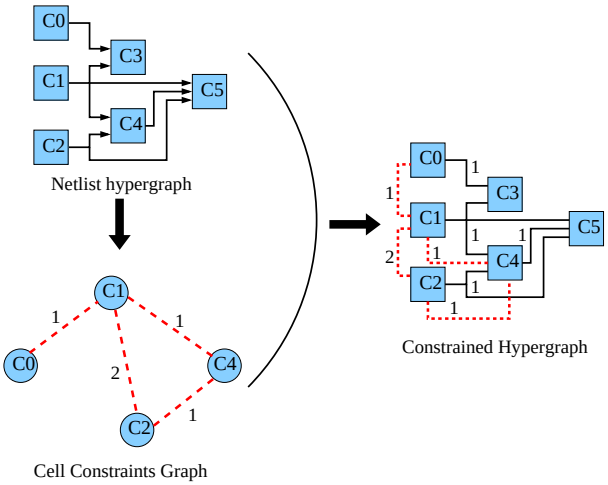


Figure 5.7: CCH: Cell Constraints Hypergraph

5.2.3 Partitioning

The way we distribute logic blocks between MFPGA clusters has an important impact on routing congestion reduction. Based on the upward interconnect specificity, we notice that the number of different paths to connect a source to a destination depends on their enclosing clusters. If they are packed in the same cluster, the source can use more levels to reach its destination and therefore more paths. From this remark we can consider that the netlist cut reduction is an important factor for routability improvement.
A second partitioning objective is deduced from routability conditions presented in $lemma1$. Consider 2 leaf sources $cluster(0, c)$ and $cluster(0, c')$ driving a common destination $cluster(0, c'')$. If we pack both sources in the same cluster we obtain on the one hand $\ell_{up} - \ell'_{up} = 0$ (ℓ_{up} is the lowest used level to reach the common destination). On the other hand we get $pos(cluster(0, c)) - pos(cluster(0, c')) \neq 0$. In this case referring to $lemma1$, the conflict condition is not verified and no resource conflict occurs.
To include this objective in the clustering technique, we propose to construct a Cells Constraints Graph (CCG). The CCG consists of a set of vertices and weighted edges derived from the netlist. An edge is established between 2 vertices (adjacent) when they drive the same destination cluster. Each edge contains a weight equal to the number of common destinations between two adjacent vertices. Using only this graph in the partitioning weakens the obtained clusters netlist results in terms of external communication. To take both objectives into account, we propose, as presented in figure 5.7, to generate a new Constrained Cells Hypergraph (CCH) from the initial netlist hypergraph and the CCG. In this hypergraph, vertices are cells (as in the netlist) and it contains all hyperedges of the netlist and all edges of the CCG. This constrained hypergraph is partitioned using a top-down partitioner and objective priorities are defined according to hyperedges weights. We first construct clusters of the top level and then each cluster is partitioned into sub-clusters. This is done until the bottom of the hierarchy is reached.

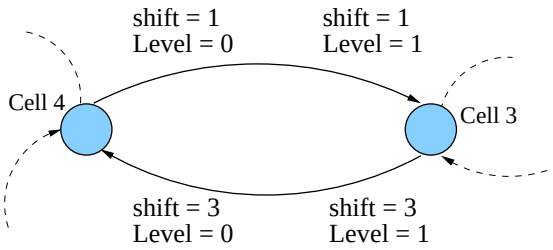


Figure 5.8: ACCG: Advanced Cell Constraints Graph

5.2.4 Detailed placement

If during detailed placement we take $lemma1$ conditions into account, significant gain can be obtained in terms of routability and congestion reduction. For this purpose we introduce the Advanced Cells Constraints Graph (ACCG) which is associated to a given multilevel clustered netlist (previous section) and a placement problem. Input and output Pads are also concerned by the detailed placement. To simplify the approach description, we consider only logic blocks detailed placement. Inputs pads are treated exactly like logic blocks. Output pads are clustered with their logic block drivers.

Advanced Cell Constraints Graph:

We can say that an ACCG is a CCG that contains extra cells partitioning informations required to check conditions of $lemma1$. An ACCG consists of a set of vertices and directed edges derived from the netlist and the way its cells are partitioned between clusters in each level. Each vertex corresponds to a cell of the netlist. A pair of opposite directed edges is established between 2 vertices when they drive the same destination cluster (located at any level), which are then called adjacent. To be able to verify conditions proposed in $lemma1$, we need to add some informations to the constraints graph. Those informations are stored in each directed edge connecting 2 adjacent vertices as a list of pairs $(shift, level)$, featuring:

- The forbidden shift between adjacent vertices positions.

$$shift = (\ell_{up} - \ell'_{up})\ modulo\ (k)$$

- The level where is located the common destination cluster.

It is worthwhile to use the lowest level feedback link to connect a source to its destination, since it has an important impact on delay reduction. That is why, when we construct the ACCG, ℓ_{up} corresponds to the lowest level where the source has to go up to reach its destination. Reducing the conflict between sources using the lowest level is beneficial for the first routing iteration. In fact, as it will be explained in section 5.3, we use an iterative rip-up routing algorithm based on the congestion negotiation. We assign an adjustable cost to each feedback. A lower level induces lower cost; consequently in the first routing iteration, signals will be routed using the lowest levels. Using the lowest levels to construct the ACCG has two advantages:

```
foreach Leaf Cluster cl do
    foreach level ℓ do
        foreach Receiver rc of cl in level ℓ do
            foreach leaf driver dr of rc do
                /*cl and dr both drive rc*/
                if (*) rc has no common subreceiver of dr and cl then
                    if No edge e between cl and dr then
                        create edge e between cl and dr;
                    endif
                    level = GetLevel(rc);
                    shift = ShiftCompute(cl, dr, rc);
                    append pair(shift, level) to edge e;
                endif
            endfch
        endfch
    endfch
endfch
```

Algorithm 5.1: ACCG construction

- Fewer switches will be crossed to route signals.
- A good initial solution for the iterative router exists: first iteration is run with the least number of resource conflicts.

Figure 5.8 presents the Advanced Cell Constraints Graph constructed from the instances placement described in figure 5.5. Algorithm 5.1 describes the used method to construct ACCG. In line (*) of the algorithm, we test whether the common receiver rc has already a sub-cluster (slave) which is also a common receiver of cl and dr. This verification is important to avoid computing many times the same conflict to reach a destination. The conflict can occur when reaching the destination or its owners. For example, in the netlist described in section 5.2.2, we have a conflict driving $cluster(0,5)$ and its owner $cluster(1,1)$. In the routing phase this conflict will be considered only once. That is why, in the generated ACCG we append in the edge only the couple $(1,0)$ corresponding to destination $cluster(0,5)$ and the couple $(1,1)$ corresponding to destination $cluster(1,2)$, but we do not append the couple $(1,1)$ corresponding to destination $cluster(1,1)$. In addition, referring to $lemma1$, if there is no conflict to reach $cluster(l,c)$, there is no conflict to reach any one of its owners.

Optimal solution:

The placement problem that we are treating is a special case of $graph\ labeling$ problem also

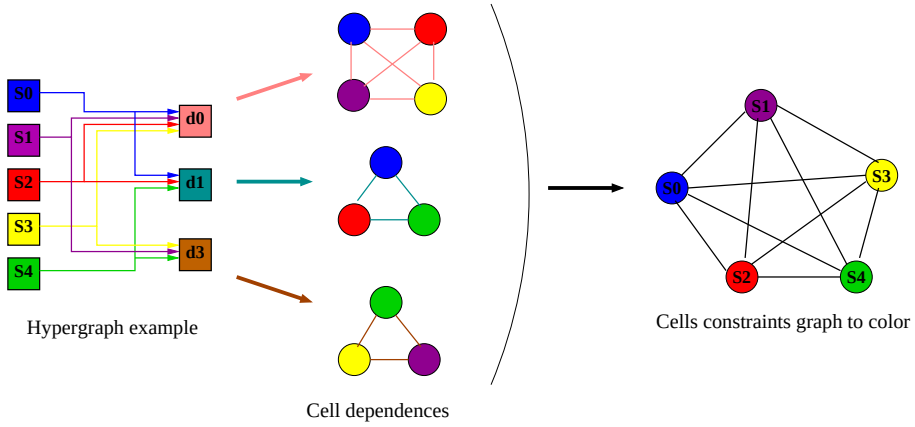


Figure 5.9: Graph coloring problem

known as $graph\ coloring$ [M.R.Garey et al., 1974] [D.J.A.Welsh and M.B.Powell, 1967]. In fact, referring to $lemma1$, if $\ell_{up} = \ell'_{up}$, the conflict condition becomes $pos(cluster(0, c)) = pos(cluster(0, c'))$. Thus to avoid conflicts we must give two different positions (labels, colors) to adjacent vertices of the CCG. The number of different colors (positions) is equal to clusters size (arity). In figure 5.9, we present a netlist example and its corresponding CCG. We suppose we have an MFPGA architecture with 2 levels of hierarchy, arity 4, and that after cells partitioning each source has to go to level 2 to reach its destination (To be consistent with the condition $\ell_{up} = \ell'_{up}$). The obtained constraints graph corresponds to a clique since the 5 vertices are mutually adjacent. In the case described in figure 5.9, it is impossible to find a placement with no conflicts. In fact on one hand, the maximum clique size equal to 5 presents a lower bound on the minimum number of labels needed to color the graph. On the other hand, since clusters arity is equal to 4, we are allowed to use only 4 different colors. The aim of the detailed placement is to reduce, as possible, conflicts number. The remaining conflicts can be solved by instances replication (section 5.2.5) and by using upper levels, if they exist, to route signals (section 5.3).

Simulated Annealing technique:

A detailed placement consists in assigning a position for each cell and each cluster of cells inside its direct owner. The objective is to reduce the number of resource conflicts. By analogy to graph coloring problem [M.R.Garey et al., 1974], we can say that instances placement is an NP-Complete problem. We propose to use metaheuristic algorithm based on simulated annealing to obtain a near optimal solution.
To compute the conflicts number of a specific placement, we take each vertex in the ACCG and we check whether conditions of $lemma1$ are verified; if they are, the global cost function is incremented by 1. Computing this cost for a specific detailed placement is given by the pseudo-code of algorithm 5.2. The cost is updated incrementally in the sequel.
To check whether there is a resource conflict (*), we must check conditions of $lemma1$. To do so

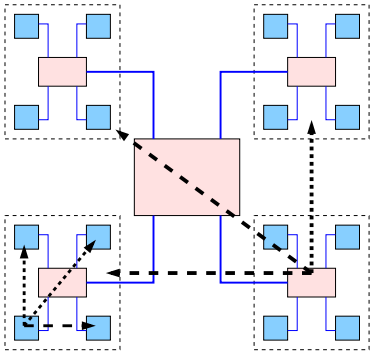


Figure 5.10: Moves range limiters

foreach *vertice* v **do**
  mark vertice v visited;
  foreach *adjacent vertice* adj *of* v **do**
    if adj *was not visited* **then**
      foreach $pair(level, shift)$ **do**
        if *(*)* $conflict(v, adj, shift, level)$ **then**
          cost++;
        endif
      endfch
    endif
  endfch
endfch

Algorithm 5.2: Incremental cost computing

we need information about the first source position, the second source position (adjacent), the forbidden shift and the destination level. All these data are provided by the ACCG.
To find the best detailed placement combination we suggest an adaptive simulated annealing algorithm [S.Kirkpatrick et al., 1983] [F.Aarts et al., 1985]. In this algorithm the operating parameters are controlled using statistical techniques.
Moves are applied randomly to the configuration and consist in assigning new positions. First we choose randomly an element to be moved; it can be a basic cell or a cluster of cells (located at any level). Second we choose randomly the new position inside the direct cluster owner; if it is occupied, we swap both elements positions. The cost function is updated incrementally by evaluating the extra cost of the moved vertices and their adjacent ones. Moving a cells cluster is important (referring to $lemma1$) and can lead to cost reduction. In this case, since the ACCG vertices correspond only to basic elements, we update the cost by visiting all basic elements of the moved cluster and their adjacent ones.
We adopt a hard windowing move restriction approach. As presented in figure 5.10, a cell or

a cluster can only move inside its direct owner. This restriction is important to keep constant the partitioning result obtained by the tool described in section 5.2.3. In addition, with this restriction, we do not have to update the ACCG since the common receivers and the levels to use to reach them always remain unchanged. This yields important run time reduction for the cost updating phase.
Moves that decrease the configuration cost are always accepted, while moves that increase the cost function are accepted with a given probability, directly related to the annealing temperature and inversely related to the cost function variation. As annealing proceeds, the temperature is slowly lowered.

Greedy technique:

We propose to apply a greedy iterative placement approach where only moves decreasing the configuration cost are accepted, thus losing the hill-climbing capability which is important to avoid local minima.
This technique is interesting, since unlike simulated annealing, it can consider an initial good solution. In fact if we build a good initial solution, the greedy technique will improve it locally. This significantly reduces placement run time.
The initial solution construction is described in algorithm 5.3: To find the best sub-clusters or-

```
while repeat do
    repeat = false;
    foreach level ℓ > 0 do
        foreach cluster cl in level ℓ do
            (*) find the best ordering of sub-clusters of cl;
            update cost;
            if cost is decreased then
                repeat = true;
            endif
        endfch
    endfch
endw
```

Algorithm 5.3: Constructed solution

dering (*), we try all combinations. Since a Cluster does not contain more than k sub-clusters, the number of combinations will be $k!$. The cost is updated incrementally using ACCG data.

5.2.5 Logic replication

The idea behind logic replication consists in making copies of one or more logic cells, in order to maintain the logical behavior of a netlist while, hopefully, enabling additional optimization.

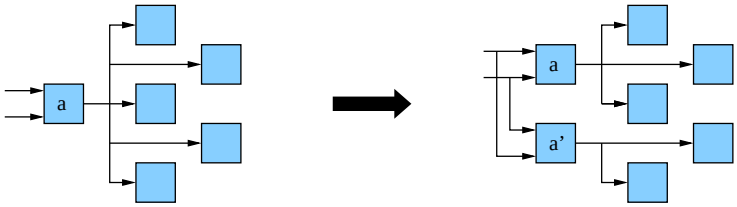


Figure 5.11: Fanout reduction by means of replication

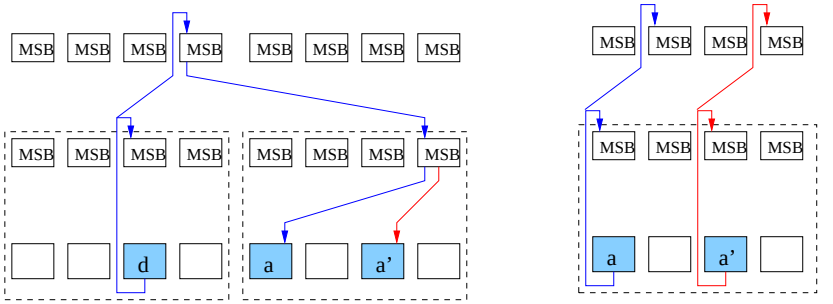


Figure 5.12: Routability improvement

In figure 5.11, we consider netlist logic cell a and suppose that we have created a duplicate cell a'. Cell a' takes precisely the same inputs as a and produces as output exactly the same boolean function. In this situation, the pins in the circuit that need this signal may now obtain it from either the output of a or a'. This adds freedom and enhances routability since it enables to reach destination cells using additional routing resources. In addition if we place the duplicate logic block a' inside the super cluster (owner) containing the original logic block a, we do not add routing congestion to connect a' inputs. In figure 5.12-a we show that, if we group a and a' together in the same cluster, the cost of connecting the same input to cell a' is only one local switch and only one wire. On the other hand, as shown in figure 5.12-b, we can use a' routing resources (2 more upward paths) to reach destination cells.
Consequently, logic replication must be done after original logic blocks partitioning. Thus we have to estimate which logic blocks that need to be duplicated before partitioning, in order to reserve vacant positions and depopulate the containing clusters. To consider this we use the CCG graph to attribute weights to logic blocks. Logic block weight is equal to the number of its adjacent vertices in the CCG. Vertices weights are added to the CCH hypergraph and the partitioner distributes vertices and controls clusters population based on these informations. Once logic blocks are partitioned between clusters, we run the detailed placement. After the last placement iteration, we define logic blocks arrangement inside clusters and we evaluate the number of conflicts that may occur in the first routing iteration. Using the ACCG we can easily identify logic blocks (drivers) leading to these conflicts and duplicate them inside their cluster

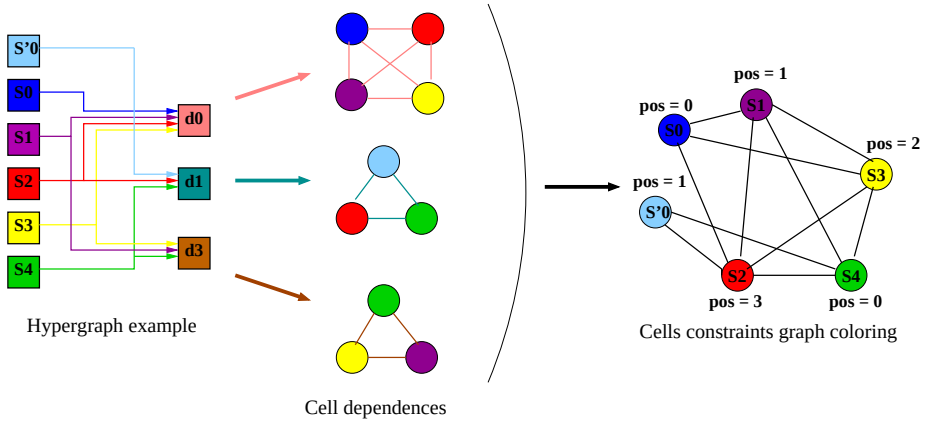


Figure 5.13: Instances replication effect on graph coloring (detailed placement)

owners. When routing iterations are progressing, the router can use the duplicate logic blocks to reach some destinations.
In figure 5.13, we take the netlist proposed in figure 5.9 and we duplicate instance $S0$. $S0$ drives only $d0$ and the copy instance $S\prime 0$ drives $d1$. In this way, the maximum clique size obtained in the CCG, is reduced to 4, allowing to find a solution to the coloring problem. In fact, since S0 and S4 are no more adjacent, we give them the same position 0. As shown in figure 5.13, we solved successfully the coloring problem by giving different positions to adjacent vertices.

5.3 MFPGA routing

The routing problem can be stated as assigning signals to routing resources in order to route all signals successfully. This goal is difficult to achieve in our architecture due to lack of routing resources (depopulated switch boxes). In fact the number of paths to reach a destination from a source is significantly reduced and those paths depend on the location of cells and on the number of levels in the architecture. Thus signals will compete for the same resources and the challenge is to find a way to allocate resources to route all signals. Paths predictability in MFPGA architecture presents a great advantage and simplifies the routing problem. Unlike with the other architecture (Mesh-connected arrays, Triptych ...) there is no need defining a directed graph to describe the routing architecture, thus reducing the routing process complexity.
To route our architecture we implemented a modified version of $PathFinder$ [L.McMurchie and C.Ebeling, 1995]. Since we have only one downward path to reach a destination, we eliminate the breadth-first search in the detailed routing part. Our detailed router corresponds to a function that determines directly the next wire to reach LB destination. Whatever corresponding feedback is chosen, only one path (only one next wire) can lead to destination. Since the feedback choice sets the path to follow, our negotiation must be done on the choice of the feedback leading to a path with less congestion. Consequently, we assign an adjustable cost to each feedback. The global router dynamically adjusts the congestion penalty for each feedback. Initially, each

feedback has a cost equal to the level index where it is located. This confirms the interest of using lower levels to reduce the path length and the number of switches to cross. During this iteration individual routing resources may be used by more than one signal. During subsequent iterations, the penalty for using shared resources is gradually increased so that signals will negotiate effectively for resources. In fact, feedbacks costs for a source will change: a feedback belonging to a higher level can get a lower cost than a feedback located in a lower level. The implemented router is described in algorithm 5.4.

```
while repeat do
    /*global router*/
    foreach signal i do
        repeat
            rip up branch B_ij;
            find feedback f_ij with lowest cost;
            B_ij ← f_ij;
            repeat
                find next_wire;
                add next_wire to B_ij;
            until new t_ij is found ;
        until all sink t_ij are found ;
    endfch
    /*backtrace*/
    foreach node in B_ij do
        /*path from t_ij to s_i*/
        Update cost of f_ij;
    endfch
endw
```

Algorithm 5.4: Adapted $Pathfinder$

The algorithm is based on two simple basic functions that depend strongly on our MFPGA routing architecture. The first one belongs to the global router and determines the feedback to be used by the source to reach the destination. Knowing the source cell index, the sink cell index, this function returns the best level to jump to, in other words the feedback with the lowest cost. The second function belongs to the detailed router and determines the next wire to use to reach destination knowing the actual wire index.

5.4 Timing Analysis

It is interesting to evaluate the performance of MFPGA architectures in terms of functional speed. Thus once an application is completely placed and routed on MFPGA, we propose to

estimate the minimum feasible clock period to run it. Since we have a Tree-connecting network, we propose to divide a path into several sub-paths. Each sub-path connects a source to a sink and consists in going from a source up to a particular level and then down to the sink. The number of sub-paths depends on the number of levels. The first step consists in estimating delays on each sub-path, next we compute the delay for each path composed of several sub-paths. This method enables us to estimate the clock frequency for applications implemented on MFPGA.

5.4.1 Sub-paths delays evaluation

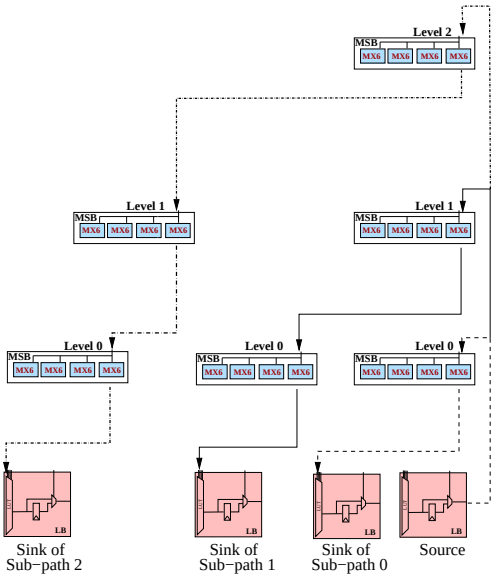


Figure 5.14: Sub-paths timing caracterisation

A sub-path connects a source to a sink and crosses several MSBs. The number of sub-paths in the architecture is limited and depends on the number of levels. Consequently, given an architecture with n levels, we can isolate the n different sub-paths (symmetric structure). In figure 5.14 we show the 3 isolated sub-paths of an architecture containing 3 levels. We use the SPICE circuit simulator to obtain highly accurate delay estimation in each sub-path. Each architecture is composed of combinational sub-paths that either start from a logic block (Combinational/Sequential) or from an input pad pi and end on a logic block (Combinational/Sequential) or an output pad po. To ensure proper circuit operation, we must also take register setup-times t_{set} and sequential propagation delays d_{seq} into account (Sometimes denoted as "Clock-to-Q" delays). Classification of sub-paths and resulting delays is given below:

1. Combinational logic block → Combinational logic block
   $d(p) = d(switches)$
2. Combinational logic block → Output-pad

$d(p) = d(switches) + d(po)$

3. Input-pad → Combinational logic block
$d(p) = d(pi) + d(switches)$
4. Sequential logic block → Sequential logic block
$d(p) = d_{seq} + d(switches) + t_{set}$
5. Sequential logic block → Combinational logic block
$d(p) = d_{seq} + d(switches)$
6. Sequential logic block → Output-pad
$d(p) = d_{seq} + d(switches) + d(po)$
7. Input-pad → Sequential logic block
$d(p) = d(pi) + d(switches) + t_{set}$
8. Combinational logic block → Sequential logic block
$d(p) = d(switches) + t_{set}$

Delays on sub-paths depend on the length of wires connecting MSB and logic blocks. These

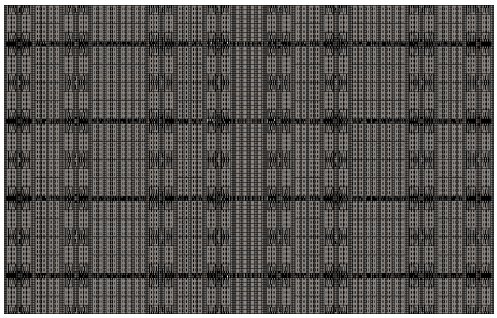


Figure 5.15: 4-levels symmetric MFPGA layout

lengths are extracted from the routed MFPGA layout. Figure 5.15 shows the placed symmetric layout of a 4-levels MFPGA architecture (256 LBs). The basic tiles of the structure are:

- The LB that contains one multiplexer 16:1, one Flip-Flop and a bypass 2:1 Multiplexer,
- The MSB that contains 4 buffered multiplexers,
- The configuration Memory blocks composed of 16 SRAM cells,
- The decoder for configuration memory addressing.

These basic tiles are duplicated at each level to construct the hierarchy recursively. We abut those tiles using a symmetric "H" planing technique.
The MFPGA prototype is targeted to 0.13μ CMOS process with 6 metal layers.

5.4.2 Critical path extraction

Once the circuit has been placed and routed we obtain a direct graph called "routing graph". This graph describes wires that are used to connect logic block pins as described in the netlist. Each wire and each logic block pin becomes a node in this "routing graph" and each passing switch (inside the MSB) becomes a directed edge. Edges are also added between logic blocks inputs and their outputs. Figure 5.16 shows a simple circuit implemented via 2-input LUTs and registers, and the corresponding "routing graph". On this graph we can isolate easily different sub-paths through a depth-first traversal. We replace each sub-path by only one edge labeled with the sub-path delay. We obtain a new direct acyclic graph called "timing graph". In this

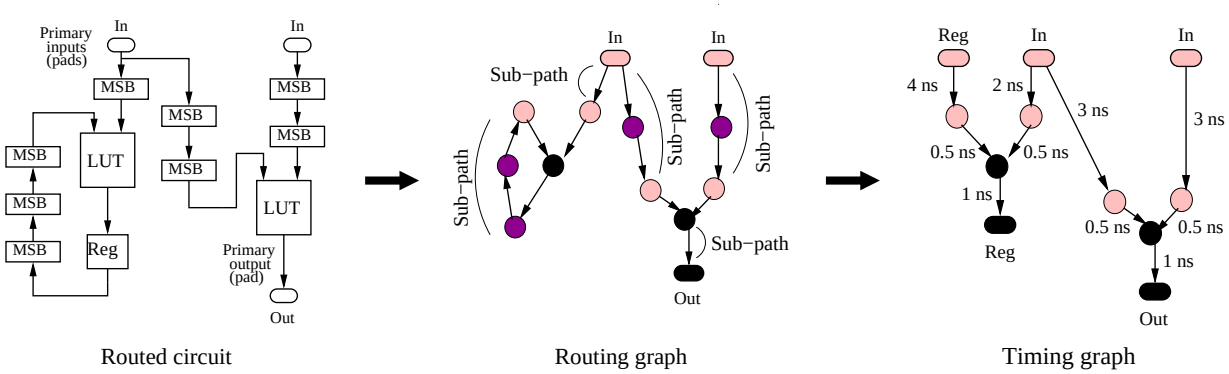


Figure 5.16: Timing graph modeling of a simple circuit

graph nodes represent the input pins and output pins of basic circuit elements, such as registers and LUTs. Register input pins are not linked to register output pins. Register outputs have no edges incident to them and register inputs have no edges leaving them (acyclic graph). Similarly, primary inputs (input pads) have no incident edges and primary outputs (out pads) have no exit edges. Each edge is labeled with the delay required to pass through circuit element or routing (sub-path delay). Figure 5.16 shows the obtained "timing graph" of the routed circuit.
One can determine the minimum required clock period with $O(n)$ computation for a "timing graph" with n nodes via a breadth-first traversal. This traversal begins at nodes with no incident edges (primary inputs and register outputs) and labels each one with a signal arrival time, $T_{arrival}$, of 0. Each node which has incident edges from previously labeled nodes is then labeled with its arrival time according to:

$$T_{arrival}(i) = \max_{j \in fanin(i)} \{T_{arrival}(j) + delay(j,i)\}$$

where node i is the node being labeled, and $delay(j,i)$ is the delay value marked on the edge joining node j to node i. This procedure continues until every node in the graph has been labeled. Then the node with the largest arrival time, which will be always a primary output or a register input, defines the maximum delay, D_{max} (= minimum clock period), through the circuit. In figure 5.16, for example, the arrival time at node Reg is 5.5 ns, which is the largest arrival time,

and hence the maximum circuit delay.

5.5 Experimental results

To evaluate architecture and tools efficiency, we placed and routed several MCNC benchmark circuits.

5.5.1 Placement techniques evaluation

First we propose to check placement techniques efficiency. Placement and routing algorithms were run on a Pentium 4 (3 GHz) machine. To evaluate detailed placement techniques we made experiments as shown in table 5.1. We notice the importance of the detailed placement to achieve circuit routing. The good quality of the iterative solution is penalized by the required run time for iterative algorithms (Simulated Annealing and Greedy). Run time is reduced with the greedy approach, but the solution quality is significantly impaired when compared to the simulated annealing approach. Note that in both cases the detailed placement is followed by logic replication described in section 5.2.5.
Despite that the cost function is updated incrementally, the Simulated Annealing placement requires substantial computing time for the following reason. The used perturbation method is based on element moves and those elements can be clusters (groups) of logic blocks, therefore updating the cost function requires to visit all the basic elements (on the ACCG) of the moved cluster and this takes a large amount of time. As shown in table 5.2 we have evaluated the effect of clusters moving on placement solution quality and run time. We observe that, by allowing only logic blocks (leaves) to move we save a large amount of run time but we decrease the resulting solution quality. This confirms the effectiveness of the conflict condition defined in $lemma1$. The routing solution is degraded by about 3%, meaning that basic elements moves are more efficient than clusters moves. Referring to $lemma1$ a cluster position can, on the one hand, be beneficial to reduce conflicts caused by some leaf sources, but on the other hand, it can introduce conflicts to others. Thus cluster moving efficiency is penalized by the high number of conflicts (between leaf sources) depending on its position. As described previously the detailed placement objective is to reduce the number of resource conflicts that occur in the first routing iteration (I-R% : the Initial percentage of Routed branches). This estimated value is verified when we run the first routing iteration. As shown in table 5.1, we also notice that starting from a good initial solution leads the iterative router to achieve better final results (F-R% : the Final percentage of Routed branches). This clearly confirms that in MFPGA architecture, the routing problem must be treated in early stages and especially in the detailed placement.
We notice that we fail to route circuit benches with high occupancy ratio (more than 80%). This was expected, since in MFPGA architecture, to avoid congested regions, we propose to compensate the depopulated interconnect by a decreased logic blocks occupancy ratio allowing instances replication.

Benchmark		Sim-Annealing			Greedy			Arch	
Circuit	LUTs	I-R%	F-R%	R-Time (mn)	I-R%	F-R%	R-Time (mn)	Levels	Occup-% ancy
alu4	584	99	100	0.9	97	100	0.3	4x4x4x4x4	55
C5315	725	96	100	1.2	94	99	0.7	4x4x4x4x4	69
C7552	881	90	96	1.3	85	93	0.7	4x4x4x4x4	86
tseng	1047	98	100	10	96	100	6	4x4x4x4x4x2	51
ex5p	1064	95	100	60	90	96	32	4x4x4x4x4x2	51
apex4	1262	97	100	100	97	100	45	4x4x4x4x4x2	61
dsip	1370	97	100	20	93	100	9	4x4x4x4x4x2	66
misex3	1397	95	100	70	91	95	34	4x4x4x4x4x2	68
diffeq	1497	96	100	43	92	97	19	4x4x4x4x4x2	73
bigkey	1707	95	100	42	93	97	39	4x4x4x4x4x2	80
apex2	1878	90	94	120	86	91	65	4x4x4x4x4x2	90
s298	1931	92	96	93	90	95	40	4x4x4x4x4x2	94
frisc	3556	89	93	220	84	90	100	4x4x4x4x4x4	86
spla	3690	88	93	255	80	85	110	4x4x4x4x4x4	90

Table 5.1: Detailed placement techniques evaluation

Bench	No Clusters Moving			Clusters Moving		
Circuits	I-R %	F-R %	R-Time (mn)	I-R %	F-R %	R-Time (mn)
alu4	96	100	0.4	99	100	0.9
C5315	94	98	0.7	96	100	1.2
C7552	88	95	0.6	90	95	1.3
tseng	95	98	1	98	100	10
ex5p	92	100	3	95	100	60
apex4	93	97	7.5	97	100	100
dsip	96	100	2	97	100	20
misex3	93	98	2.5	95	100	70
diffeq	93	100	3.5	96	100	43
bigkey	95	97	4	96	100	42
apex2	86	93	10.5	90	94	120
s298	89	90	10	92	96	93
frisc	88	87	15	89	93	220
spla	86	90	17	88	93	255

Table 5.2: Clusters Moving effect

Bench	Mesh			MFPGA	
	Switches number	Area(λ^2) x10^3	N	Switches number	Area(λ^2) x10^3
alu4	81926	207423	13	106496	299008
C5315	220004	555362	19	106496	299008
tseng	210014	531058	17	253952	679936
ex5p	315084	785075	17	253952	679936
apex4	329894	822706	19	253952	679936
dsip	396504	1009107	27	253952	679936
misex3	353376	882555	20	253952	679936
diffeq	332324	836917	20	253952	679936
bigkey	337062	864003	27	253952	679936
AVER	286243	721578		221184	595285

Table 5.3: Area Comparison: MFPGA vs Mesh

5.5.2 Density performances

We use the same benchmark circuits to compare switch and area requirements between MFPGA architecture and clustered Mesh topology (NxN clusters). We use RFPGA, which were described in chapter 3, as reference Mesh-based architecture. We use t-vpack [A.Marquart et al., 1999] to construct clusters and the channel minimizing router VPR 4.3 [V.Betz et al., 1999] to route the Mesh. VPR chooses the optimal size as well as the optimal channel width needed to place and route each benchmark.
We compare the areas of both architectures using successively a simple cost model based on routing switches count, and a more refined model that estimates effective circuit area. The Mesh area is the sum of its basic cells areas like SRAMs, Tri-states and multiplexers. The same evaluation is made for MFPGA, composed of SRAMs, multiplexers and buffers. We use the same cells library, described in chapter 3, for both architectures.
In table 5.3 we show that we can implement several circuits on MFPGA using a smaller area than with Mesh architecture. The area reduction is about 17.5%. We notice that in some cases (“tseng” and “alu4” circuits) MFPGA is penalized by its very low occupancy ratio. We also, notice that despite the effort deployed in the placement phase, several circuits are unroutable using MFPGA resources. This means that the proposed architecture cannot deal with highly congested netlists and especially in the case of high logic occupancy (>75%).

5.5.3 Speed performances

It is clear from previous comparison that MFPGA architecture is more efficient in terms of area and this has a positive effect on circuit speed: the smaller the area, the shorter the connecting wires. In addition the speed of a net is determined by the number of routing switches it must

Circuit	LUTs	Mesh T(ns)	MFPGA T(ns)
pcle	29	6.1	4.34
decod	32	5.36	3.56
cc	33	6.5	4.25
count	37	8.55	7.78
b9	61	10.77	6.98
i4	110	11.14	6.03
c2670	363	19.95	9.97
i9	471	15.67	10.90
alu4	584	20.9	11.26
C5315	725	24.63	15.17
Average	245	12.95	8.02

Table 5.4: Speed Comparison ($0.13\mu m$ CMOS, $1.2V$)

cross. In a Mesh structure, the number of segments in series increases linearly with the manhattan distance, between the logic blocks to be connected. An advantage of a Tree connectivity is that the number of switches in series in a route connecting two logic blocks increases as a logarithmic function of the manhattan distance.
We compared the speed of MFPGA architecture to the Mesh. We implemented the same circuits and we used our timing analyzing tool for MFPGA (section 5.4) and the one proposed in VPR for the Mesh (note: we applied a VPR timing-driven placement and routing). Timing results are presented in table 5.4. In this comparison we only used small benches (< 1024 BLEs). In fact we have only generated architectures layouts (16, 64, 256 and 1024) and, as explained in section 5.4, layout information (wires lengths) is important for sub-paths delay characterization. We notice that MFPGA largely outperforms clustered Mesh architecture (40%) in terms of speed despite we did not integrate timing driven techniques yet. Nevertheless, we expect that the gain ratio will decrease for larger benchmark circuits. This is due to the high fanout of the upward network signals, which increases when we add more hierarchical levels.

5.6 Conclusion

This chapter describes a new hierarchical multilevel FPGA architecture and its suitable configuration tools. The preliminary results show that good LUT and interconnect utilization balancing, reduces area compared with traditional Mesh architectures.
The new topology based on two hierarchical unidirectional networks seems to be robust and can achieve better speed than Mesh-based FPGA architectures. The downward network is a predictable interconnect which has a very interesting impact on accelerating the routing phase.
The routing key of the proposed architecture is the upward network. Enhancing routability leads

to populate the upward network to increase path number between LBs. This can increase area, but may be compensated by controlling clusters signals bandwidth based on Rent's rule.

6

Tree-based FPGA with optimized switch boxes and signals bandwidth

Based on previous investigations, we conclude that optimizing separately clusters switch boxes and signals bandwidth is not efficient to improve density and to deal with highly congested netlists. Thus, we propose an architecture where we can control simultaneously switch boxes depopulation and signals bandwidth reduction. Improvement essentially concerns the upward network of MFPGA architecture presented in the previous chapter.
This chapter is organized as follows. First, we present the improved upward network. We evaluate switches and wiring requirement based on Rent's Rule. Then, we present the developed configuration flow to implement benchmark circuits on the architecture. Finally, we evaluate MFPGA density efficiency compared to the Mesh-based architecture.

6.1 Interconnect Improvement

In the previous chapter a hierarchical Multilevel FPGA architecture (MFPGA) was designed and experimentally evaluated. This architecture unifies 2 unidirectional networks. The downward network has a "Butterfly Fat Tree" topology and allows to connect switch blocks to LBs (leaves) inputs. The upward network uses a limited connectivity Tree to connect LBs outputs to Switch Blocks. While providing good density and some interesting features like an almost predictable routing once the placement is defined, this approach revealed some drawbacks hindering highly congested netlists routing:

- The very depopulated upward network, which only allows each LB output to reach any

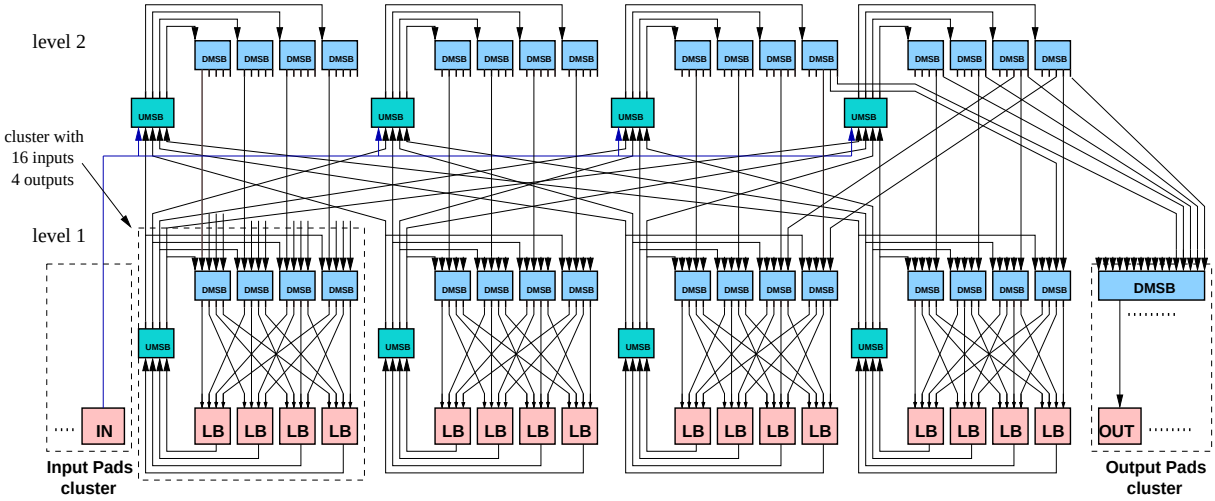


Figure 6.1: Tree-based interconnect: upward and downward networks

destination through paths as the number of levels in the hierarchy, is detrimental for highly congested netlists.

• The placement of clusters (or LBs) inside their owner cluster critically controls available routing resources, thus limiting freedom to re-arrange them and making impossible to construct carry chains in this type of architecture.

• Rent's growth rate ($p = 1$) is very penalizing in terms of wiring requirement. Interconnect wiring takes space and dominates the array size. The wiring topology should be chosen to balance interconnect bandwidth with array size and expected design interconnect requirement. Thus we really need to control clusters signals bandwidth based on Rent's rule.

• In the upward network an LB output fanout depends on the number of levels. Thus in a large architecture with high levels number, performance may be penalized in terms of delays. Switching can be used to improve flexibility (more stop-off points) and to reduce fanout on a feedback line by segmenting tracks .

To alleviate those weaknesses we propose to add routing flexibility by modifying specifically the upward network. We propose, as shown in figure 6.1, to add Upward Mini Switch Boxes (UMSB). These UMSBs allow LBs outputs to reach a larger number of Downward MSBs (DMSBs) and to reduce fanout on feedback lines. The UMSBs are organized in a way that allows logic blocks (LBs) belonging to the same "owner cluster" (at level 1 or above) to reach exactly the same set of DMSBs at each level. Therefore, we can ensure the following points:

- Pads, clusters or logic blocks positions inside the direct owner cluster become equivalent and re-arranging them is unnecessary.

- The interconnect offers more routing paths to connect a net source to a given sink. In this case we are more likely to achieve highly congested netlists routing. In fact, while in the previous architecture each LB output had only one reachable DMSBs per level, with the new upward network, LBs can negotiate with their siblings the use of a larger number of DMSBs. This is more efficient for mapping netlists since instances may have different fanout sizes. For example in fig-

ure 6.1, an LB ouput can reach all 4 DMSBs of its owner cluster at level 1 and all the 16 DMSBs of its owner cluster at level 2.

- By adding UMSBs, long wire segments with high fanout are eliminated. There are no more wires spanning more than one level.

6.2 Interconnect Depopulation

When we add UMSBs in the upward network, the number of architecture switches increases. This can be compensated by reduction of in/out signals bandwidth of clusters at every level. In fact Rent's rule [B.Landman and R.Russo, 1971] is easily adapted to Tree-based structure:

$$IO = c.k^{\ell.p} \tag{6.1}$$

Where ℓ is a Tree level, k is the cluster arity, c is the number of in/out pins of an LB and IO the number of in/out pins of a cluster situated at level ℓ.
Intuitively, p represents the locality in interconnect requirements. If most connections are purely local and only few of them come in from the exterior of a local region, p will be small. In Tree-based architecture, both the upward and downward interconnects populations depend on this parameter. As shown in figure 6.2, we can depopulate the routing interconnect by reducing from 16 to 12 the number of inputs in each cluster of level 1 and outputs from 4 to 3 ($p = 0.79$). This induces a reduction from 16 to 12 of the number of DMSBs in each cluster of level 2 and the UMSBs number from 4 to 3. In this case, if we consider an architecture with 2 levels of hierarchy, we get a reduction of the interconnect switches number from 521 to 416 (19%). By doing so the architecture routability is reduced too. Thus we have to find the best tradeoff between interconnect population and logic blocks occupancy. Dehon showed in [A.DeHon, 1999] that the best way to improve circuit density is to balance logic blocks and interconnect utilization. In MFPGA architecture, the logic occupancy factor is controlled by N, the leaves (LBs) number in the Tree. N is directly related to the number of levels and the clusters arity k. In most cases N is larger than the number of netlists instances. This means that in these cases we have a low logic utilization. This is not really penalizing since it can be compensated by a high interconnect utilization. In other words, the area overhead due to unused LBs is compensated by congestion spreading and interconnect reduction.

6.3 Connection with Outside

As shown in figure 6.1, output and input pads are grouped into specific clusters. The cluster size and the level where it is located can be modified to obtain the best design fit. Each input pad is connected to all UMSBs of the upper level. In this way each input pad can reach all LBs of the architecture with different paths.
Similarly, output pads are connected to all DMSBs of the upper level; in this way they can be

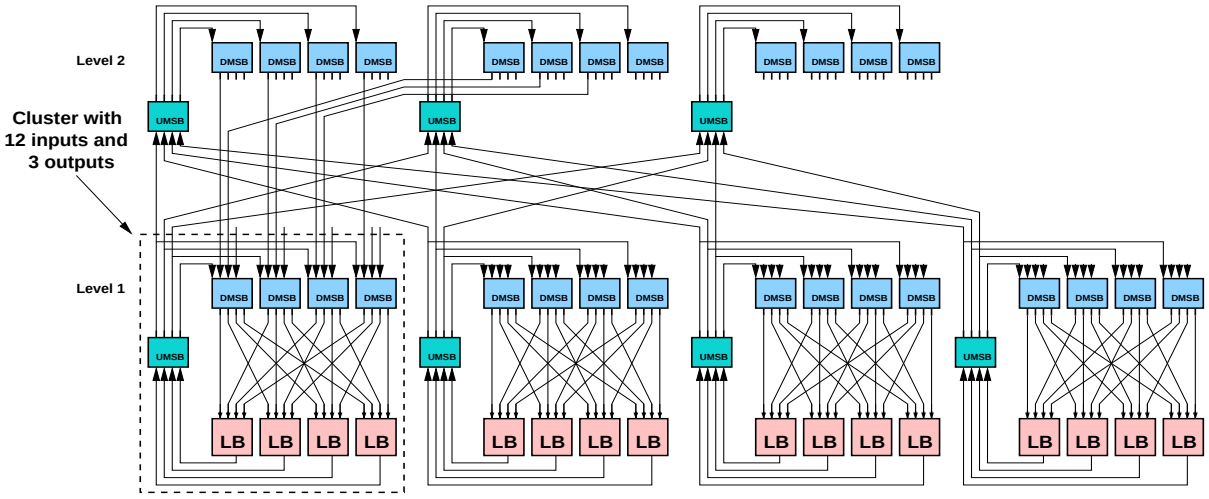


Figure 6.2: Tree-based interconnect depopulation using Rent's rule (level 1 with $p = 0.79$)

reached from all LBs through different paths. As one can notice, in/out pads have higher interconnection flexibility than LBs.
For example we consider $frisc$ netlist which contains 3556 LBs, 20 input pads and 116 output pads. The Tree architecture with 4 arity clusters and 6 hierarchical levels (4096 LBs) fits to this netlist. If input/output pads clusters are located at level 2 (256 clusters) every one contains 1 input/output pad. If we place input/output pads clusters in level 3 (64 clusters), every input cluster contains 1 input and every output cluster contains 2 outputs in order to fit to the netlist.

6.4 Rent's Rule Based Model

Based on Rent's rule presented in equation (6.1), we evaluate the Tree architecture switches requirement to connect LBs.

6.4.1 Switches requirement

We model upward and downward networks separately:

Downward network:

We note:
- $N_{in}(\ell)$ the number of inputs of a cluster located at level ℓ.
- $N_{out}(\ell)$ the number of outputs of a cluster located at level ℓ.
- c_{out} the number of outputs of an LB.
- c_{in} the number of inputs of an LB.

- k clusters arity (size).

Clusters located at level ℓ contain $N_{in}(\ell-1)$ DMSB with k outputs and $\frac{N_{in}(\ell)+kN_{out}(\ell-1)}{N_{in}(\ell-1)}$ inputs. As we assume that the DMSB are full crossbar devices, we get $k(N_{in}(\ell)+kN_{out}(\ell-1))$ switches in the switch box of a level ℓ cluster. Since we have $\frac{N}{k^\ell}$ clusters in level ℓ, we get a total number of switches, related to the downward network, given by:

$$\sum_{\ell=1}^{\log_k(N)} k \times N \times \frac{N_{in}(\ell)+kN_{out}(\ell-1)}{k^\ell}$$

$N_{out}(0) = c_{out}$ is the number of outputs of a Basic Logic Block. Following equation (6.1), we get $N_{in}(\ell) = c_{in}.k^{\ell.p}$ and $N_{out}(\ell-1) = c_{out}.k^{(\ell-1)p}$. The total number of switches used in the downward network is:

$$N_{switch}(down) = N \times (k^p c_{in} + kc_{out}) \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)}$$

Upward network:

Clusters located at level ℓ contain $N_{out}(\ell-1)$ UMSB with k inputs and k outputs. As we assume that UMSB are full crossbar devices, we get $k^2 \times N_{out}(\ell-1)$ switches in the switch box of a level ℓ cluster. As we have $\frac{N}{k^\ell}$ clusters at level ℓ we get the total number of switches, related to the upward network:

$$\sum_{\ell=1}^{\log_k(N)} \frac{k^2 \times N}{k^\ell} \times N_{out}(\ell-1)$$

$N_{out}(0) = c_{out}$ is the number of outputs of a Basic Logic Block. Following (6.1), we get $N_{out}(\ell-1) = c_{out}.k^{(\ell-1)p}$.

The total number of switches used in the upward interconnect is:

$$N_{switch}(up) = N \times k \times c_{out} \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)}$$

The total number of Tree-based interconnect switches is

$$N_{switch}(Tree) = N_{switch}(down) + N_{switch}(up)$$

$$N_{switch}(Tree) = N \times (k^p c_{in} + 2kc_{out}) \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)}$$

The number of switches per Logic Block is:

$$N_{switch}(LB) = (k^p c_{in} + 2kc_{out}) \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)}$$

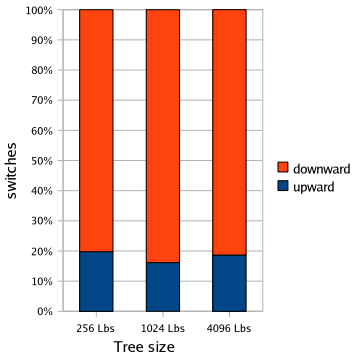


Figure 6.3: Interconnect switches distribution

$$N_{switch}(LB) = \begin{cases} (k^p c_{in} + 2kc_{out}) \times \frac{1-N^{p-1}}{1-k^{p-1}} & \text{if } p \neq 1 \\ (k^p c_{in} + 2kc_{out}) \times \log_k(N) & \text{if } p = 1 \end{cases}$$

$$N_{switch}(LB) = \begin{cases} O(1) & \text{if } p < 1 \\ O(\log_k(N)) & \text{if } p = 1 \end{cases} \tag{6.2}$$

The cost of adding the upward can be compensated by reducing the architecture Rent's parameter. In addition we notice that the number of the upward network switches is smaller than the switches number in the downward network:

$$\frac{N_{switch}(down)}{N_{switch}(up)} = \frac{k^p c_{in} + kc_{out}}{k \times c_{out}}$$

With $p = 1$, $k = 4$, $c_{in} = 4$ and $c_{out} = 1$ this ratio is equal to 5. In figure 6.3, we show the distribution of interconnect resources between the upward and the downward networks for different Tree sizes (we include in/out pads connections).

6.4.2 Wiring Requirements

At each level ℓ of the hierarchy, every switching node has $n_{in}(\ell)$ inputs and $n_{out}(\ell)$ outputs. This makes the bisection width equal to $(c_{in} + c_{out})k^{\ell.p}$. Since $\forall \ell \in \{1, \ldots, \log_k(N)\} \quad k^{\ell.p} \leq N$, the bisection width is $O(N^p)$. For a 2-dimensional network layout this bisection width must cross the perimeter out of the subarray. Thus the perimeter of each subarray is $O(N^p)$. The areas of the subarray will be proportional to the square of its perimeter, making: $A_{subarray} \propto N^{2p}$. The required area per logic block (LB) based on wiring constraints, is therefore evaluated by:

$$A_{LB} \propto N^{2p-1}$$

Unlike, the architecture discussed in the previous chapter, with the new MFPGA architecture we can control bisection bandwidth in each level based on Rent's parameter ($p < 1$). Consequently, physical layout generation may be much optimized since wiring is no more dominant.

6.4.3 Comparison with Mesh Model

Concerning switches per logic block growth, it was established in [A.DeHon, 1999] that in the Mesh architecture:

$$N_{switch}(LB) = O(N^{p-0.5}) \tag{6.3}$$

Equations (6.2) and (6.3) show that in the MFPGA architecture, switches requirement grows more slowly than in common Mesh architecture. These results are encouraging for constructing very large structures, especially when p is less than 1. But this does not mean that our Tree-based topology is more efficient than Mesh-based architecture, since they do not have the same routability. The best way to check this point is through experimental work. Based on benchmark circuits implementation, we compare the resulting areas in the case of Tree-based and the VPR clustered Mesh FPGA.

6.5 Configuration Flow

To explore the modified architecture we must adapt the configuration flow. Since logic blocks positions inside the owner cluster are equivalent, the detailed placement phase (Arrangement inside clusters) is eliminated.

6.5.1 Multilevel Partitioning

The way how logic LBs are distributed between Tree clusters has an important impact on congestion. It is worthwhile to reduce external communications, since local connections are cheaper in terms of delay, but also in terms of routability, as it allows to get more levels (more paths) for connecting sources to destinations. Another way to decrease congestion consists in eliminating competition between nets sources reaching their sinks. This can be achieved by depopulating clusters based on netlist instances fanout. Instances with high fanout need more resources to reach their sinks. Thus in the partitioning phase, instances weights are attributed according to their fanout size.
We use a top-down recursive partitioning approach. First, we construct the top level clusters, then every cluster is partitioned into sub-clusters, until the bottom of the hierarchy is reached.

6.5.2 Routing

Once the netlist is partitioned into a Tree of nested clusters, we attribute randomly to every cluster a position inside its owner (no detailed placement is required). The routing problem consists in assigning the nets that connect placed logic blocks to routing resources in the interconnect structure. The new topology of the upward interconnect adds extra paths to connect a LB to a destination but eliminates the predictability property. Hence we must model the routing resources as a directed graph abstraction $G(V, E)$. As illustrated in figure 6.4, the set of vertices V represents the in/out pins of logic blocks and the routing wires in the interconnect structure. An

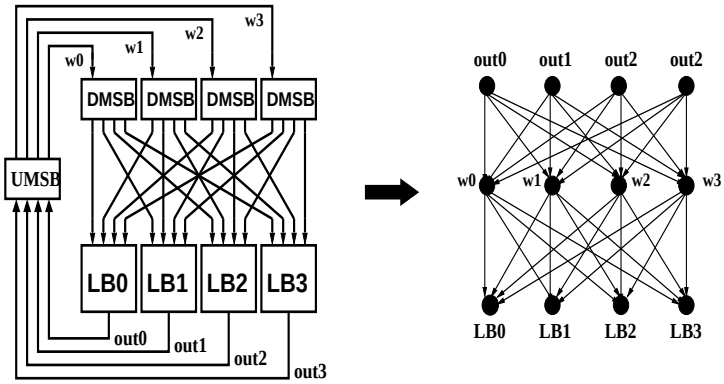


Figure 6.4: Routing graph

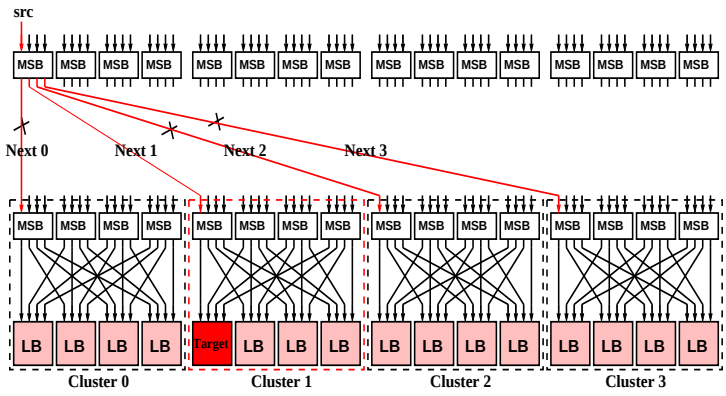


Figure 6.5: Search space pruning

edge between two vertices represents a potential connection between the two vertices. The routing algorithm we implemented is "PathFinder" [L.McMurchie and C.Ebeling, 1995], which uses an iterative, negotiation-based approach to successfully route all nets in a netlist. During the first routing iteration, nets are freely routed without paying attention to resource sharing. Two terminal nets are routed using Dijkstra's shortest path algorithm [T.Cormen et al., 1990], and multi-terminal nets are decomposed into terminal pairs by the Prim's minimum-spanning tree algorithm [T.Cormen et al., 1990]. At the end of an iteration, resources can be congested because multiple nets use them. During subsequent iterations, the cost of using a resource is increased, taking into account the number of nets that share the resource, and the history of congestion on that resource. Thus, nets are made to negotiate for routing resources.
To speed up graph-based search we use the following techniques:

- A^* algorithm: The A^* algorithm speeds up routing by reducing the search space of Dijkstra's algorithm. The search space is reduced by preferentially expanding the search wavefront in the direction of the target node. When the search is expanded around a given wire, the routing algorithm expands the search through the neighbor wire that is nearest the target node. This form of directed search is accomplished by increasing the cost of routing

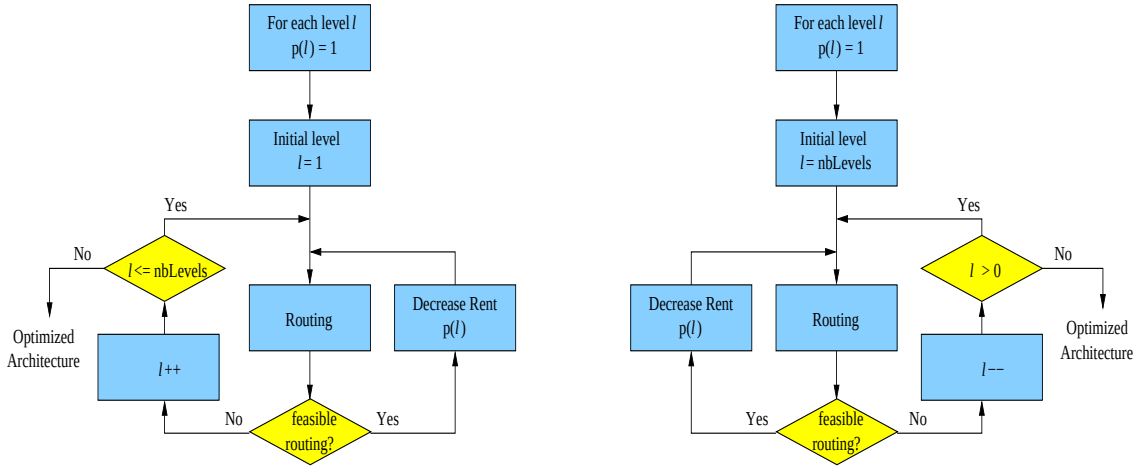


Figure 6.6: MFPGA architecture evaluation flow

wire with a calculated estimate of the cost to the target node. The cost from a node n to the target is easily estimated thanks to hierarchy. In fact by knowing the level where node n (wire) is located, we can determine exactly the minimum number of nodes to cross in the routing graph to reach a target node.

- Hierarchy properties: If we look carefully to the target node position we notice that from a source wire some paths can never reach it. As shown in figure 6.5, if we want to connect src wire to the target LB, we notice that there is only one path from the src to the target. The idea is to prune useless node in the graph expansion phase and consequently reducing search effort. In the case of figure 6.5, only wire $Next1$ allows to reach the target LB. This information is available to us since we know that this wire belongs to the owner cluster (cluster 1) of the target LB.

6.6 Experimental Evaluation

To evaluate the proposed architecture and tool performances, we place and route the largest MCNC benchmark circuits, and consider as a reference the optimized clustered Mesh (VPR-style) architecture. This reference architecture RFPGA was described in chapter 3. We use t-vpack [A.Marquart et al., 1999] to construct clusters and the channel minimizing router VPR 4.3 [V.Betz and J.Rose, 1997] to route it. VPR determines the optimal size as well as the optimal channel width W to place and route each benchmark circuit.

6.6.1 Tree-based architecture optimization

First, we evaluate the efficiency of the new Tree-based architecture to implement MCNC benchmark circuits. With the previous MFPGA architecture (previous chapter), several MCNC circuits were unroutable. As shown in table 6.3, we achieved all the 21 benchmarks routings. This illustrates the improvement in routing flexibility provided by the new upward network.
As explained in section 6.2, MFPGA routability and switches number depend on 2 parameters: p (architecture Rent's parameter) and N (number of LBs in the architecture which defines occupancy ratio). To find the best tradeoff between device routability and switches (area) requirement, we explore MFPGA architectures with various N and p parameters. The purpose is to find for each netlist, the architecture with the smallest area that can implement it. N depends on Tree levels number and clusters arity. For a specific clusters arity, we determine the smallest levels number to implement the circuit. With our tools we can consider, in the same architecture, different p values at each level. Clusters located at the same level have the same Rent's parameter. In the case of Mesh, VPR adjusts the channel width W and for the Tree-based interconnect, we adjust Rent's parameters at every level in order to obtain the smallest architecture fitting every benchmark circuit. Like VPR which applies a binary search to find the smallest architecture channel width, we apply to each level a binary search to determine the smallest Rent's parameter. Depending on levels order processing we distinguish 3 different approaches:

- Bottom-up approach: As shown in figure 6.6 a), we start by optimizing the lowest level up to the highest one. To each level we apply a binary search to determine the smallest input/output signals number allowing to route the benchmark circuit.

- Top-down approach: As shown in figure 6.6 b), we start by optimizing the highest level down to the lowest one. To each level we apply a binary search to determine the smallest input/output signals number allowing to route the benchmark circuit.

- Random approach: All levels are optimized simultaneously. We choose a level randomly, we decrease its input/output signals number, depending on the previous result obtained in this level; then we move to an other level. In this way we move randomly from a level to another until all levels are optimized.

The 3 approaches have the same objective and aim at reducing clusters signals bandwidth for every level. The difference is the order in which levels are processed. In table 6.1, we show architecture Rent's parameter (in each level) obtained with each technique. The first column of the table shows Rent's parameters, at each level, obtained after circuits partitioning. Results correspond to averages of all 21 circuits. We notice that in all cases, architecture Rent's parameters are larger than partitioned circuits Rent's parameters. This is due to the depopulated switch boxes topology. In fact, to solve routing conflicts, a signal may enter from 2 different DMSB to reach 2 different destinations located at the same cluster. In figure 6.7 we show an example of a partitioned netlist to place and route on an architecture with LBs inputs number equal to 2 (2 DMSBs in each cluster located at level 1) and clusters size equal to 4. As shown in figure 6.7, if each signal enters from only one DMSB, we cannot solve conflicts. To deal with such problem we propose to

Level	Circuits partitioning	Architecture top-down	Architecture bottom-up	Architecture random
1	0.64	0.98	0.79	0.88
2	0.55	0.88	0.74	0.79
3	0.50	0.80	0.77	0.76
4	0.49	0.75	0.86	0.73
5	0.45	0.59	0.87	0.7

Table 6.1: Levels Rent's rule parameters

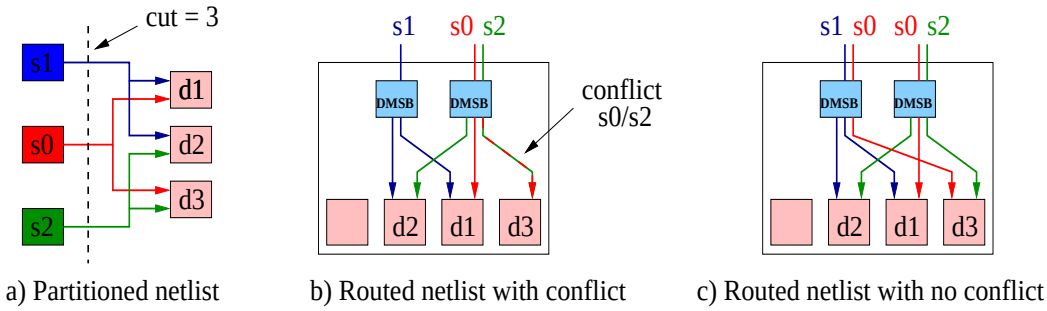


Figure 6.7: A netlist routing example

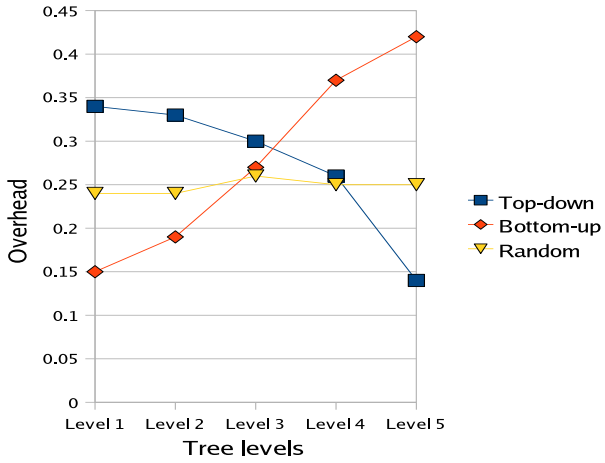


Figure 6.8: Overhead between Architecture and partitioned netlist Rent's parameters (21 benchmark avg.)

Optimizing approach	Area (λ^2) $\times 10^6$	Critical path switches
Top-down	1498	98
Bottom-up	1326	106
Random	1221	101

Table 6.2: Area and performance comparison between various optimizing approaches

enter the signal driven by S0 from two different DMSBs. Thus, the resulting architecture cluster degree is equal to 4, whereas the corresponding part degree is equal to 3 (number of crossing signals).
In figure 6.8, we show the average overhead between partitioning and architecture Rent's parameters with each optimizing approach. We notice that in the case of the top-down (bottom-up) approach, overhead increases when we go down (up) in the Tree. This was expected since the top-down (bottom-up) approach first optimizes high (low) levels. With the random approach, we notice that levels overheads are balanced.
We compared the resulting architectures (with the 3 approaches) in terms of area and speed performance. Average results are shown in table 6.2. We notice that with the random approach we obtain the smallest area (22% less than Top-down and 8% less than bottom-up). This means that optimizing levels simultaneously allows avoiding local minima and obtaining a balanced congestion distribution over levels. The bottom-up approach provides a smaller area than the top-down one. Nevertheless, it is penalizing in terms of critical path switches number (8% more switches than top-down approach). In fact starting by optimizing low levels means that local routing resources are intensively reduced and signals are routed with resources located at higher levels. Consequently, signals routing uses more switches in series.
To reduce the gap between circuit and architecture Rent's parameters, we must improve the partitioning tool and especially the objective function in order to reduce congestion and resources (clusters inputs) required to route signals.

6.6.2 Area Efficiency

We compare MFPGA to the Mesh-based architecture in terms of area efficiency. In both cases we consider architectures with clusters arity 4 and LUT size 4. We determine in each case the smallest architecture implementing every benchmark circuit. As shown in table 6.3, in the case of Mesh we use VPR to find the smallest channel width and in the case of MFPGA we use the random optimizing approach described in the previous section to determine the smallest levels Rent's parameters.
In table 6.4, we observe that the Tree-based architecture has a better density and can implement circuits with lower switches number than the Mesh-based architecture. An average 59% reduction of the switches number is achieved. We achieve a 42% switches reduction with $alu4$

MCNC benchmarks				**Clustered Mesh cluster size 4**			**Tree architecture**		
Circuits Names	LUTs Number	IN Pads	OUT Pads	Arch NxN	Occup %	Channel Width	Architecture levels	Occup %	Routed signals %
alu4	584	14	8	13x13	86	32	4x4x4x4x4	57	100
apex2	1878	39	3	23x23	88	40	4x4x4x4x4x2	91	100
apex4	1262	9	19	19x19	87	42	4x4x4x4x4x2	61	100
ava	14964	11	74	64x64	91	63	4x4x4x4x4x4x4	91	100
bigkey	1707	263	197	21x21	96	28	4x4x4x4x4x2	83	100
clma	8383	61	82	47x47	94	51	4x4x4x4x4x4x4	51	100
des	3235	256	245	29x29	96	29	4x4x4x4x4x4	78	100
diffeq	1497	64	39	20x20	93	29	4x4x4x4x4x2	73	100
dsip	1370	229	197	19x19	95	31	4x4x4x4x4x2	67	100
elliptic	3604	131	114	31x31	94	41	4x4x4x4x4x4	87	100
ex1010	4589	10	10	35x35	93	43	4x4x4x4x4x4x2	56	100
ex5p	1064	8	63	17x17	92	44	4x4x4x4x4x2	51	100
frisc	3556	20	116	30x30	98	45	4x4x4x4x4x4	86	100
misex3	1397	14	14	20x20	87	36	4x4x4x4x4x2	68	100
pdc	4575	16	40	35x35	93	61	4x4x4x4x4x4x2	55	100
s298	1931	4	6	23x23	91	27	4x4x4x4x4x2	94	100
s38417	6406	29	106	41x41	95	37	4x4x4x4x4x4x2	78	100
s38584	6447	39	304	41x41	96	36	4x4x4x4x4x4x2	78	100
seq	1750	41	35	22x22	90	40	4x4x4x4x4x2	85	100
spla	3690	16	46	31x31	96	53	4x4x4x4x4x4	90	100
tseng	1047	52	122	17x17	90	27	4x4x4x4x4x2	51	100
Average	2567	78	87		92	38		74	100

Table 6.3: Netlists and architectures characteristics

MCNC	**Clustered Mesh Cluster size 4**			**Tree architecture**			**Gain**		
Circuits	SW $\times 10^3$	SRAM $\times 10^3$	Area (λ^2) $\times 10^6$	SW $\times 10^3$	SRAM $\times 10^3$	Area (λ^2) $\times 10^6$	SW %	SRAM %	Area (λ^2) %
alu4	100	74	319	47	43	182	53	41	42
apex2	506	375	1541	173	127	565	65	66	63
apex4	359	267	1092	138	103	466	61	61	57
ava	3570	2326	10131	1428	1047	4661	60	55	54
bigkey	349	253	1056	129	101	450	63	60	57
clma	2541	1879	7672	1031	821	3614	59	56	52
des	667	487	2047	326	247	1087	51	49	46
diffeq	307	226	954	121	108	445	60	52	53
dsip	310	224	934	143	107	484	52	48	46
elliptic	944	701	2883	326	247	1087	65	48	62
ex1010	1234	915	3763	515	410	1804	58	55	52
ex5p	305	224	915	134	103	460	56	54	49
frisc	952	811	3287	346	254	1134	63	68	65
misex3	354	263	1085	150	113	502	53	53	50
pdc	1636	1207	4889	714	523	2329	56	56	52
s298	380	280	1192	121	108	445	68	61	62
s38417	1508	1126	4662	493	439	1807	67	60	61
s38584	1501	1113	4590	535	452	1898	64	59	58
seq	463	343	1411	163	123	541	64	64	61
spla	1144	847	3448	428	299	1350	62	64	60
tseng	216	157	665	110	90	370	41	36	33
Average	920	670	2787	411	314	1221	59	55	56

Table 6.4: Tree vs. clustered VPR-style Mesh

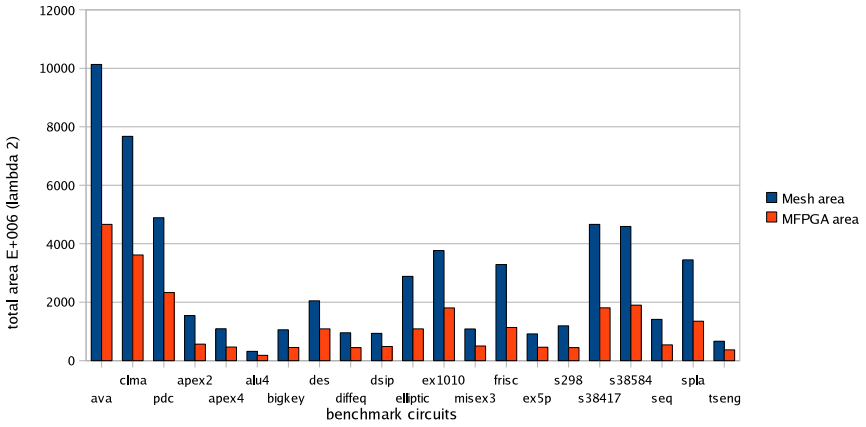


Figure 6.9: MFPGA area vs Mesh area (21 benchmark circuits)

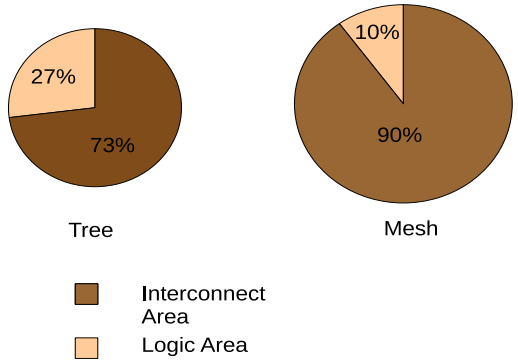


Figure 6.10: Area distribution between interconnect resources and logic blocks

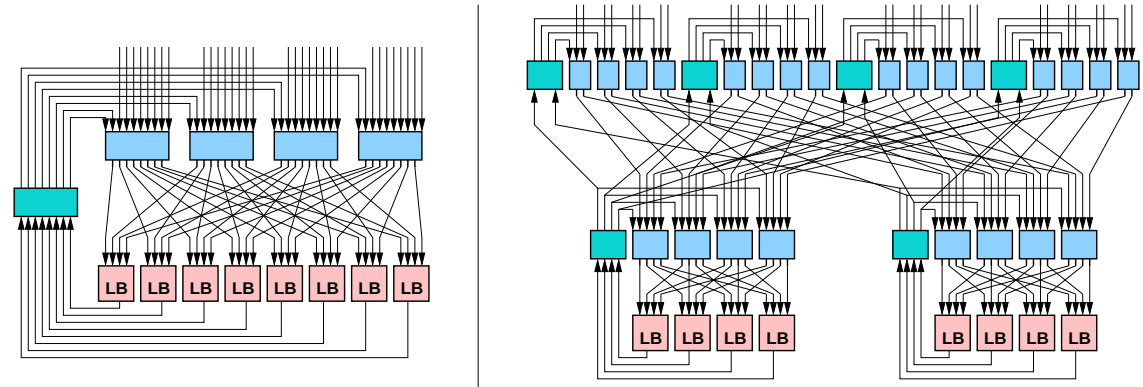
(smallest circuit), and 60% with ava (largest circuit). This confirms that Tree-based interconnect is very attractive for both small and large circuits.

We compare the areas of both architectures using a refined estimation model of effective circuit area. The Mesh area is the sum of its basic cells areas like SRAMs, Tri-states and Multiplexers. The same evaluation is made for the Tree, composed of SRAMs and Multiplexers. Both architectures use the same symbolic cells library. As presented in figure 6.9, in all cases, the required Tree area is smaller than the Mesh one. On the average with the Tree architecture we save 56% of the total area.

The Tree architecture efficiency is due essentially to the ability to control simultaneously logic blocks occupancy and the interconnect population, based on LBs number N and architecture Rent's parameter p respectively. For example in the case of $apex2$ circuit, we used an architecture with a high logic occupancy (91%) and a high Rent's parameters as shown in table 6.5. In the case of $tseng$ circuit, we have a low occupancy (51%) and we achieve routability with a low architecture Rent's parameters as illustrated in table 6.5. This confirms that we can balance interconnect

Circuits	Level 1	Level 2	Level 3	Level 4	Level 5
apex2	1	0.89	0.86	0.84	0.77
tseng	0.79	0.79	0.79	0.72	0.67

Table 6.5: Levels Rent's parameters for 2 circuits



8 LBs grouped into the same cluster (arity 8)
Multiplexers 10:1 / 484 switches / 80 wires

8 LBs grouped into 2 clusters (Arity 4)
Multiplexers 5:1 / 296 switches / 108 wires

Figure 6.11: MFPGA architectures with different arity factors

and logic blocks utilization thanks to logic occupancy decreasing and congestion spreading. In fact we have a 20% lower LBs occupancy than Mesh case, the logic extra area allows us to better exploit interconnect. The Tree high-interconnect/low-logic utilization approach is just opposed to the high logic utilization approach that has been adopted for Mesh-based FPGA. As shown in figure 6.10, unlike Mesh case where interconnect occupies 90% of the overall area, in Tree-based architecture interconnect occupies 73%. In this case logic area is increased by 20% and interconnect area is reduced by 69%.

6.6.3 Clusters Arity Effect

As one can notice, we considered in table 6.3 Tree architecture with clusters arity equal to 4. To get an idea about arity effect on architecture density and speed performances, we vary clusters arity and evaluated for every benchmark circuit the required switches and wires number and the resulting critical path. Since we have no information about layout characteristics, to evaluate performances, we used a simple model based on evaluation of the number of switches crossed by the critical path. This estimation consists in determining the longest path in terms of switches (ignoring wires delays). The extraction of the longest path were described in section 3.2.5.
We notice that when we increase clusters arity, the required switches number increases. When clusters arity increases, the required multiplexers grow larger and consequently the bound on area efficiency goes down. For example as shown in figure 6.11, in the case of architecture with

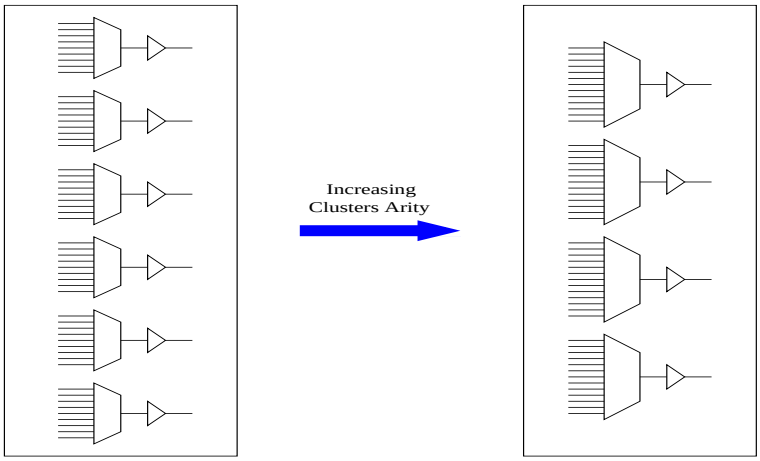


Figure 6.12: Varying multiplexers sizes and numbers

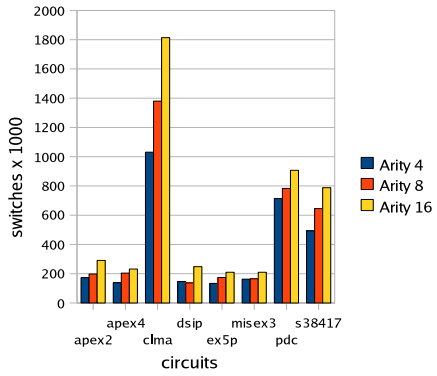


Figure 6.13: Clusters arity effect on switches number

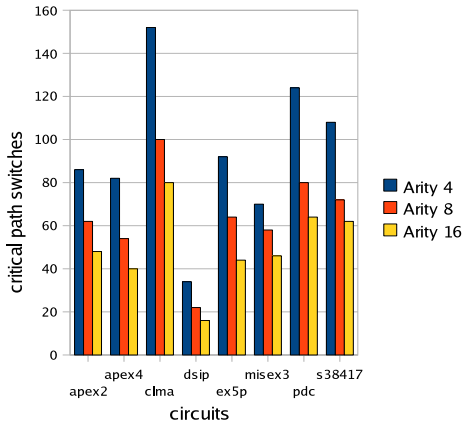


Figure 6.14: Clusters arity effect on critical path crossed switches

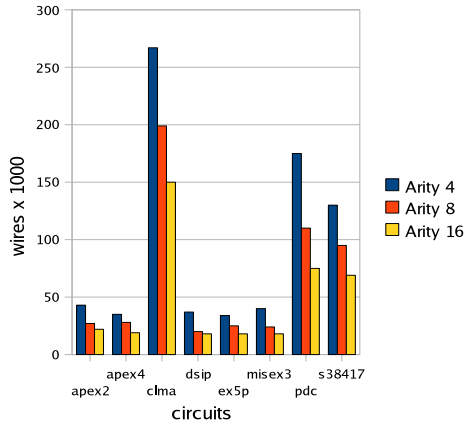


Figure 6.15: Clusters arity effect on wires number (<=> Muxes number)

clusters arity 4 we use muxes 4:1 and muxes 5:1. With clusters arity 8, we use muxes 8:1 and muxes 10:1. As shown in figure 6.13, switches number is increased by 23% when we increase clusters arity from 4 to 8.
When we increase clusters arity, the architecture levels number decreases. Consequently multiplexers sizes increase and their total number decreases. Thus the total number of wires decreases. For example, as shown in figure 6.15, wires number is reduced by 32% when we increase clusters arity from 4 to 8. As shown in figure 6.12, FPGAs use unidirectional buffers nowadays. According to [S.Kaptanoglu, 2007], buffers and SRAM are the major factors behind static power dissipation. By increasing clusters arity we can reduce multiplexers number and consequently buffers number without reducing routability.
In terms of performance we notice, as shown in figure 6.14, that the number of switches crossed by the critical path decreases when we increase arity. With larger clusters arity, we can absorb larger number of nets, and communication becomes local. For example when we increase clusters arity from 4 to 8, the crossed switches number in the critical path is reduced by 27%.
The choice of clusters arity must be consistent with the application specifications and constraints. For applications requiring high speed performance and low power dissipation, it is recommended to use clusters with high arity (8-16). If we need to reduce silicon area, using small clusters arities seems to be more efficient.

6.6.4 LUT Size Effect

In this section we evaluate the effect of LUTs size k (number of LUT inputs) on MFPGA performances. Mapping is the phase where logic gates are transformed into k-bounded cells. When k increases the size of LUTs increases and their number decreases. Thus, as shown in [E.Ahmed and J.Rose, 2000], the effect of k increasing is not predictable and can only be determined by experimentation.
Our experimentation is based on generating circuits with LUTs sizes ranging from 3 to 7 and

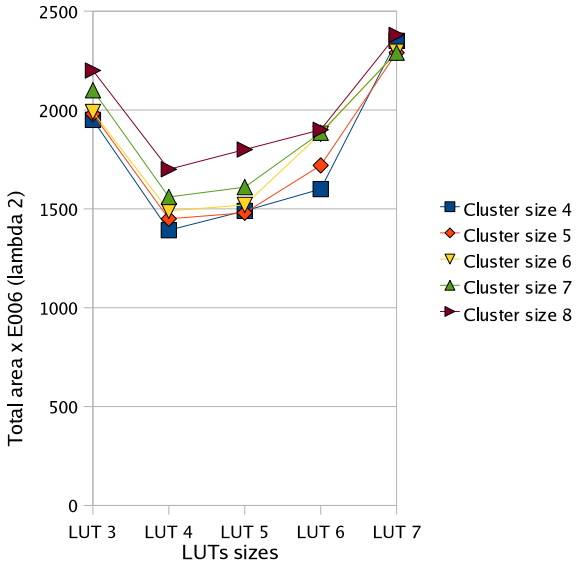


Figure 6.16: Total area for clusters sizes 4-8 (21 benchmark avg.)

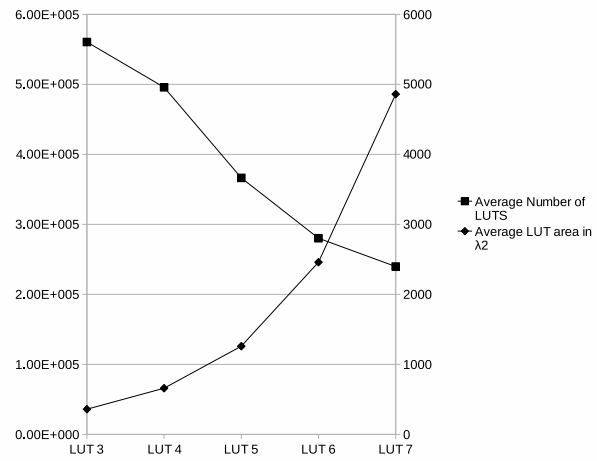


Figure 6.17: LUTs number and LUT area versus LUT size (for cluster arity = 4)

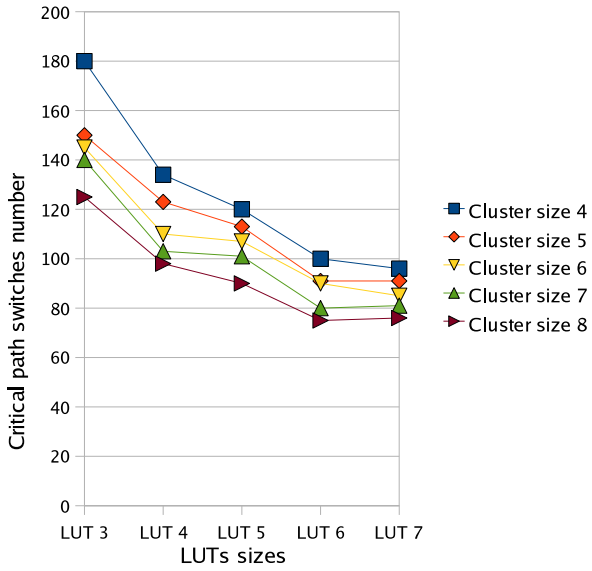


Figure 6.18: Critical path switches number clusters sizes 4-8 (21 benchmark avg.)

implementing them on MFGA with these LUTs sizes and clusters arities ranging from 4 to 8. First, as shown in figure 6.16, we evaluate the effect of LUTs size changing on MFPGA area. Results correspond to the average area of all the 21 largest circuits. We notice that initially there is a reduction in area between 3-LUT and 4-LUT and afterwards there is an increase in area with the rise in LUT and cluster size. It can be noted that architecture with LUT size 4 and cluster size 4 gives overall most efficient average area for benchmark circuits. The bad effect of cluster size increase on area confirms results and discussions made in the previous section.
The total area can be broken into two parts, the logic block area and interconnect area. From our experimentation we notice that logic area increases as LUT size increases. This area is the product of the total number of LUTs times the area per LUT. A plot of these two components for clusters arity equal to 4, is given in figure 6.17 (the left vertical axis presents area per LUT in (λ^2) and the right vertical axis presents LUTs number). The logic block area grows exponentially with LUT size as there are 2^k bits in a k-inputs LUT. As k increases, though, the number of LUTs decreases (because each LUT can implement more logic functions) as shown by the downward curve in figure 6.17. However, the rate of increase in area is steeper than the rate of decrease in LUTs number. Concerning the interconnect area we notice that it decreases with LUT size increase. Since logic area increase is steeper than interconnect area decrease, we obtain the upward trend in figure 6.16.

The second key metric is critical path delay. Since we have not an accurate wires lengths estimation (we do not have yet a complete layout generator), we only evaluate the number of switches crossed by the critical path. Figure 6.18 shows the average critical path switches number across all the 21 circuits as a function of clusters arities and LUTs sizes. Observing the figure, it is clear that increasing clusters arity and LUTs size decreases the number of switches crossed by the

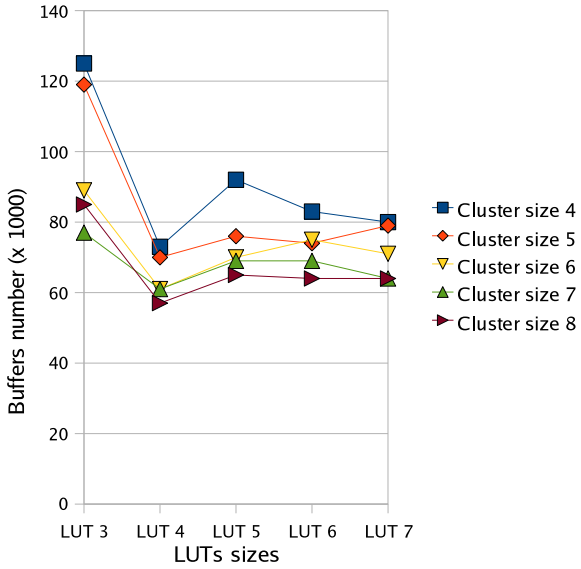


Figure 6.19: Buffers number clusters sizes 4-8 (21 benchmark avg.)

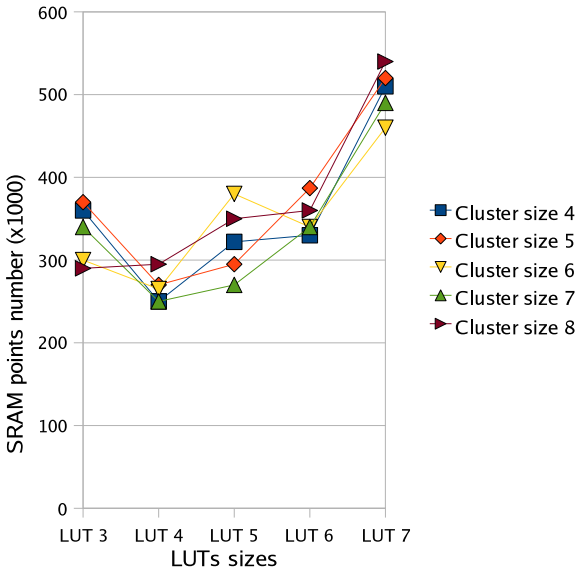


Figure 6.20: SRAM points number clusters sizes 4-8 (21 benchmark avg.)

critical path. These decreases are significant: an architecture with cluster arity 4 and LUT size 3 has an average critical path switches number of 180 while cluster arity 8 and LUT size 7 has an average critical path switches number of just 76. This behavior is explained by the decrease of the number of LUTs and clusters in series on the critical path. Nevertheless, to get an idea about the accurate delay we have to consider the increase of intrinsic LUT delay when its size increases. In addition, clusters arity increasing, as it will be explained in the next section, may induce wires lengths increase.
To get idea about LUT size effect on static power dissipation, we evaluate buffers and SRAM points numbers in function of LUT size. We notice, as shown in figure 6.19, that initially there is a reduction in buffers number between 3-LUT and 4-LUT and afterwards buffers number increases with the rise in LUT size. As shown in figure 6.20, we notice that SRAM points number has the same behavior. Clearly, the results for all clusters sizes consistently show that LUT size 4 gives minimum leakage energy compared to other LUT sizes. This result is expected since LUT size 4 achieves the highest total-area efficiency.

6.7 Conclusion

The improved Tree-based architecture significantly alleviates placement constraints and offers better routability. Based on MCNC benchmark implementation, we showed that the Tree-based architecture has better area efficiency than the common VPR-Style clustered Mesh. We also showed that Tree-based architecture efficiency in terms of area, performance and static power can be controlled by interconnect Rent's parameters, clusters arity and LUTs size. In this way this architecture can be tuned and adapted to specific domains and to satisfy various tradeoffs. Nevertheless, this Tree-based architecture can be penalizing in terms of physical layout generation, it does not support scalability and does not fit with a planar chip structure, especially for large circuits. Conversely, the Mesh and in particular the Mesh of Tree (a Mesh where clusters local interconnect has a Tree topology) has a good physical scalability: once the cluster layout is generated we can abut it to generate Mesh layouts with the desired size and form factor. We are interested to take advantage of both architectures strong points by unifying Mesh and Tree interconnects (Mesh of Tree) to get better area efficiency and layout scalability.

7 Mesh of Tree Architecture

As shown previously, Tree-based architecture is better optimized in terms of switches number than VPR Mesh architecture. Nevertheless, the Butterfly Fat Tree in stand alone mode, is very penalizing in terms of physical layout generation. As illustrated in figure 7.1, it does not support scalability and does not fit with a planar chip structure especially for large circuits. In the Tree lower levels, clusters are small and close by, but as we get higher into the Tree (connecting large clusters), wiring distances increase. Interconnect in the higher part of the Tree may need to be subdivided. Our idea is to use Tree topology as an intra-cluster interconnect and to use Mesh topology to achieve inter-clusters interconnection. In this way we can limit the Tree size and generate a layout of any size by tiles abutment.
First, we describe the Mesh of Tree interconnect topology. Then, we present 3 different configuration approaches to implement circuits. Finally, we compare the proposed architecture to the Tree-based and Mesh-based architectures in terms of switches requirement.

7.1 Mesh of Tree architecture

The architecture we propose has a Mesh of Tree interconnect topology and is built as a matrix of abutted nodes presented in figure 7.2; every node has a Tree-based intra-cluster interconnect. The resulting network corresponds to a Mesh of clusters (each one encapsulating the intra-cluster interconnect and the LBs). Clusters surrounded by Mesh interconnect are called Mesh clusters and clusters included in the diverse levels of the Tree are called Tree clusters. This topology is proposed as an alternative to the common cluster-based Mesh architectures. As shown in figure 7.2, there exist different ways to connect signals to the LUT input muxes. In Xilinx Vir-

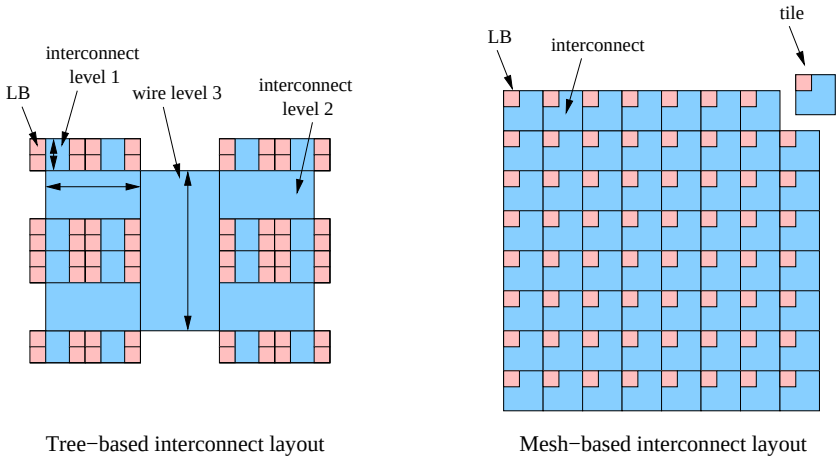


Figure 7.1: Tree-based and Mesh-based layouts view

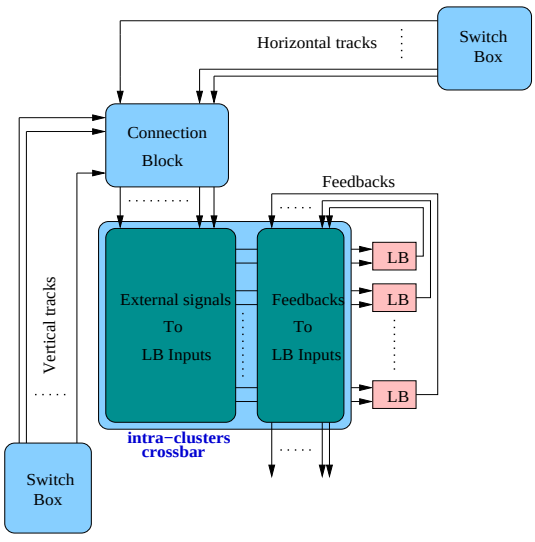


Figure 7.2: cluster-based Mesh interconnect components

tex architectures [Virtex, 5], routing tracks are connected directly to input muxes. In the VPR architecture [V.Betz et al., 1999] and the Altera Stratix architecture [D.Lewis and al, 2003], routing tracks are connected to input muxes via an intermediate level of muxes called connection block. VPR-style interconnect has a sparsely populated connection block and a fully populated intra-cluster crossbar. The fully populated intra-cluster crossbar is simple but takes no advantage of the logical equivalence of LUT inputs and induces significant inefficiency. Lemieux and Lewis [Lemieux and Lewis, 2004] improved the basic VPR-style interconnect in two ways. They proposed an approach to generate highly routable sparse connection block. Furthermore, they showed that the intra-cluster full crossbar can be depopulated to achieve significant area reduction without performance degradation. A practical example is Stratix, which depopulates this crossbar by 50% [D.Lewis and al, 2003]. All these studies consider the connection block interconnect level and the intra-cluster crossbar separately. In [W.Feng and S.Kaptanoglu, 2007], authors investigate joint optimization of both crossbars and proposed a new class of efficient topology. Nevertheless, in the intra-cluster crossbar they optimized only the part connecting external signals to LBs inputs. Using a full crossbar to connect feedbacks (LBs outputs) to LBs inputs is very penalizing and imposes a very low bound on the cluster LBs number. For example we assume we have a cluster with 256 LBs and we use a full crossbar to connect feedbacks to 4-LUT inputs. This means we need 256×1024 switches to route clusters internal signals only, which is very expensive. In the proposed Mesh of Tree architecture, our first contribution corresponds to a joint optimization of connection blocks and intra-cluster interconnect topologies. We optimize both crossbars: 1) connecting external signals to LBs inputs 2) connecting feedbacks to LBs inputs. Our second contribution consists in using only single-driver interconnect based on unidirectional wires. As illustrated in [G.Lemieux et al., 2004], single-driver interconnect has a good impact on density improvement. In the sequel, architecture and its configuration tools are described.

7.1.1 Cluster local interconnect

Mesh clusters are composed of Logic Blocks (LBs) which communicate within a programmable local interconnect. The intra-cluster interconnect is organized as a Tree and has the topology previously used to connect MFPGA LBs (previous chapter). Mesh Clusters input and output pins are connected to LBs as MFPGA input and output pads. Figure 7.3 illustrates Mesh cluster Tree-based local networks and its interface with the Mesh-based interconnect. Input and output signals are grouped into clusters located at level ℓ of the Tree (in figure 7.3 we have $\ell = 1$):

- every cluster of input signals contains 4 inputs connected to the 4 adjacent channels. As shown in figure 7.3, every input is connected to all UMSBs located at level $\ell + 1$ of the Tree. In this way the 4 inputs are logically equivalent since they exactly share the same routing resources to reach LBs. Inputs located at different clusters can all reach every LB but with different paths.

- every cluster of output signals contains 4 outputs connected to the 4 adjacent switch boxes. As shown in figure 7.3, all outputs are connected to all DMSBs located at level $\ell + 1$ of the Tree. In this way the 4 outputs are logically equivalent. Outputs located at different clusters can all be

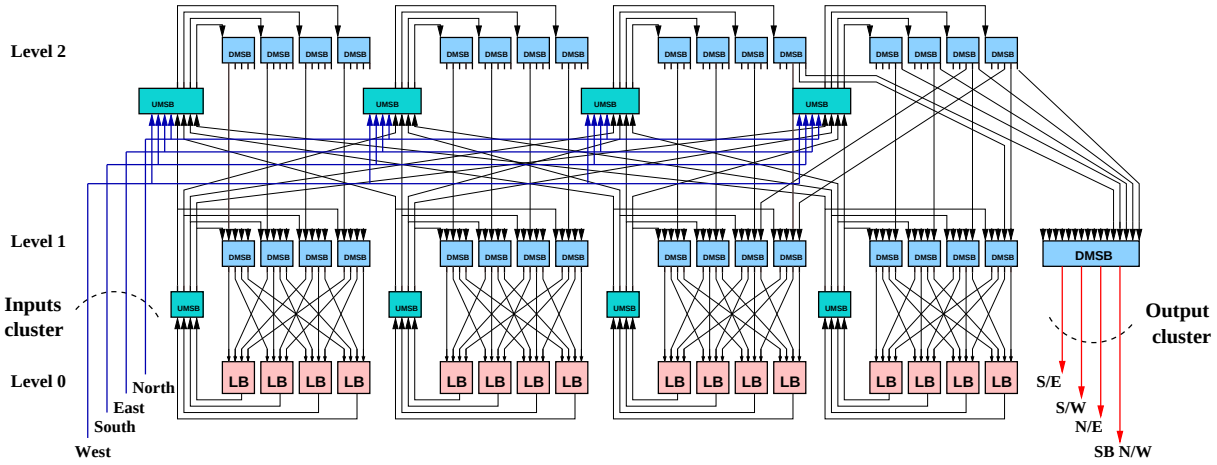


Figure 7.3: 16-LBs cluster interface: External input and output connections

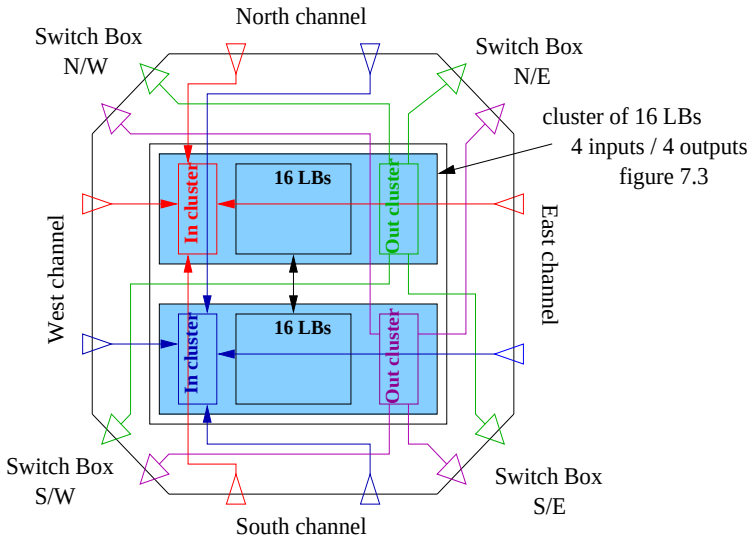


Figure 7.4: Mesh clusters pins distribution

reached by every LB but from different paths.
The distribution of equivalent inputs and outputs over the 4 sides has an important impact on routability and eliminates constraints in the placement of Logic Blocks inside clusters. All four Mesh cluster sides have the same number of inputs and outputs. Side inputs and outputs numbers depend on the number of Tree leaves and on the level where they are located.

$$Nb_{in} = \frac{N}{k^{\ell_{in}+1}}$$

$$Nb_{out} = \frac{N}{k^{\ell_{out}+1}}$$

k is Tree clusters arity and N is the number of Tree leaves; ℓ_{in} and ℓ_{out} are respectively levels where input and output clusters are located. For example the number of inputs in figure 7.3 is equal to 4 since input cluster is located at level 1. If we move input clusters to level 0, we get 4 input clusters (16 inputs) connected to UMSBs of level 1. The number of inputs and outputs can be different if input and output clusters are located at different levels.
In figure 7.4, we show an example of a Mesh cluster containing 2 16-LBs clusters (figure 7.3). This cluster has in total 8 inputs and 8 outputs equally distributed on the 4 sides. In all sides we have the same number of inputs and outputs.

7.1.2 Mesh routing interconnect

As presented in figure 7.2, clustered Mesh architecture is composed of logic blocks clusters, switch blocks, and connection blocks. Interconnection between clusters is made by routes through switch blocks, along horizontal and vertical routing channels.

Mesh with bidirectional wires

The connection block is the region where the cluster input and output pins connect to the routing channels. In the case of bidirectional wiring, Mesh cluster output can connect to any channel track. Figure 7.5 shows how every output can connect one or several tracks. Connection block population is defined by $F_{c_{in}}$ and $F_{c_{out}}$ parameters, where $F_{c_{in}}$ is routing channel to cluster input switch density and $F_{c_{out}}$ is cluster output to routing channel density. In figure 7.5 $F_{c_{in}} = 0.5$ and $F_{c_{out}} = 0.25$.
The switch block is the place where connections are made between horizontal and vertical routing channels, allowing nets to turn around corners or to extend farther along the channel. Figure 7.6 shows a design using tri-states to drive a channel track.
Every routing channel contains W parallel wires tracks, where W is called the channel width. The same width is used for all channels. The Mesh cluster inputs are connections from the external routing, carrying signals from other clusters into this one. Cluster local and external interconnect flexibilities can be controlled separately:

- Cluster local interconnect: as was illustrated in the previous chapter, we can use different Rent's parameters to change Tree-based interconnect flexibility,

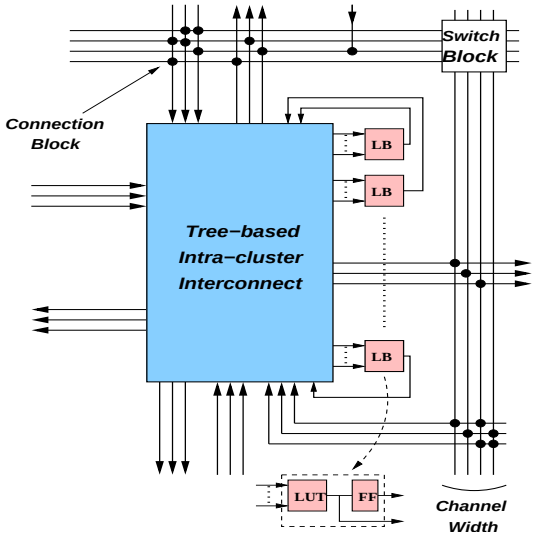


Figure 7.5: Node of Mesh of Tree architecture with bidirectional wires

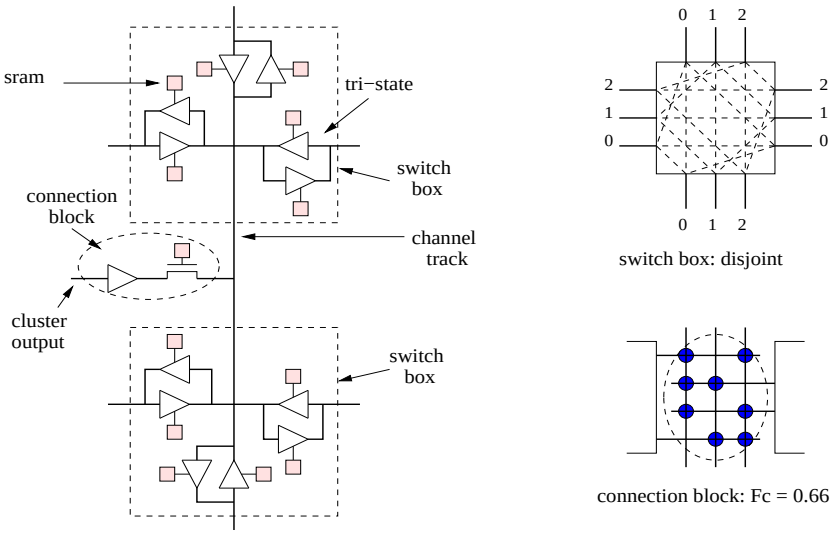


Figure 7.6: Connection block and switch box designs: bidirectional wires

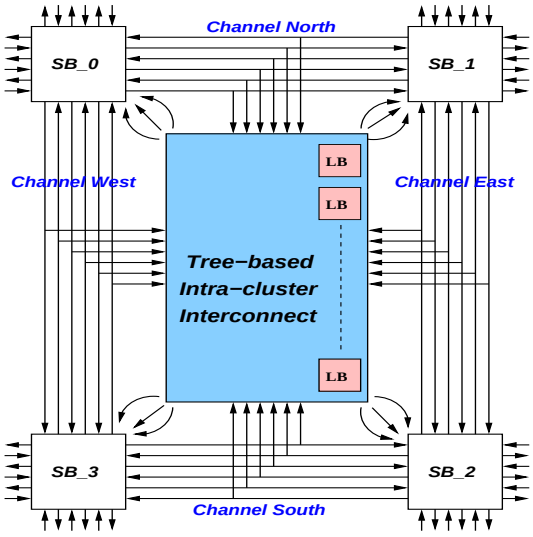


Figure 7.7: Mesh interconnect based on single-driver and unidirectional wires

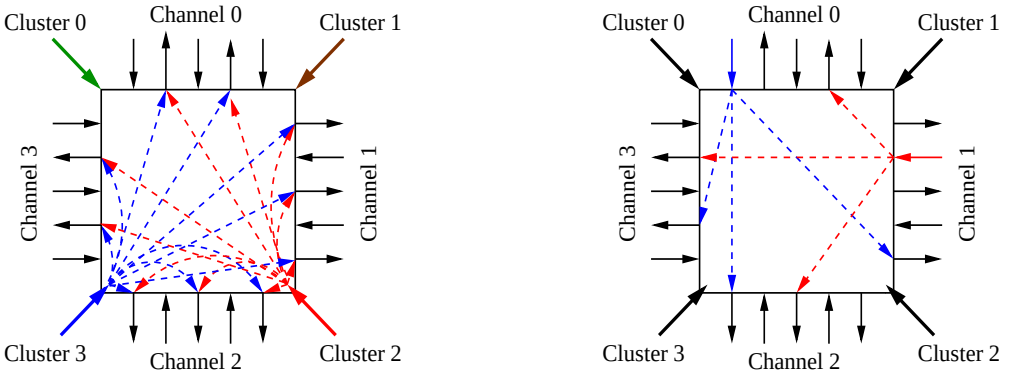
- External interconnect: to vary Mesh interconnect flexibility we change the channel width W.

Mesh with single-driver unidirectional wires

In the Mesh interconnect we use only single-driver unidirectional wires, in fact in [G.Lemieux et al., 2004], authors show that single-driver based interconnect leads to a 25% improvement in area density. Each Mesh cluster is surrounded by 4 channels which are connected by Switch Boxes (SB). We do not use connection blocks in the Mesh to connect channel tracks to cluster inputs and outputs. In fact, as presented in [W.Feng and S.Kaptanoglu, 2007], interconnect is better optimized when the connection block is combined with the cluster local interconnect.
As described in figure 7.7, Mesh cluster input signals are connected to the 4 adjacent channels tracks. Thus, channel width W is given by:

$$W = \frac{Nb_{in}}{4} = \frac{N}{k^{\ell_{in}+1}}$$

Consequently, W depends on the cluster inputs number and is very expensive to modify in terms of routing resources. In fact modifying the channel width induces modification of the cluster interface and consequently the Tree interconnect structure.
A Mesh Switch Box (SB) allows to connect horizontal and vertical channel tracks together and also to clusters outputs. SB inputs come from the 4 channel tracks and the 4 adjacent clusters outputs. Since we use a single-driver based interconnect, each SB output is driven by a multiplexer. SB has a disjoint topology. As presented in figure 7.8-b), input track j of a channel is connected to output tracks j of the other channels. SB also allows to connect Mesh cluster outputs to channels tracks. As illustrated in figure 7.8-a), each cluster output is connected to all switch box outputs located at the 4 sides.



a) SB: Cluster outputs to channel tracks **b) SB: channel tracks to channel tracks**

Figure 7.8: Mesh switch box topology

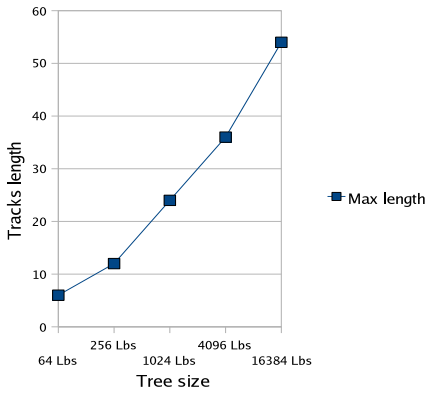


Figure 7.9: Maximum wire lengths depending on Tree size (arity 4)

7.1.3 Track length

New deep-submicron semiconductor technologies involve smaller transistors and wires. This makes transistors faster and wires get slower [E.Lee et al., 2006]. In FPGA architecture design we have to pay attention to long interconnect wires distribution since they behave like an RC transmission line where delay grows quadratically with length.
Considering Tree-based architecture, we notice that when LBs number increases, wires become longer. Unlike for Mesh-based architecture, where wires distance is fixed with no regard to architecture size, in Tree-based architecture when we add levels and increase Tree size, wiring distances increase. In figure 7.9 we show how tracks length increases when the Tree size increases. Tracks length is estimated through the layout generation approach presented in figure 7.11. The length of a wire corresponds to the number of LBs that it spans. In figure 7.10, we show the effect of Tree clusters arity on wires length. We notice that clusters arity increasing induces tracks lengths increasing.
In figure 7.11 we show the layout of a Mesh of Tree tile (Mesh cluster and Mesh switch box). The

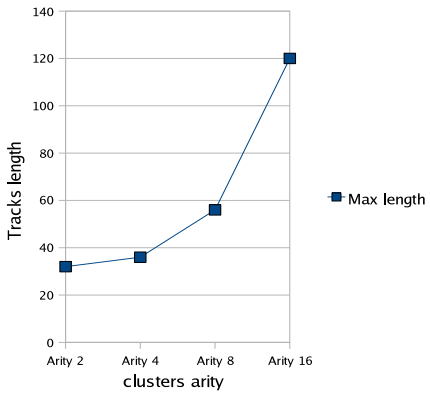


Figure 7.10: Maximum wire lengths depending on Tree clusters arity (4096 LBs)

Length	Fraction
2	29%
3	29%
6	21.8%
12	16.2%
16	4%

Table 7.1: Mesh of Tree track length

Mesh cluster corresponds to a Tree with 256 LBs and $p = 0.88$. In table 7.1 we show the percentages of different routing track lengths. We notice that with such distribution, this architecture presents a good tradeoff between area and speed. In fact, based on the the experimentations shown in [V.Betz et al., 1999], authors expect that the best FPGA architectures include some wires shorter and some wires longer than length 8.

7.2 Configuration flow

Now we present the different steps to implement a netlist on the proposed Mesh of Tree architecture. The placement of a netlist on this architecture is run in two stages:

• The Mesh stage, where clusters are considered as black boxes with i inputs and j outputs. The initial netlist is partitioned into N parts where N corresponds to Mesh clusters number. We obtain N independent sub-netlists and an external netlist describing communication between clusters. The external netlist is used to place clusters on the 2-D Mesh grid.

• The Tree stage, where every sub-netlist (cluster internal netlist) is partitioned individually to define instances positions on the Tree.

A Mesh of Tree architecture can be set flat in the routing phase or can be separated into Mesh and Tree levels. Flat routing consists in routing the circuit netlist considering the total Mesh of Tree

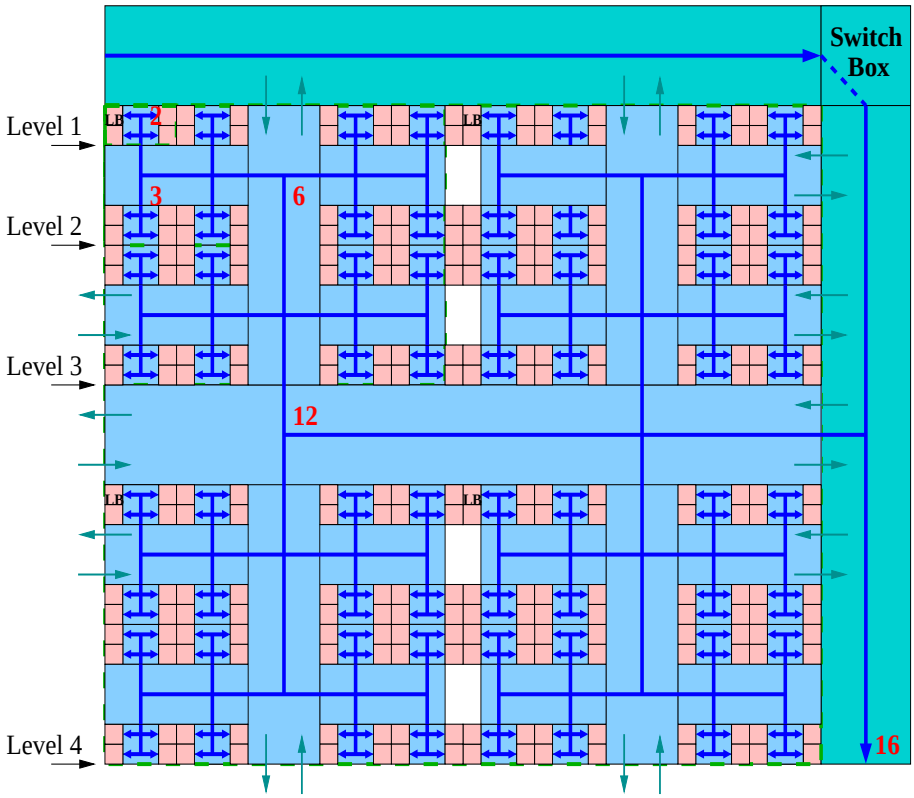


Figure 7.11: Layout view of Mesh of Tree basic tile

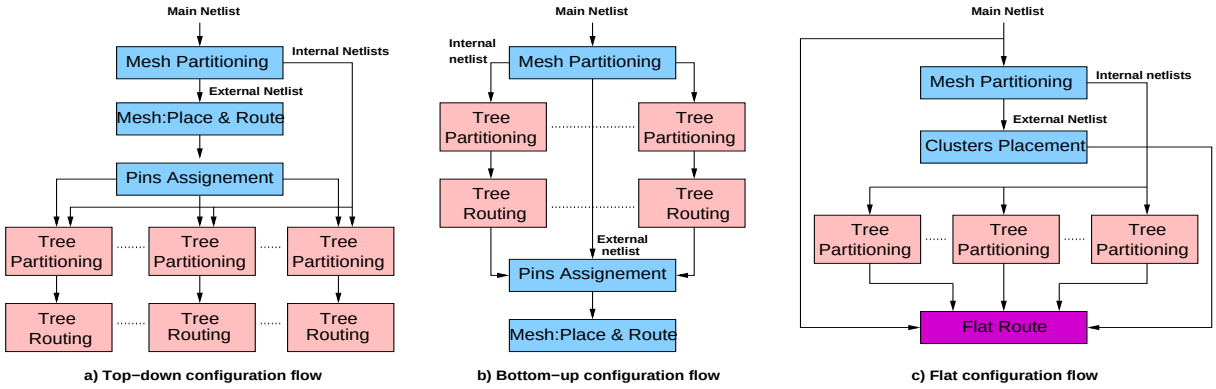


Figure 7.12: Top-down Mesh of Tree configuration flow

resources. Separating Mesh and Tree routing means to consider clusters interface as a boundary between both levels. The order in which we route both levels has an impact on the final result, thus we distinguish three different routing strategies:

- The top-down routing approach: The complete configuration top-down flow is described on figure 7.12-a). First, the external netlist (inter-clusters) is routed on the Mesh; consequently, input and output signals in clusters interfaces are assigned. Then, every internal netlist (intra-clusters) is placed and routed separately respecting the external signals assignment.

- The bottom-up routing approach: The complete configuration bottom-up flow is described on figure 7.12-b). First, internal netlists (intra-clusters) are routed separately. Consequently, input and output signals in clusters interfaces are assigned. Then, the external netlist is routed on the mesh respecting the external signals assignments.

- The flat routing approach: The complete configuration flat flow is described on figure 7.12-c). The total Mesh of Tree routing resources are first presented by a single routing graph. Then all initial netlist signals are routed on this graph.

7.2.1 Mesh partitioning

The purpose of the partitioning step is to distribute netlist instances between the N Mesh clusters (sub-domains) in order to reduce external communication (cut) and congestion. Since we have a balanced Mesh interconnect (the same width is used for all channels), it is mandatory to match cluster in/out resources and worthwhile to spread the congestion over all the interconnect. To address this problem we used the multi-objective function presented in chapter 4, in which the cut and MED (Max External Degree) are taken into account. After main netlist partitioning, we obtain an inter-clusters netlist and N intra-cluster netlists.

7.2.2 Mesh placement

To place clusters on the Mesh 2-D grid, we use the inter-clusters netlist to evaluate signals bounding boxes cost. A simulated annealing strategy is used to optimize the total wire length. As presented in figure 7.13, the way to evaluate signals bounding boxes depends on the configuration strategy. Figure 7.13-a) shows the case where clusters pins positions were assigned by intra-clusters routing and are taken into account in the cost function. Each pin has a position on the 2-D grid and the bounding box contains all pins connected to the specific signal. In figure 7.13-b), the Bounding box function considers clusters positions since pins positions are not defined.

7.2.3 Top-down pins assignment

The logical boundary between inter and intra-clusters levels is defined by the Mesh clusters in/out pins. After completing the inter-clusters netlist routing, the clusters in/out pins are assigned to specific signals. Those signals are considered as in/out pads in the generated intra-

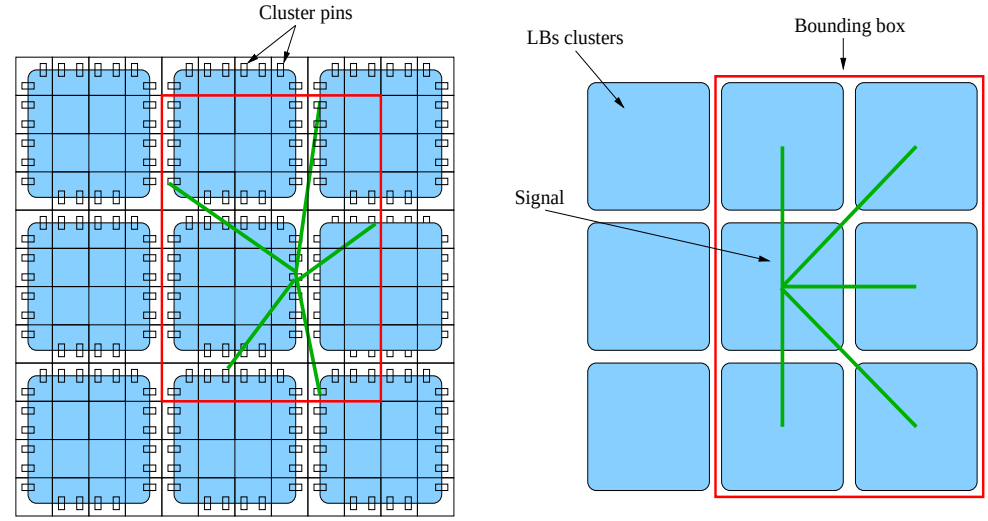


Figure 7.13: Bounding box function evaluation

cluster netlists. Thus, as presented in figure 7.12-a), external netlist routing assigns a specific position to each internal netlist pad (pin assignment). Those positions are strictly respected when we place and route an internal netlist on the Tree-based architecture. In order to handle this point in the Tree partitioning process, these pins are considered as fixed vertices and their signals are taken into account in the cut objective function evaluation.

7.2.4 Bottom-up pins assignment

As we do not have a full crossbar inside Mesh clusters, inputs and outputs located at different clusters cannot be considered as logically equivalent. As presented in figure 7.12-b), each sub-netlist is partitioned and routed separately. Sub-netlist inputs/outputs are assigned to specific cluster inputs and outputs pins. This new ordering is back annotated in the clusters interfaces of the inter-clusters netlist. The pins ordering constraint is very penalizing in the inter-clusters netlist routing. To alleviate the effect of this penalty, we propose the following actions:

- As presented in figure 7.13-a), we consider pins positions (and not cells positions) to evaluate signals bounding boxes in the placement phase.
- We take advantage of equivalence between input/output pins located in the same cluster. In fact, as presented in figure 7.3, all 4 inputs and all 4 outputs are grouped into Tree clusters and are respectively connected to the same UMSB and DMSB. In this case the router can process pins located in the same Tree cluster as logically equivalent and keeps freedom to route any one of them without degrading the intra-clusters routing performed before.

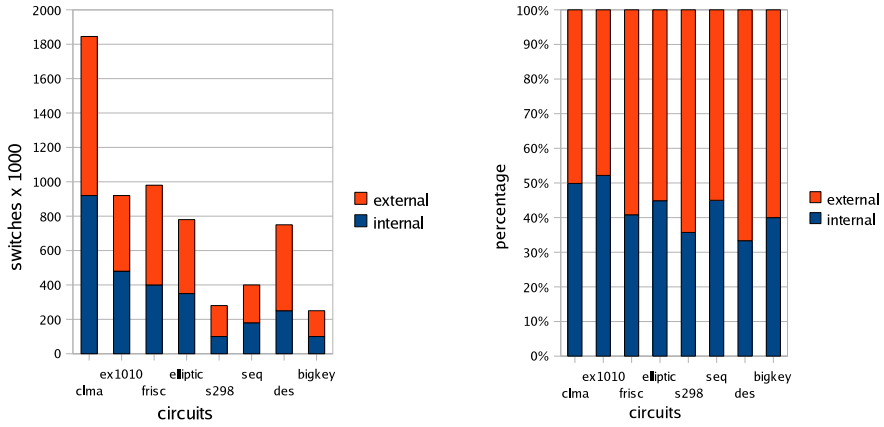


Figure 7.14: Interconnect distribution in Mesh of Tree architecture: Bottom-up approach

7.2.5 Routing

In all cases we generate a routing graph to describe the routing resources. In the flat routing case we use only one routing graph describing the total resources and in the other cases we use two routing graphs, the first describes the Mesh routing resources and the second describes the Tree routing resources. In all cases we use the $Pathfinder$ algorithm to route signals. In the bottom-up case, Tree routing defines Mesh clusters boundary and Mesh routing must comply with it. In the top-down case, the Mesh routing takes clusters input and output pins as logically equivalent (every input can reach all Tree LBs and every output can be reached by all LBs) and defines their positions and consequently the Tree routing must be consistent with these positions.

7.3 Experimental results

First, we evaluate the various configuration approaches on architecture area optimization. The idea is to compare the required switches number to implement a circuit with every routing approach. Next, we compare the Mesh of Tree architecture to the VPR-based Mesh and the Tree-based architectures.

7.3.1 Mesh of Tree: top-down vs. bottom-up

Here we compare the different configuration approaches. In all cases, we use clusters containing 256 LBs and we compare the required architecture characteristics to implement the same circuits. We consider a Mesh architecture with bidirectional wires since it provides flexibility in the Mesh channel width (not the case of unidirectional Mesh architecture). Switch blocks have $disjoint$ topology. The connection block flexibility is $Fc_{in} = 0.5$ and $Fc_{out} = 0.25$.
In all cases, we vary channel width and Tree Rent's parameter to match the smallest architecture to route a particular circuit.

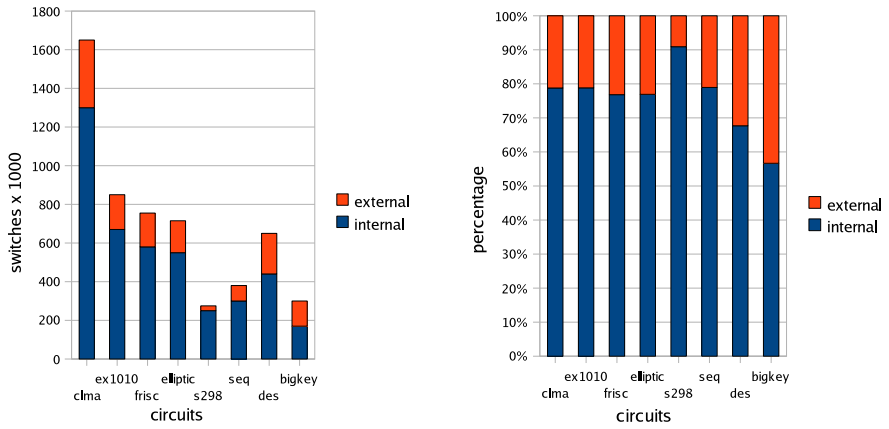


Figure 7.15: Interconnect distribution in Mesh of Tree architecture: Top-down approach

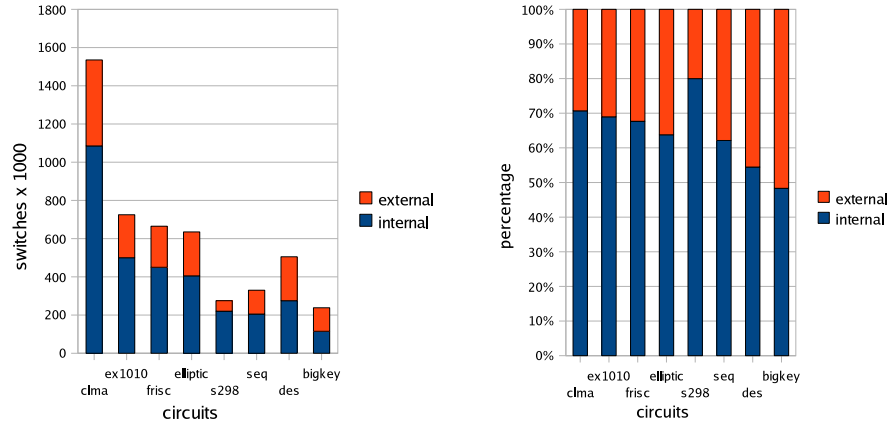


Figure 7.16: Interconnect distribution in Mesh of Tree architecture: Flat approach

With the top-down approach we reduce the required Mesh channel width. The external interconnect switches reduction can be seen in figure 7.14 and figure 7.15. External switches are reduced by 58%. Nevertheless, intra-clusters interconnect (Tree level) were increased to get more flexibility to satisfy the pins assignment constraints. Consequently, the internal switches number is increased by 43%. Using the top-down approach, we have achieved on average a total switches reduction of 14%. As illustrated in figure 7.16, the most optimized architecture is the one generated by the flat routing approach. This seems obvious since this technique has larger routing search space. Nevertheless it requires 4 to 7 times more CPU run time compared to the other approaches.

7.3.2 Mesh of Tree vs. VPR-style Mesh

To evaluate the proposed architecture and tool performances, we place and route the 3 largest MCNC benchmark circuits and the ava circuit which is the largest design containing only LUTs. We consider as references the optimized cluster-based (VPR-style) Mesh and the MFPGA architectures. We map the 4 largest benchmark circuits on the Mesh of Tree architecture. We consider an architecture with unidirectional wires and Mesh clusters size equal to 256 LBs. Every cluster has 256 inputs and 64 outputs equally distributed on the 4 sides. This is obtained by putting input clusters at Tree level 0 and output clusters at level 1. As shown in table 7.2, for every benchmark circuit we adjust only the Mesh clusters array size. We do not tailor the interconnect flexibility to every circuit. The Mesh channel width is equal to 64 and Tree signals growth rate p is equal to 0.88. The Mesh of Tree switches requirement and its distribution between Tree and Mesh levels is presented in figure 7.17. As shown in figure 7.18, we notice that, compared to the VPR-based Mesh architecture, total area is reduced by 42%. This is due essentially to the depopulated intra-cluster crossbar. In fact with $p = 0.88$ the Tree required switches number is equal to 20×10^3 switches only.
We also notice that, compared to a stand-alone Tree, the total area is increased by 28%. This increase is compensated by the Mesh of Tree layout generation simplicity and wires length reduction, compared to stand-alone Tree, especially when we target large circuits sizes. We also, notice that in the Mesh of Tree architecture, the use of Mesh interconnect with unidirectional wires leads to 20% area saving compared to the use of bidirectional wires (figure 7.18).

7.4 Conclusion

We notice that the Tree-based architecture is the most optimized architecture in terms of switches requirement. Nevertheless, this Tree-based architecture, in stand-alone mode, is very penalizing in terms of physical layout generation; it does not support scalability and does not fit with a planar chip structure, especially for large circuits. Conversely, the Mesh of Tree has a good physical scalability: once the cluster layout is generated we can abut it to generate Mesh layouts with the desired size and shape factor. The proposed Mesh of Tree architecture is a good tradeoff between area density and layout scalability.

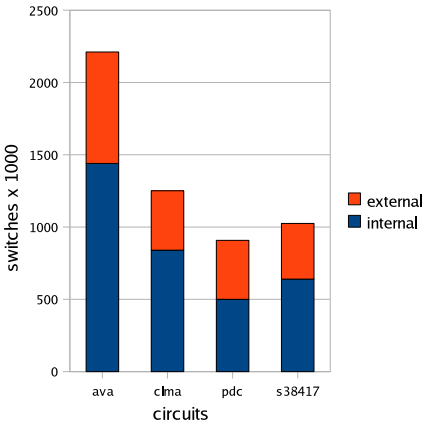


Figure 7.17: Interconnect distribution in Mesh of Tree architecture

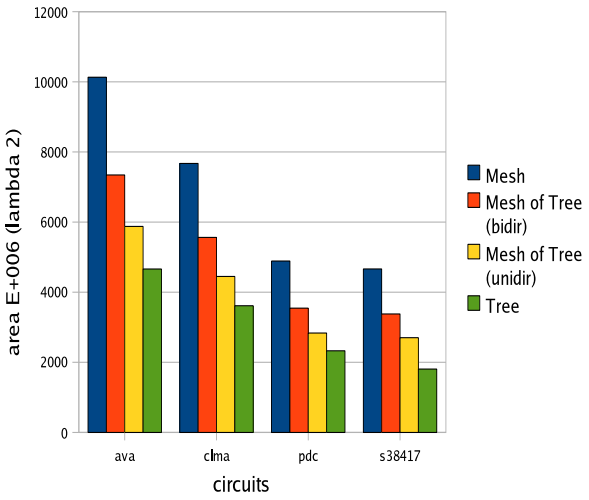


Figure 7.18: Comparison of various FPGA architectures areas

Benchmark circuits	Array	Mesh W	Cluster size	Tree p	External SW $\times 10^3$	Internal SW $\times 10^3$	Total SW $\times 10^3$
ava	9×8	64	256	0.88	771	1440	2211
clma	6×8	64	256	0.88	411	840	1251
pdc	5×5	64	256	0.88	408	500	908
s38417	8×4	64	256	0.88	385	640	1025
average		64	256	0.88	493	855	1348

Table 7.2: Mesh of Tree: switches requirement

This chapter presents an introduction to the exploration of Mesh of Tree architecture. There is still room to optimize it and to investigate the effect of Mesh cluster sizes, Mesh switch boxes topology and the best tradeoff between internal and external interconnect populations.

Conclusion and Future Lines of Research

1 Conclusion

The subject of this dissertation was the exploration of new interconnect topologies and architectures that may play important roles in FPGA performances improvement. The main architectures under exploration were Tree-based and Mesh-Based. Mesh is the most common architecture in academic and industrial fields. Much research effort was deployed to improve architecture performances in terms of area, speed and power dissipation. Modern Mesh architectures have depopulated interconnect, different wire lengths and well adapted algorithms to optimize circuit implementation. Despite its good properties, Tree-based architecture has been overlooked up to now, and there was still room to optimize it. First, we proposed a simple Tree-based architecture and optimized it progressively in terms of density. The major features we used to depopulate interconnect are clusters signals bandwidths and switch blocks topologies. Then, we compared the resulting architecture to Mesh based-topology in terms of performance and area requirement. Finally, we took advantage of both topologies strong points, and we proposed an architecture unifying Mesh and Tree interconnects to get good area efficiency and layout scalability. To compare different architectures we used an adaptive configuration platform. The largest MCNC benchmark circuits were placed and routed using interconnects resources and required area is evaluated based on switches counting and cells areas summation. The following remarks were retained along various architectures exploration:

Architecture and tools interaction:

In chapter 4 we showed that interconnect predictability is an interesting feature that can be taken into account in the placement phase to reduce congestion. MFPGA architecture presented in this chapter had a poor interconnect and much effort was done by configuration tools to deal with routability. It was shown that using poor interconnect but making much effort in the placement phase does not produce efficient solution and does not lead essentially to an optimized architecture. FPGA designers must think about flexibility when interconnect topology is defined. With the improved MFPGA architecture proposed in chapter 5 the algorithmic effort was reduced at the cost of adding some resources. We showed that by reducing external communication in the partitioning phase, added resources were compensated by reduction of signals bandwidth. The resulting architecture presents a good tradeoff between flexibility and density.

Density efficiency:

In chapter 5 we proposed an architecture with depopulated switch boxes and reduced signals

bandwidth. We showed that with such architecture we can obtain a better density compared to Mesh. We achieved a gain of 56% in terms of area. This confirms that we can balance interconnect and logic blocks utilization by logic occupancy decreasing and congestion spreading. In fact despite increasing logic blocks area by 20% we reduced the total area by more than 2 times. Our idea consisted in leaving some logic blocks unused to create white spaces to spread congestion and efficiently exploit interconnect structure which accounts for 75% of the total area (in the case of MFPGA).

We also took advantage of communication locality of designs thanks to hierarchy. In fact, we controlled clusters signals bandwidth at each hierarchical level separately. This was an important feature since designs Rent's parameters depend on the partitioning level and congestion is not the same over all levels. This is similar to the use of a Mesh architecture with various channel widths in different regions. The problem in the case of a Mesh is how to consider interconnect heterogeneity in the placement phase. In the case of Tree-based architecture we showed that the distribution of Rent's parameters has an impact on area and performance characteristics. For example with high Rent's parameters in the lowest levels and low parameters in the highest levels we get good performance at the cost of area increase. The opposite case has the opposite effect.

Clusters arity has an important impact on area. In fact this feature allows to control multiplexers sizes. We showed that the smaller clusters arity is, the smaller multiplexers are and consequently switches number and area are reduced. We also showed that LUT size has an impact on area and that LUT with 4 inputs provides the best density. We proposed an architecture with an interconnect having a single driver at starting point of each wire. Instead of tri-states, each driver has a multiplexer to select from many possible sources. This organization resulted in unidirectional wires. The benefit of using unidirectional wires is the elimination of bidirectional buffering and tri-states and consequently area reduction, performance improvement and power dissipation reduction. In chapter 7, we proposed two Mesh of Tree based interconnects. In both cases Tree interconnect has unidirectional wires. We showed that by using a Mesh interconnect with unidirectional wires, we save area by 20% compared to using a Mesh interconnect with bidirectional wires.

Performance efficiency:

We proposed a Tree-based interconnect organization to take advantage of designs locality and exploit local resources to route signals. This feature was taken into account in the partitioning phase which aimed at reducing external communication between clusters. In addition, as we showed in chapter 7, Tree based architecture has a good wire lengths distribution. This is an interesting feature since minimizing the interconnect delay, always, requires technology dependent tradeoffs between number of switches and the length of wires runs. Clusters arity and LUT size have also an important impact on paths delays. We showed that increasing clusters arity and LUTs size decreases the number of switches crossed by the critical path. This behavior is explained by the decrease of the number of LUTs and clusters in series on the critical path.

Power dissipation efficiency:

According to the work presented in [F.Li et al., 2005], interconnect power is dominant and leakage power is significant in nanometer technologies. Thus, reducing interconnect switches leads

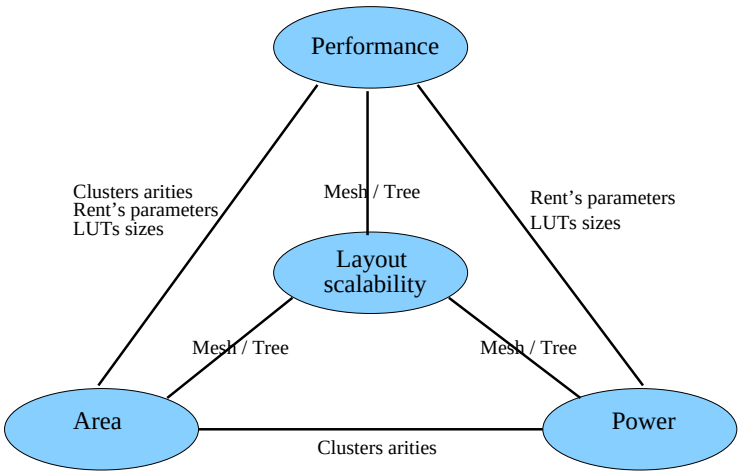


Figure 1: FPGA architectures tradeoffs

to power consumption reduction. To evaluate dynamic power one needs to estimate circuit activity. Since we do not have such feature in our exploration platform we were interested to take care about buffers and SRAMs points numbers since they are the major factor behind static power dissipation. We showed that buffers number can be controlled through clusters arities. When arity increases, multiplexers get larger but their number is reduced and consequently total buffers required at the output of every multiplexer are reduced. We also showed that architecture with LUTs size equal to 4 provides the smallest number of buffers and SRAM points, and consequently the minimum leakage energy.

Layout scalability:

As we showed in chapter 7, Tree layout generation is not adapted to planar shape. Each added level interconnect has its specific design and there is no scalability. The proposed Mesh of Tree architecture represents a good tradeoff between density and layout generation. In fact a single cluster with its Tree local interconnect and its neighboring routing channels is designed; when creating a complete architecture this single tile is replicated.

In figure 1, we show some tuning factors to balance different tradeoffs. For example, architecture Rent's parameters and clusters arity can be determined to optimize area; however this may induce a bad effect on speed performances. In the same way, increasing arity has a good effect on speed and static power dissipation but may induce area increase. We also showed that LUT sizes increase improves speed performance but increases area. Design choice to enhance area, performance or power depends on application domain. Thus, it is difficult for a single FPGA family to cope with different market needs. This is confirmed by to day's industry practice. Vendors have moved to provide different FPGA families to comply with different requirements. It is now common for FPGA manufacturers to offer a high end, high performance family [D.Lewis and al, 2003] [Virtex, 5] and a lower cost, lower performance family [Cyclone, 3] [Spartan, 3e].

2 Future work

The presented work has tackled different aspects of FPGA interconnect optimization. Diverse research directions have emerged from the type of problems explored. In particular, the following points seem interesting to investigate:

Logic blocks heterogeneity

The aim of this thesis was to optimize interconnect in order to reduce area and to improve performance. Interconnect is the major but not the unique factor to improve FPGA performance. In fact we showed that LUT size influences density, speed and power. The use of architectures composed of LUTs with various sizes and associated to logic block with specific functions (adders, multipliers ...) deserves exploration. Most industrial FPGAs contain an increasingly larger number of hard macro blocks. These macro blocks can include embedded memories, adders and multipliers. Figure 2 shows an example of a Tree-based architecture with heterogeneous logic blocks. Using such specific blocks may reduce flexibility but improves performances and density.

Clusters size effect on Mesh of Tree

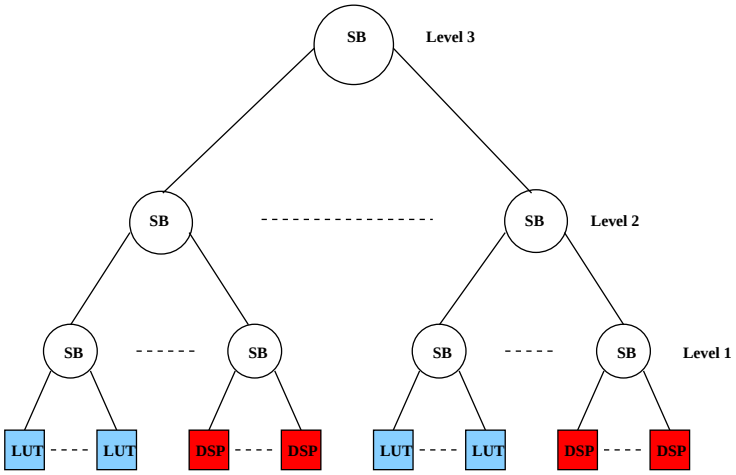


Figure 2: Tree-based architecture with heterogeneous logic blocks

Mesh of Tree has a good layout scalability but it is penalizing in terms of area compared to a stand-alone Tree-based architecture (MFPGA). Two points deserve to be explored: 1) clusters size effect on area: in fact in this thesis we evaluated an architecture with 256 LBs per cluster. Varying cluster size has an impact on the ratio between interconnect at Mesh and Tree levels. 2) Mesh switch boxes topologies: in this thesis we used Mesh switch boxes with $disjoint$ topology. Using a topology similar to the one used inside clusters (Tree-based) may induce better density.

Tree layout generation

Layout generation is very important to get accurate timing and area characteristics. In fact the exact wire lengths in the Tree topology can only be determined by layout. This is important to determine buffers number exactly, their positions and consequently delays. In addition it is often

seen in the custom design world, that there are a range of logically equivalent but electrically distinct implementations [J.M.Rabey, 1996]. We believe the same holds true for FPGA circuits and it is very interesting to explore the range of area and delay tradeoffs that are possible in the design of an FPGA architecture by varying both its logical architecture (LUTs size, clusters size, Rent's parameters ...) and its electrical implementation.

Congestion aware partitioning

As shown in chapter 6, architecture Rent's parameters are larger than circuits partitioning Rent's parameters. This is due essentially to the depopulated interconnect and presence of congested regions. Thus, by reducing congestion and spreading it over all the interconnect we can narrow Rent's parameters gap and obtain a much optimized area. Reducing only external communication is not sufficient and a congestion aware partitioning becomes obvious. To predict congestion in the partitioning phase we propose two solutions: 1) To use a congestion prediction function, for example it was shown in [A.Pandit and A.Akoglu, 2007] that considering ISPL (Intrinsic Shortest Path Length) [A.B.Kahng and S.Reda, 2005] in the clustering phase reduces the required channel width to route circuits on a Mesh-based architecture. 2) To take advantage of Tree-based interconnect predictability. In fact there is a single predictable path connecting a source LB to a sink LB crossing a specific DMSB. This property can be exploited to alleviate considerably the detailed search in the routing phase, thus reducing the required CPU run time. In this way a technique predicting routing conflicts in the partitioning phase, similar to the one proposed in [A.Sharma et al., 2005], can be used with reduced run time penalties.

List of Publications

Some of the following publications can be downloaded from: http://www-asim.lip6.fr/publications/

- FPGA Interconnect Topologies Exploration
  Marrakchi Zied, Mrabet Hayder, Mehrez Habib
  International Journal of Reconfigurable Computing, Selected Papers from ReCoSoc'2008
- A New Coarse-Grained FPGA Architecture Exploration Environment
  Pavez Husain, Marrakchi Zied, Mehrez Habib
  IEEE International Conference on Field Programmable Technology (ICFPT'2008), Taipei, Taiwan, To appear in December 2008
- Enhanced Methodology and Tools for Exploring Domain-specific Coarse Grained FPGAs
  Pavez Husain, Marrakchi Zied, Mehrez Habib
  IEEE International Conference on Reconfigurable Computing (Reconfig'2008), Cancun, Mexico, To appear in December 2008
- The effect of LUT and Cluster Size on Tree-based FPGA Architecture
  Umer Farooq, Marrakchi Zied, Mrabet Hayder, Mehrez Habib
  IEEE International Conference on Reconfigurable Computing (Reconfig'2008), Cancun, Mexico, To appear in December 2008
- Optimized Local Interconnect for Cluster-based Mesh FPGA Architecture
  Marrakchi Zied, Mrabet Hayder, Mehrez Habib
  IEEE International Conference on Microelectronics(ICM'2008), Sharjah, UAE, To appear in December 2008
- Efficient Tree Topology for FPGA Interconnect Network
  Marrakchi Zied, Mrabet Hayder, Amouri Emna, Mehrez Habib
  ACM Great Lakes Symposium on VLSI (GLSVLSI'2008), Orlando, Florida, USA, May 2008, pp. 321-326
- Performances Comparison between Multilevel Hierarchical and Mesh FPGA Interconnects
  Marrakchi Zied, Mrabet Hayder, Masson Christian, Mehrez Habib
  International Journal of Electronics, Jan. 2008, vol. 95, num. 3, pp. 275-289, Taylor and Francis

- Efficient Mesh of Tree Interconnect for FPGA Architecture
  Marrakchi Zied, Mrabet Hayder, Masson Christian, Mehrez Habib
  International Conference on Field-Programmable Technology (ICFPT'2007), Kitakyushu, JAPAN, December 2007, pp. 269-272

- Mesh of Tree: Unifying Mesh and MFPGA for Better Device Performances
  Marrakchi Zied, Mrabet Hayder, Masson Christian, Mehrez Habib
  1st ACM/IEEE International Symposium on Networks-on-Chip (NoC'2007), Princeton, USA, May 2007, pp. 243-252

- A Routability Driven Partitioning and Detailed Placement Approach for Multilevel Hierarchical FPGA (Poster)
  Marrakchi Zied, Mrabet Hayder, Souffleteau Gregory, Masson Christian, Mehrez Habib
  ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'2007), Monterey, Californie, USA, February 2007, pp. 225-225

- Implementation of Scalable Embedded FPGA for SOC
  Mrabet Hayder, Marrakchi Zied, Mehrez Habib, Tissot Andre
  IEEE International Conference on Design & Test of Integrated Systems in Nanoscale Technology (DTIS'2006), Tunis, Tunisia, September 2006, pp. 74-77

- Performances Improvement of FPGA using Novel Multilevel Hierarchical Interconnection Structure
  Mrabet Hayder, Marrakchi Zied, Souillot Pierre, Mehrez Habib
  IEEE/ACM International Conference on Computer-Aided Design (ICCAD'2006), San Jose, California, USA, November 2006, pp. 675-679

- Evaluation of Hierarchical FPGA partitioning methodologies based on architecture Rent Parameter
  Marrakchi Zied, Mrabet Hayder, Mehrez Habib
  2nd IEEE Conference on Ph.D. Research in MicroElectronics and Electronics (PRIME'2006), Otranto, Italy, June 2006, pp. 85-88

- A multilevel hierarchical interconnection structure for FPGA (Poster)
  Mrabet Hayder, Marrakchi Zied, Souillot Pierre, Mehrez Habib
  14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'2006), Monterey, California, USA, February 2006, pp. 225

- Configuration tools for a new multilevel hierarchical FPGA (Poster)
  Marrakchi Zied, Mrabet Hayder, Mehrez Habib
  14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'2006), Monterey, California, USA, February 2006, pp. 229

- A new Multilevel Hierarchical MFPGA and its suitable configuration tools
  Marrakchi Zied, Mrabet Hayder, Mehrez Habib

IEEE Computer Society Annual Symposium on Emerging VLSI (ISVLSI'2006), Technologies and Architectures, Karlsruhe, Germany, March 2006, pp. 263-268

- Implementation of Scalable Embedded FPGA for SOC
  Mrabet Hayder, Marrakchi Zied, Mehrez Habib, Tissot Andre
  Reconfigurable Communication-centric SoCs (ReCoSoC'2005), Montpellier, France, June 2005

- Hierarchical FPGA clustering based on multilevel partitioning approach to improve routability and reduce power dissipation
  Marrakchi Zied, Mrabet Hayder, Mehrez Habib
  International Conference on Reconfigurable Computing and FPGAs (ReConFig'2005), Puebla city, Mexico, September 2005

- Hierarchical FPGA clustering to improve routability
  Marrakchi Zied, Mrabet Hayder, Mehrez Habib
  IEEE Conference on Ph.D. Research in MicroElectronics and Electronics (PRIME'2005), Lausanne, Switzerland, July 2005, pp. 139-142

- Automatic Layout of Scalable Embedded Field Programmable Gate Array
  Mrabet Hayder, Marrakchi Zied, Mehrez Habib, Tissot Andre
  International Conference on Electrical Electronic and Computer Engineering (ICEEC'2004), Cairo, Egypt, September 2004, pp. 469-472

Bibliography

[A.Aggarwal and D.M.Lewis, 1994] A.Aggarwal and D.M.Lewis (1994). Routing Architecture for Hierarchical Field Programmable Gate Arrays. *International Conference on Computer design*, pages 475–478.

[A.B.Kahng and S.Reda, 2005] A.B.Kahng and S.Reda (2005). Intrinsic Shortest Path Length: A New, Accurate a Priori Wirelength Estimator. *International Conference on Computer-Aided Design*, pages 173–180.

[A.DeHon, 1996] A.DeHon (1996). Reconfigurable Architectures for General Purpose Computing. *AI Technical Report 1586, MIT Artificial Intelligence Laboratory*.

[A.DeHon, 1999] A.DeHon (1999). Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization). *International symposium on Field Programmable Array FPGA, Monterey, CA*.

[A.DeHon, 2001] A.DeHon (2001). Rent's Rule Based Switching Requirements. *System Level Interconnect Prediction Workshop*.

[A.Dunlop and B.Kernighan, 1985] A.Dunlop and B.Kernighan (1985). A Procedure for Placement of Standard-cell VLSI Circuits. *IEEE Transactions on CAD*, pages 92–98.

[A.Greiner and F.Pecheux, 1992] A.Greiner and F.Pecheux (1992). Alliance: A complete set of CAD tools for teaching VLSI design. *Eurochip Workshop*.

[A.Marquart et al., 1999] A.Marquart, V.Betz, and J.Rose (1999). Using Cluster-based Logic Block and Timing-driven Packing to improve FPGA Speed and Density. *International symposium on FPGA, Monterey*, pages 37–46.

[A.Pandit and A.Akoglu, 2007] A.Pandit and A.Akoglu (2007). Net Length Based Routability Driven Packing. *International Conference on Field Programmable Technology*, pages 225–232.

[A.Sharma et al., 2005] A.Sharma, C.Ebeling, and S.Hauck (2005). Architecture Adaptive Routability-Driven Placement for FPGAs. *International conference on Field Programmable Logic and Applications FPL*, pages 427–432.

[A.Singh and M.Marek-Sadowska, 2002] A.Singh and M.Marek-Sadowska (2002). Efficient Circuit Clustering for Area and Power Reduction in FPGAs. *International Symposium on Field Programmable Gate Arrays*, pages 59–66.

[B.Kernighan and S.Lin, 1970] B.Kernighan and S.Lin (1970). An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Tech. Journal*, 49:291–307.

[B.Landman and R.Russo, 1971] B.Landman and R.Russo (1971). On Pin Versus Block Relationship for Partition of Logic Circuits. *IEEE Transactions on Computers*, 20(1469-1479).

[C.J.Alpert et al., 1997a] C.J.Alpert, L.W.Hagen, and A.B.Kahng (1997a). Multilevel Circuit Partitioning. *Design Automation Conference*, pages 530–533.

[C.J.Alpert et al., 1997b] C.J.Alpert, T.Chan, D.Huang, A.Kahng, I.Markov, P.Mulet, and K.Yan (1997b). Faster Minimization of Linear Wirelength for Global Placement. *ACM Symposium on Physical Design*, pages 4–11.

[C.Leiserson, 1985] C.Leiserson (1985). Fat-trees: Universal Networks for Hardware Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901.

[C.M.Fiduccia and R.M.Mattheyeses, 1982] C.M.Fiduccia and R.M.Mattheyeses (1982). A Linear-time Heuristic for Improving Network Partitions. *Design Automation Conference*, pages 175–181.

[C.Sechen and A.Sangiovanni-Vincentelli, 1985] C.Sechen and A.Sangiovanni-Vincentelli (1985). The Timberwolf Placement and Routing Package. *JSSC*, pages 510–522.

[C.Thompson, 1979] C.Thompson (1979). Area-time complexity for VLSI. *ACM Annual symposium on theory of computing*, pages 81–88.

[Cyclone, 3] Cyclone (3). Cyclone III device handbook, September 2007. .

[D.A.Papa and I.L.Markov,] D.A.Papa and I.L.Markov. Hypergraph Partitioning and Clustering. *Technical Report, University of Michigan, EECS Department*.

[D.Huang and A.Kahng, 1995] D.Huang and A.Kahng (1995). When Clusters Meet Partitions: New Density based Methods for Circuit Decomposition. *IEEE European Design and Test Conference*, pages 60–64.

[D.Huang and A.Kahng, 1997] D.Huang and A.Kahng (1997). Partitioning-based Standard-cell Global Placement with an Exact Objective. *ACM Symposium on Physical Design*, pages 18–25.

[D.J.A.Welsh and M.B.Powell, 1967] D.J.A.Welsh and M.B.Powell (1967). An upper bound for chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86.

[D.Lewis and al, 2003] D.Lewis and al (2003). The Stratix Logic and Routing Architecture. *International Symposium on Field Programmable Gate Arrays*, pages 12–20.

[D.Lewis and al, 2005] D.Lewis and al (2005). The Stratix II Logic and Routing Architecture. *International symposium on Field Programmable Gate Array*, pages 14–20.

[E.Ahmed and J.Rose, 2000] E.Ahmed and J.Rose (2000). The Effect of LUT and Cluster Size on Deep-submicron FPGA Performance and Density. *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 3–12.

[E.Bozorgzadeh and al, 2004] E.Bozorgzadeh and al (2004). Routability-driven Packing: Metrics and Algorithms for Cluster-based FPGAs. *IEEE Journal of Circuits, Systems and Computers*, 13(1):77–100.

[E.Lee et al., 2006] E.Lee, G.Lemieux, and S.Mirabbasi (2006). Interconnect Driver Design for Long Wires in Field-Programmable Gate Arrays. *IEEE International Conference on Field Programmable Technology*, pages 1–8.

[E.M.Sentovich et al., 1992] E.M.Sentovich, K.J.Singh, L.Lavagno, C.Moon, R.Murgai, A.Saldanha, H.Savoj, Stephan, P., R.K.Brayton, and A.Sangiovanni-Vincentelli (1992). SIS: A System for Sequential Circuit Synthesis. *Technical Report No. UCB/ERL M92/41. University of California, Berkeley*.

[F.Aarts et al., 1985] F.Aarts, F.Debont, and E.Habers (1985). Statistical Cooling: A General Approach to Combinatorial Optimization Problems. *Philips Journal*, pages 193–226.

[F.Li et al., 2005] F.Li, Y.Lin, L.Hei, D.Chen, and J.Cong (2005). Power Modeling and Characteristics of Field Programmable Gate Arrays. *IEEE Transactions on Computer Aided Design*, 24(11).

[G.Karypis et al., 1997] G.Karypis, R.Aggarwal, V.Kumar, and S.Shekhar (1997). Multilevel Hypergraph Partitioning: Application in VLSI Design. *Design Automation Conference*, pages 526–529.

[G.Karypis and V.Kumar, 1999] G.Karypis and V.Kumar (1999). Multilevel k-way Hypergraph Partitioning. *Design automation conference*.

[G.Lemieux and D.Lewis, 2002] G.Lemieux and D.Lewis (2002). Analytical Framework for Switch Block Design. *International Conference on Field-Programmable Logic (FPL)*, pages 122–131.

[G.Lemieux et al., 2004] G.Lemieux, E.Lee, M.Tom, and A.Yu (2004). Directional and Single-Driver Wires in FPGA Interconnect. *IEEE International Conference on Field-Programmable Technology*, pages 41–48.

[G.Sigl et al., 1991] G.Sigl, K.Doll, and F.Johannes (1991). Analytical Placement: A Linear or a Quadratic Objective Function? *Design Automation Conference*, pages 427–432.

[I.Kuon and J.Rose, 2007] I.Kuon and J.Rose (2007). Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on CAD*, 26(2):203–215.

[J.Cong and Y.Ding, 1994a] J.Cong and Y.Ding (1994a). FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table based FPGA Designs. *IEEE Transactions on Computer-Aided Design*, pages 1–12.

[J.Cong and Y.Ding, 1994b] J.Cong and Y.Ding (1994b). On Area/Depth Trade-off in LUT-Based FPGA Technology Mapping. *IEEE Transactions on VLSI Systems*, 2(2):137–148.

[J.Cong and Y.Ding, 2000] J.Cong and Y.Ding (2000). Structural Gate Decomposition for Depth-Optimal Technology in LUT-Based FPGA Designs. *ACM Transactions on Design Automation of Electronic Systems*, 5(3).

[J.Cong and Y.Hwang, 1995] J.Cong and Y.Hwang (1995). Simultaneous Depth and Area Minimization in LUT-Based FPGA Mapping. *ACM/SIGDA International Symposium on Field Programmable Gate Array*, pages 68–74.

[J.Frankle, 1992] J.Frankle (1992). Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing. *ACM/IEEE Design Automation Conference*, pages 536–542.

[J.Greene et al., 1993] J.Greene, E.Hamdy, and S.Beal (1993). Antifuse Field Programmable Gate Arrays. *Proceedings of the IEEE*, pages 1042–1056.

[J.M.Rabey, 1996] J.M.Rabey (1996). Digital Integrated Circuits A Design Perspective. *Prentice Hall*.

[J.Pistorius and M.Hutton, 2003] J.Pistorius and M.Hutton (2003). Placement Rent Exponent Calculation Methods, Temporal Behavior and FPGA Architecture Evaluation. *System Level Interconnect Prediction Workshop*, pages 31–38.

[J.Pistorius et al., 2007] J.Pistorius, M.Hutton, A.Mishchenko, and R.Brayton (2007). Benchmarking Method and Designs Targeting Logic Synthesis for FPGAs. *International Workshop on Logic and Synthesis IWLS*, pages 230–237.

[J.Rose et al., 1990] J.Rose, R.Francis, D.Lewis, and P.Chow (1990). Architecture of Field-Programmable Gate Arrays: The Effect of Logic Functionality on Area Efficiency. *IEEE Journal of Solid State Circuits*.

[L.A.Sanchis, 1989] L.A.Sanchis (1989). Multiple-way Network Partitioning. *IEEE Transactions on Computers*, 38(1):62–81.

[Lemieux and Lewis, 2004] Lemieux, G. and Lewis, D. (2004). Design of Interconnection Networks for Programmable Logic. *Kluwer Academic Publishers*, pages 12–20.

[L.Hagen et al., 1994] L.Hagen, A.B.Kahng, F.J.Kurdahi, and C.Ramachandran (1994). On the Intrinsic Rent parameter and Spectra-based Partitioning Methodologies. *IEEE Transactions on Computer Aided Design*, 13(1):27–37.

[L.Hagen and A.Kahng, 1997] L.Hagen and A.Kahng (1997). Combining Problem Reduction and Adaptive Multi-start: A new Technique for Superior Iterative Partitioning. *IEEE Transactions on Computer-Aided Design*, pages 92–98.

[L.McMurchie and C.Ebeling, 1995] L.McMurchie and C.Ebeling (1995). Pathfinder: A Negotiation-Based Performance-Driven Router for FPGAs. *International Workshop on Field Programmable Gate Array*.

[L.Shang et al., 2002] L.Shang, A.S.Kaviani, and K.Bathala (2002). Dynamic Power Consumption in Virtex-II FPGA Family. *International Symposium on FPGA, Monterey*, pages 157–164.

[M.Dehkordi and S.Brown, 2002] M.Dehkordi and S.Brown (2002). The Effect of Cluster Packing and Node Duplication Control in Delay Driven Clustering. *IEEE International Conference on Field Programmable Technology*, pages 227–233.

[M.Garey and D.Johnson, 1979] M.Garey and D.Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. *San Francisco, CA: Freeman*.

[M.Huang et al., 1986] M.Huang, F.Romeo, and A.Sangiovanni-Vincentelli (1986). An Efficient General Cooling Schedule for Simulated Annealing. *ICCAD*, pages 381–384.

[M.Hutton, 2005] M.Hutton (2005). The Design of Modern FPGA Architectures. *International Symposium: The Future of Configurable Hardware, Gent, Belgium*.

[M.Hutton et al., 2001] M.Hutton, K.Adibsamii, and A.Leaver (2001). Timing-Driven Placement for Hierarchical Programmable Logic Devices. *International Symposium on Field Programmable Gate Array*, pages 3–11.

[M.R.Garey et al., 1974] M.R.Garey, D.S.Johnson, and L.Stockmeyer (1974). Some simplified NP-complete problems. *Sixth annual ACM symposium on Theory of computing*, pages 47–63.

[N.Selvakkumaran and G.Karypis, 2006] N.Selvakkumaran and G.Karypis (2006). Multi-objective Hypergraph-Partitioning Algorithms for Cut and Maximum Subdomain-Degree Minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 25(3):504–517.

[P.Du et al., 2004] P.Du, G.W.Grewal, S.Areibi, and D.K.Banerji (2004). A Fast Hierarchical Approach to FPGA Placement. *ESA/VLSI*, pages 497–503.

[P.Guerrier and A.Greiner, 2000] P.Guerrier and A.Greiner (2000). A Generic Architecture for on Chip Packet-switched Interconnections. *Proceedings of the Design Automation and Test in Europe Conference 2000 (DATE 2000), Paris, France*, page 250 256.

[R.Hitchcock et al., 1983] R.Hitchcock, G.Smith, and D.Cheng (1983). Timing Analysis of Computer-Hardware. *IBM Journal of Research and Development*, pages 100–105.

[R.Murgai et al., 1991] R.Murgai, R.Brayton, and A.Sangiovanni-Vincentelli (1991). On Clustering for Minimum Delay / Area. *IEEE International Conference on Computer Aided Design*, pages 6–9.

[R.Tessier, 2005] R.Tessier (2005). A reconfigurable, Power-Efficient Adaptive Viterbi Decoder. *IEEE Transactions on VLSI Systems*, 13(4):484–488.

[R.Tessier and H.Giza, 2000] R.Tessier and H.Giza (2000). Balancing Logic Utilization and Area Efficiency in FPGAs. *International workshop on Field Programmable Logic and Applications*.

[S.Brown, 1994] S.Brown (1994). An Overview of Technology, Architecture and CAD Tools for Programmable Lgic Devices. *Custom Integrated Circuits Conference*, pages 69–76.

[S.Kaptanoglu, 2007] S.Kaptanoglu (2007). Power and the Future FPGA Architectures. *International Conference on Field Programmable Technology*.

[S.Kirkpatrick et al., 1983] S.Kirkpatrick, C.D.Gelatt, and M.P.Vecchi (1983). Optimization by Simulated Annealing. *Science 220*, pages 671–680.

[Spartan, 3e] Spartan (3e). November 2006. .

[Stratix, II] Stratix (II). Stratix II FPGAs. *available at http://www.altera.com/products/devices/stratix2/st2-index.jsp*.

[S.Yang, 1991] S.Yang (1991). Logic Synthesis and optimization benchmarks, Version 3.0. *Microelectronics Center of North Carolina (MCNC), Raleigh*.

[T.Bui et al., 1987] T.Bui, S.Chaudhuri, T.Leighton, and M.Sipser (1987). Graph Bisection Algorithms with Good Average Behavior. *Combinatorica*.

[T.Cormen et al., 1990] T.Cormen, C.Leiserson, and R.Rivest (1990). Introduction to Algorithms. *MIT Press, Cambridge*.

[V.Adler and E.G.Friedman, 1997] V.Adler and E.G.Friedman (1997). Repeater Insertion to Reduce Delay and Power in RC Tree Structures. *Conference on Signals, Systems and Computers*, pages 749–752.

[V.Betz et al., 1999] V.Betz, A.Marquardt, and J.Rose (1999). Architecture and CAD for Deep-Submicron FPGAs. *Kluwer Academic Publishers*.

[V.Betz and J.Rose, 1997] V.Betz and J.Rose (1997). VPR: A New Packing Placement and Routing Tool for FPGA Research. *International Workshop on FPGA*, pages 213–22.

[Virtex, 5] Virtex (5). Virtex-5 Multi-Platform FPGA. *Available at http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/*.

[W.E.Donath, 1979] W.E.Donath (1979). Placement and Average Interconnection Lengths of computer Logic. *IEEE Transactions on Circuits and Systems*, pages 272–277.

[W.Feng and S.Kaptanoglu, 2007] W.Feng and S.Kaptanoglu (2007). Designing efficient input interconnect blocks for LUT clusters using counting and entropy. *International Symposium on Field Programmable Gate Array*, pages 23–32.

[Y.Lay and P.Wang, 1997] Y.Lay and P.Wang (1997). Hierarchical Interconnection Structures for Field Programmable Gate Arrays. *IEEE Transactions on VLSI Systems*, 5(2):186–196.

[Y.Sanker and J.Rose, 1999] Y.Sanker and J.Rose (1999). Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs. *International FPGA symposium*.

[Z.Marrakchi et al., 2005] Z.Marrakchi, H.Mrabet, and H.Mehrez (2005). Hierarchical FPGA Clustering to Improve Routability. *Conference on Ph.D Research in Microelectronics and Electronics, PRIME*.