Fast Placement and Floorplanning Methods in Modern Reconfigurable FPGAs

Doctoral dissertation submitted by

Pritha Banerjee

for award of the Ph.D. degree of the

Indian Statistical Institute, Kolkata

Advisor :

Professor Susmita Sur-Kolay

Advanced Computing and Microelectronics Unit
Indian Statistical Institute
203, B. T. Road, Kolkata, India 700 108
June 2009

Dedicated to

my parents

Acknowledgement

Although a thesis is known by the name of its author, there are many without whose direct or indirect help, a thesis does not materialize. It is time to thank them.

First and the foremost, I am profoundly grateful to my supervisor Dr. Susmita Sur-Kolay for her constant motivation, guidance and advices throughout this work with patience.

I would like to express my gratitude towards Prof. Subhas C. Nandy, Dr. Sandip Das, Dr. Arijit Bishnu and Subasis Bhattacharjee for the numerous technical discussions we had and their motivation during this period. I thank my co-authors Megha Sangtani and Debasri Saha.

I would like to thank Indian Statistical Institute for providing me the financial support through fellowship and job assignments, which made this work successful. My thank goes to all the members of the faculty, office staff and my colleagues of Advanced Computing and Microelectronics Unit for making this unit a cordial and wonderful place to work.

I consider myself to be fortunate to have met my friends Partha, Sasanka, Subhasis, Debasis, Sandeep, Arindam, Gautam, Buddha da, Nilanjana, Sruti, Sahadeb, Sudip and many others, whose support, encouragement and willingness to listen made things bearable during difficult times and delightful the rest of the times during these years.

I would like to give a special acknowledgement to my husband, without whose help, patience and encouragement I would not have completed this work. I am also thankful to my parents-in-law for their support and encouragement during these years. Last but not the least, I thank my Grandmother, Mother, Father, Brothers - Partho, Argho, and my aunts Manju Chatterjee and Late Gopa Mukherjee for their support, encouragement, patience, faith and love which made this work a success.

(Pritha Banerjee)

June 1, 2009

Advanced Computing and Microelectronics Unit,
Indian Statistical Institute,
203 B T Road, Kolkata
Pin 700 108, India

Contents

List of Tables

List of Figures

List of Symbols

κ	maximum number of shapes generated for a module
ρ_m	resource requirement vector of module m
σ_L	left super module
σ_R	right super module
A	basic tile
b^c_{ij}	number of common blocks in cones τ_i and τ_j
b_i	any block CLB or IOB
BB_i	bounding box of net i
C	Set of CLBs
c_i	a CLB
d	maximum number of terminals in a net
D^b	block netlist graph
DM_i	set of dynamic modules in instance I_i
G^a	cone adjacency graph
G^b	bi-partite graph for IOB placement
G^o	flow graph for reallocation of overloaded regions
G_c	netlist graph for clustering
g_f	a group, consisting of one slicing tree for each instance in a task scedule
H	height of FPGA chip in terms of CLB height
$H = (V, E)$	hypergraph representing netlist of CLBs or modules
H_t	height of FPGA chip in terms of basic tile
I_i	i^{th} instance of a task schedule
k	number of signal nets
l^b_i	level of a block in cone τ_i
M	set of modules
m	module
m^{ram}_i	RAM requirement of a module

n	number of CLBs or modules
n_i	number of dynamic modules in instance I_i
n_{pi}	number of primary inputs
n_{po}	number of primary outputs
O_i	*linear arrangement* of subgraph SG_i
p	number of primary input and output blocks (IOB)
p^i	a primary input block
p^o	a primary output block
p_i	an IOB
q	number of instances in a task schedule
R_i	rectangular region assigned to a module m_i
S	set of signal nets
S_i	a signal net
SM	set of static modules among q instances in a task scedule
T_h	height of floorplan in terms of basic tile
T_m	tile required by module m
T_w	width of floorplan in terms of basic tile
t_{ij}	j^{th} slicing tree of i^{th} instance
W	width of FPGA chip in terms of CLB width
W_i^c	width of cone τ_i
W_{ij}^o	width of overlap of cones τ_i and τ_j
w_A	width of basic tile
W_t	width of FPGA chip in terms of basic tile
w_{ij}	weight of edge (v_i, v_j) in G^a
$w_{ij}(b)$	weight of edge (v_i, v_j) due to block b in G^a
δ_i	a dynamic module
η_i	number of slicing trees generated in instance I_i
σ_i	a static module
τ^i	input cone
τ^o	output cone

CHAPTER 1

Introduction

Contents

1.1 FPGA

Field-programmable gate-arrays (FPGA) are programmable hardware platforms with pre-fabricated logic and interconnects, which are electrically programmed by the user to realize a variety of circuits frequently required in a wide range of applications. Unlike *application-specific integrated-circuits* (ASICs), where realization of a circuit design takes several man-hours and enormous effort, the pre-fabricated logic and interconnects can be quickly programmed according to the design specification and made functional. Thus, in contrast to the ASICs, FPGAs can be customized and reconfigured depending on the need of the user. A basic FPGA chip consists of a set of *configurable logic blocks* (CLB) and interconnects which can be connected by means of transistor switches or *anti-fuses.* Each CLB consists of small memory units in the form of *lookup tables* (LUT) which can be programmed at run-time. To realize a circuit on to FPGA, these LUTs need to be loaded with appropriate functionality in terms of bits at run-time. Given a circuit design, it undergoes several optimization steps [Betz 1999] to get realized on to the FPGA chip. Finally, a *bitstream* of the design is generated that is downloaded on to the FPGA chip. This process of realizing a circuit on to the FPGA chip is called *mapping*. Once the bitstream is loaded on to the chip, the circuit starts functioning. To realize a different circuit on the same chip, one has to download only the corresponding bitstream on to the FPGA chip. Being reconfigurable, the turn-around time for the application is significantly less than realizing an application on ASIC.

FPGAs have experienced an exponential growth in the past twenty years and are increasingly competing with ASICs in medium to low volume market [Wang 2003]. FPGAs were introduced in mid to late eighties with merely 64 *lookup tables* (LUTs) as simple glue logic, whereas modern FPGAs offer up to over two billion *programmable logic cells* along with a large number of macro blocks such as memory, DSP blocks, embedded processors, high speed Input/Outputs and many other pre-placed blocks [Kuon 2007]. The reason for the success of FPGAs is their low *non-recurring engineering* (NRE) costs and reconfigurability. Because of reconfigurability and fast turn-around time, FPGAs are not only used in ASIC prototyping as in earlier days, but also in mission critical applications. Now with millions of logic gates in an FPGA, with shorter design and production time, lower setup cost and risk, it is extensively used in space applications, *digital signal processing* (DSP), software defined radio, aerospace, defense, medical imaging, speech recognition and bio-informatics [Xilinx , Brown 1995, Manimegalai 2007].

In order to take the full advantage of FPGA's reconfigurability, the *mapping* time of a given design on to FPGA chip has to be minimized. The process of mapping is a complex task, involving conflicting objectives to be satisfied during the process. As in ASICs, the mapping of a design on to the FPGA consists of the following design steps: *synthesis*, *technology mapping*, *floorplanning* and *place-and-route*. The problems to be solved for almost all the steps are difficult in nature, and belong to the class of NP-hard problems [Sherwani 1993]. Thus, it is not possible to design polynomial time algorithms to solve the problems optimally. The *computer-aided-design* (CAD) tools play a critical role in obtaining solutions of *high quality* with *efficiency*. Although these design steps seem to be similar to those for the traditional ASIC design, each step of FPGA design has additional constraints and optimization criteria to be satisfied [Wang 2003, Taghavi 2004].

This thesis focuses on the floorplanning and placement step of the FPGA CAD flow. The research in this area aims at either obtaining a very high quality solution for the *physical design problem*, or getting the solution quicker with minimal sacrifice in quality of solution. The objective of this thesis is to develop fast yet efficient placement and floorplanning technique for mapping a given design on to FPGA with minimal sacrifice in the quality of solution in the context of run-time reconfiguration in FPGAs. In this chapter, the basic architecture of FPGAs is presented in Section 1.2 followed by different categories of FPGA architectures in Section 1.3. Section 1.4 discusses the FPGA physical design cycle. The scope of the proposed work is presented in Section 1.5. Finally, the chapter concludes with the organization of the thesis in Section 1.6.

1.2 Components of a basic FPGA chip

The basic component of any FPGA chip is a set of *programmable* or *configurable logic blocks* (CLB) arranged in a two dimensional array with routing wires laid out in *horizontal* and *vertical channels* between rows and columns of CLBs respectively. A CLB (also called *logic blocks* in general) consists of *lookup tables* (LUT), *flip flops* (FF) and/or *multiplexers* for implementation of logic. Figure 1.1 shows a simple earlier generation FPGA architecture which is a widely accepted architecture model used in the FPGA research community [Betz 1999, Emmert 1999b, Chang 2000, Vicente 2004, Maidee 2005, Vorwerk 2009]. The *input/output blocks* (IOB) are located around the periphery of the chip, providing programmable I/O connections and support for various I/O standards [Anderson 2000]. Each CLB is surrounded by *routing channels* connected through *switch blocks* and *connection blocks*. The wires in the channel are segmented, and are of varying length. Commercial FPGA chips have many dedicated interconnects of different lengths which are point-to-point and unidirectional. A *switch block* connects wires in adjacent channels through programmable switches such as *pass-transistors* or *bi-directional buffers*, and is represented as a matrix of possible connections. A *connection block* connects the wire segments in a channel to the input and output pins of a CLB using *programmable switches*. All the programmable switches are identical. While programming, the appropriate switches are turned on or off. Figure 1.1 shows the internal configuration of a part of an FPGA chip.

The routing architecture of an FPGA is defined by *channel width* W_c, *switch block flexibility* F_s, *connection block flexibility* F_c and segmented wire length. The *channel width* defines the number of wires laid out and pre-fabricated in the channel of an FPGA chip. The *switch block flexibility* is the number of wires in each channel

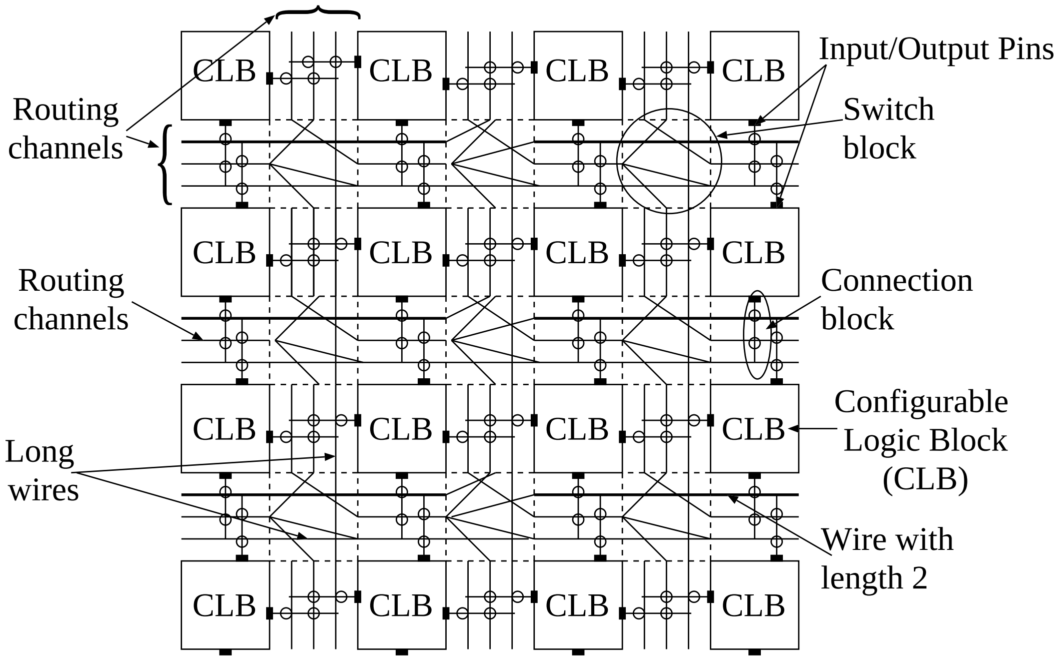
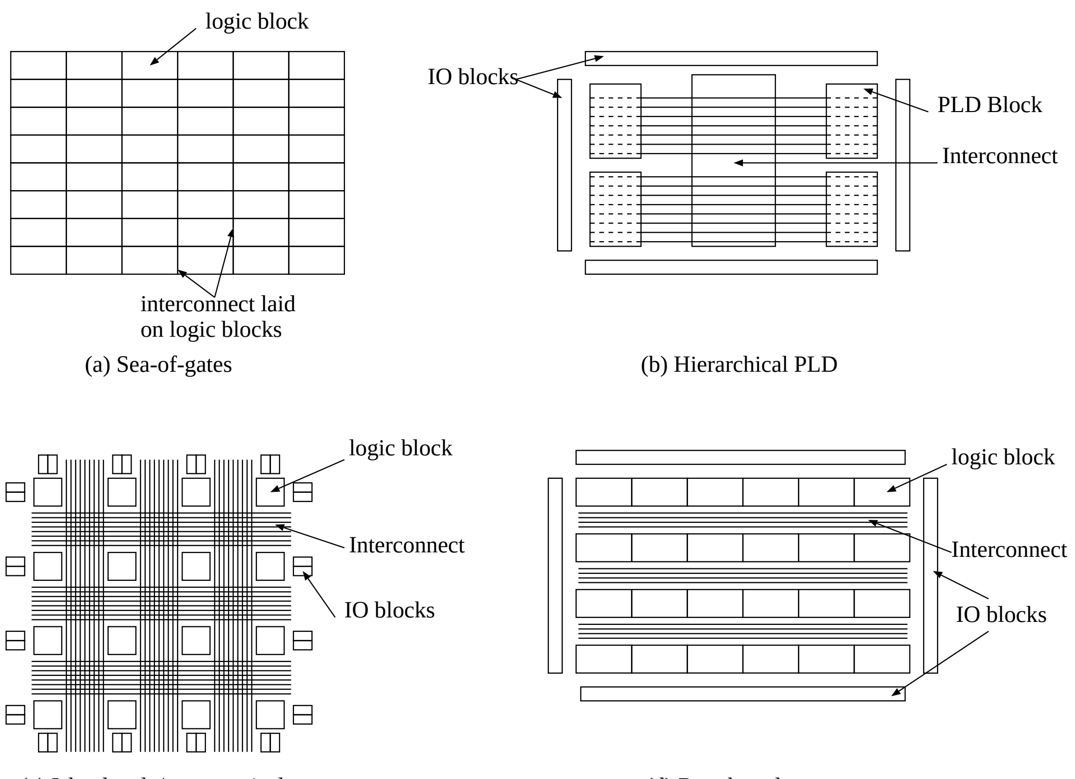


Figure 1.1: Basic FPGA architecture.

to which each incoming wire can connect in a switch block. The *connection block flexibility* is the number of wires in each channel to which a logic block input or output pin can connect. The *segmented wire length* is the number of logic blocks a wire segment spans. As the wires cross different number of logic blocks in segmented wire architecture, the FPGA interconnect delays become highly non-linear, discrete, and in some cases, even non-monotone with respect to the distance. If there is a connection between two CLBs whose distance is 3 CLB units, and there are two options of connecting them; (i) using two wires spanning 1 CLB each, or (ii) one wire spanning 3 CLBs, the second option incurs less delay and hence leads to better performance than the first one. Finally, an H-tree based clock network [Sherwani 1993] is laid on to the chip for feeding the *flip flops* in LUTs.

Each *logic block* or CLB usually contains a group of *basic logic elements* (BLE). In an LUT based FPGA architecture, typically each BLE contains a K-input *lookup table* and a register. The output of K-LUT [Brown 1995, Kuon 2007] can be registered by connecting it to a flip flop, or it can remain unregistered. Commercial FPGAs, for example, Altera's Stratix II FPGA use an *adaptive logic module* (ALM) which contains a group of LUTs and a pair of flip flops [Altera]. The ALM can expand and share the LUT inputs and has more logic capacity than traditional 4-input LUT structures for an equivalent function. An ALM is similar to a 7-input LUT



(a) Sea-of-gates (b) Hierarchical PLD (c) Island style/ symmetrical array (d) Row based

Figure 1.2: Types of FPGA architecture.

that can be flexibly re-partitioned into a number of configurations such as either a 5-input LUT and a 3-input LUT, or two 4-input LUTs. With input sharing, larger combinations can be created, such as two 5-input LUTs or even two 6-input LUTs. The larger logic capacity in Stratix II ALMs improves the overall performance of the design by packing more logic in an ALM.

1.3 Types of FPGA

FPGAs are used in different types of applications and the resource requirements are also diverse. FPGAs were first developed by Xilinx in mid $80's$ [Xilinx]. Later on, many companies such as Altera [Altera], Lattice Semiconductor [Lattice], Actel [Actel], SiliconBlue Technologies [SiliconBlue], Achronix [Achronix], Quicklogic [QuickLogic] developed their own FPGAs of different structures. FPGAs can either be classified by the structure and organization of the logic and routing resources, called *structural classification*, or by the programming technologies used to configure it, called *programming technology based classification.*

1.3.1 Structural classification

FPGAs can be categorized into four classes depending on the arrangement of its logic and routing resources as shown in Figure 1.2. These are:

1.3.1.1 Sea-of-gates architecture

Logic blocks are arranged in a two dimensional array, and interconnects are overlaid on top of the logic blocks as shown in Figure 1.2(a). The *logic block* is generally composed of a multiplexer which feeds a NAND gate or some other functional unit, and a *latch*. Logic blocks are connected to routing resources through multiplexers, and a Static RAM (SRAM) cell controls the multiplexer.

1.3.1.2 Hierarchical PLD architecture

Figure 1.2(b) shows the hierarchical *programmable logic device* (PLD) architecture. The logic blocks or PLD blocks and programmable interconnects are arranged in a hierarchy. Each PLD block contains logic modules with programmable combinational and sequential elements. The logic modules are programmed using the bits stored in configuration memory. Altera FPGAs [Altera] belong to this category of FPGAs.

1.3.1.3 Symmetrical or island-style architecture

This type of FPGAs is shown in Figure 1.2(c). The CLBs are arranged uniformly in rows and columns as explained in Section 1.2, with wires laid out in the horizontal and vertical channels between rows and columns of CLBs respectively. The IOBs are located on the periphery of the two dimensional array. Each CLB contains one or more K-input LUTs and flip flops. The routing is done through the wires laid out in the channel and the switching blocks present at every junction of horizontal and vertical channel by appropriate programming of the switch blocks. Xilinx, Achronix, SiliconBlue and QuickLogic FPGAs provide this type of FPGA architecture.

1.3.1.4 Row based architecture

Row based architecture has the logic modules and programmable interconnects arranged in alternate rows, as shown in Figure 1.2(d) in one layer. The vertical interconnects, laid out on top of it in a separate layer are used to connect modules in different rows. The IOBs are located on the periphery surrounding the logic modules. Actel Corporation provide this type of FPGAs.

Company	General Architecture	Type of logic block	Programming Technology
Xilinx	Island-style	LUT-based	static RAM
Altera	Hierarchical PLD	PLD block	EEPROM
Actel	Row based	MUX based	Anti-fuse
Quicklogic	Island-style	MUX based	Anti-fuse

Table 1.1: Summary of FPGA architectures provided by different companies

In summary, most of the commercial FPGAs belong to the category of island-style FPGAs and are used widely in the FPGA research community. Thus, island-style FPGA architecture is assumed for the proposed methods in this thesis. However, the CAD tools developed for one type of FPGA architecture can be adapted to other architectures by adding constraints relating to the location of the logic blocks pertaining to that architecture.

1.3.2 Programming technology based classification

FPGAs can be classified into three groups depending on the programming technologies used to configure the FPGAs [Brown 1995, Kuon 2007].

Anti-fuse programming technology: FPGAs using this technology are one time programmable as *anti-fuses* make permanent connections between logic blocks. Programming is done by blowing the anti-fuse. A high voltage breaks down the metal-to-metal anti-fuse and causes the fuse to conduct [Kuon 2007]. It provides a low resistance, bidirectional connection between segments. This technology is used in Actel and QuickLogic FPGAs.

Static RAM (SRAM) programming technology: Programmable connections are realized using pass gates or multiplexers. SRAM cell controls the pass gates and multiplexers. Since SRAM is volatile, the FPGA needs to be reconfigured each time the power is applied to the chip. Xilinx, Lattice Semiconductor, Altera FPGAs mostly use SRAM programming technology.

EEPROM programming technology: It uses EEPROM (*electrically erasable programmable read-only memory*) memories and it helps reprogramming of device by erasing and rewriting into the configuration memory several times. Unlike SRAM technology, it does not require external memory storage to program it at power up. Altera FPGAs also use this technology for programming.

Of these approaches, *static memory* and *anti-fuse* technologies are widely used in modern FPGAs. Although SRAM based cells occupy large areas, it is widely

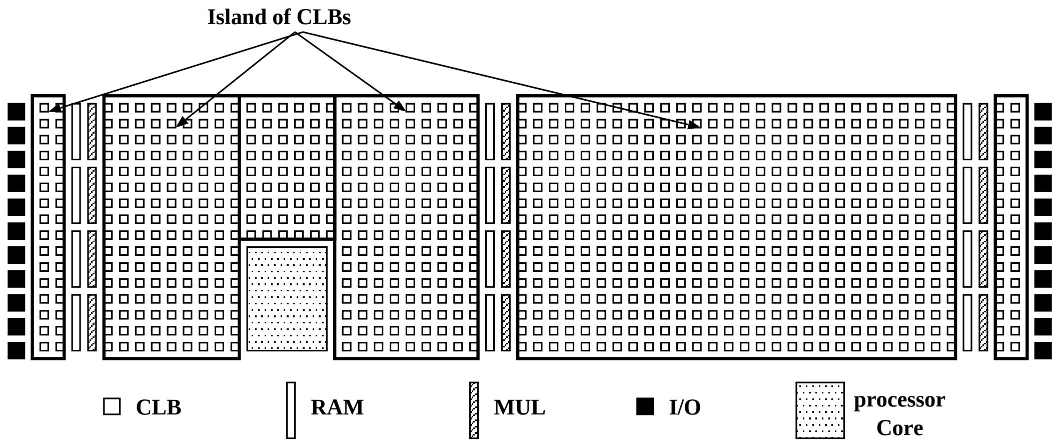


Figure 1.3: Modern FPGA architecture with Block RAM and Multipliers.

used for its re-programmability features. Moreover, an FPGA can even be partially reconfigured, while a portion of it is still executing some operations using the SRAM technology. Most of the symmetrical array based FPGA architectures use SRAM programming technology, and row based structures use anti-fuse technology. Table 1.1 shows a summary of the existing major architectures adopted by the leading FPGA fabrication companies [Xilinx , Altera , Actel , QuickLogic].

1.3.3 Modern FPGAs

The basic FPGA consisted of only the CLBs and IOBs in the beginning. These are termed as FPGAs with *homogeneous* resources. Unlike earlier generation FPGAs, modern FPGAs such as Xilinx's Spartan and Virtex Series [Xilinx], Altera's Stratix series [Altera] consist of not only the CLBs, but also block RAMs (BRAM), Multipliers (MUL), DSP cores, and even small microprocessors on the FPGA chip to support *System-on-Chip* (SoC) designs. One such configuration with CLBs, RAMs, multipliers (MULs) and processor core is shown in Figure 1.3.

The first type of *heterogeneous* resources in FPGAs was BRAM which first appeared commercially in the Altera Flex 10K series FPGA [Ngai 1995, Altera 2003, Kuon 2007]. This memory block consisted of 2Kb of static RAM, which could be configured as either a 2048 $\times$ 1, or 1024 $\times$ 2, or 512 $\times$ 4 or 256 $\times$ 8 bit memory. FPGAs in the Xilinx Virtex-4 series onwards have block RAMs of 18Kb [Xilinx]. Xilinx Virtex II FPGA [Xilinx 2005] first introduced the multiplier (MUL) integrated into the chip. This series of FPGAs contained 18 $\times$ 18 multiplier that sat along a block memory. Since the introduction of Virtex II, Xilinx and other manufacturers have introduced more sophisticated hard computational units that include multiplier-accumulators, and some multiplexer functions [Kuon 2007]. The Stratix I [Kuon 2007] of Altera contains a single 36×36 multiplier-accumulator block that can be broken into eight 9×9 multipliers and an adder to sum results. Before long, Altera introduced *Excalibur*, an FPGA which included a hard processor core from ARM Inc. [ARM], connected to an Altera Apex 20K series FPGA [Kuon 2007]. Almost at the same time, Xilinx introduced Virtex II Pro FPGAs which included one, two, or four IBM Power PC microprocessor cores [IBM] integrated with a Virtex II logic fabric [Xilinx 2005]. Several Xilinx Virtex-4 and Virtex-5 subfamilies also support Power PC cores [Kuon 2007].

The most popular Xilinx Spartan series has about $478K$ gates, $35K$ FFs, which can operate at 300MHz. These FPGAs are built on $90nm$ technology, while Virtex series FPGAs use $65nm$ technology having up to 2.2 million gates, $153K$ FFs, operating at 550MHz [Xilinx]. Altera's Stratix series has up to 7.7 million gates, $430K$ FFs in its logic cells, and uses *adaptive logic module* (ALM) technology [Altera]. This chip can operate at 600MHz and it is made in $65nm$ technology. Lattice Semiconductor uses SRAM technology having FPGAs with 1.7 million gates operating at 550MHz. Actel produces FPGAs with $752K$ gates operating at 350MHz. These FPGAs use anti-fuse technology for programming and are built on $130nm$ technology. SiliconBlue Technologies and Achronix FPGAs are used for smaller applications. SiliconBlue Technologies provides low power FPGAs with $17K$ gates, operating at 32MHz fabricated in 65nm technology. Achronix FPGAs are RAM based and power efficient for being asynchronous.

Almost all of the modern FPGA architectures have CLBs, Block RAMs (BRAM) and Multipliers (MUL) embedded in the FPGA chip. Thus, the Xilinx Spartan like architecture is assumed for the efficient floorplanning methods presented in this thesis. However, the proposed methods can be suitably adapted for other architectures by adding appropriate location constraints for each additional resource on a modern FPGA chip.

1.4 Design flow for FPGA

In order to obtain high performance circuit on an FPGA, highly efficient CAD tools are required in each step of the FPGA design flow. Figure 1.4 depicts the typical CAD flow for modern FPGAs. A circuit to be mapped on to an FPGA, is first written in any of the hardware description languages such as VHDL/ Verilog. This description is then *synthesized* and *technology mapped* to the target FPGA architecture depending on the number of inputs and outputs available in the LUT of

CLBs. Next, this *technology-mapped netlist of CLBs* are *placed* to physical locations on the target chip optimizing certain objective functions such as minimization of wirelength, delay and power. In recent times, due to large design size, increased complexity and partial reconfigurability in FPGAs, circuits are partitioned into modules and the *technology-mapped* netlist of modules are *floorplanned* before placement of individual CLBs in a module. The placement is then *routed* by determining the exact pre-fabricated wire segments to be used for inter-connecting the terminals of the placed CLBs. This defines the programming of the appropriate programmable interconnects on chip, and is stored as a *bitstream*. Finally, the bitstream of the placed and routed circuit is downloaded on to the FPGA chip through a hardware interface such as a dedicated serial or parallel port of a PC, for execution. A more detailed discussion on each of this step follows [Chen 2006]:

Design entry: Typically, the digital design is represented at *register transfer level* (RTL) describing the transfer of signals between registers and the logical operations performed on these signals during transfers. This RTL design is then described using various hardware description languages such as VHDL [IEEE 1987] / Verilog [Verilog International 1993]. The design constraints such as expected operating frequency, delay bounds of signal path delays (i) from input pads to output pads (I/O delay), (ii) from input pads to registers, and (iii) from registers to output pads are specified as input. The design constraints also include physical location constraints specifying the position of certain type of logic in a particular place. The target FPGA device to which the given design is to be mapped, is next selected from the set of available architectures depending on the size and resource requirements of the design.

Synthesis: For a design, different *datapath* operations such as additions, multiplications, register files, memory blocks, control logic etc, are identified from the given Register-Transfer-Level (RTL) design. Next, architecture independent optimizations such as *datapath optimization* and *control logic optimization* is performed.

Technology mapping: After synthesis and architecture independent optimization, the technology mapping step maps the optimized datapath and control logic to dedicated circuit structures such as multipliers, embedded memory blocks and CLBs available on the FPGA chip. This generates a *netlist* of CLBs, IOBs and other resources to be assigned to the target FPGA chip. The *netlist of CLBs* defines a set of *signal nets*, where each *signal net* connects a set of CLBs or IOBs.

Floorplan: With increasing complexity of modern FPGAs and design size, physical mapping of a flattened *technology mapped netlist* of CLBs and IOBs on to a chip has become more complex, and is likely to be counter-productive for obtaining

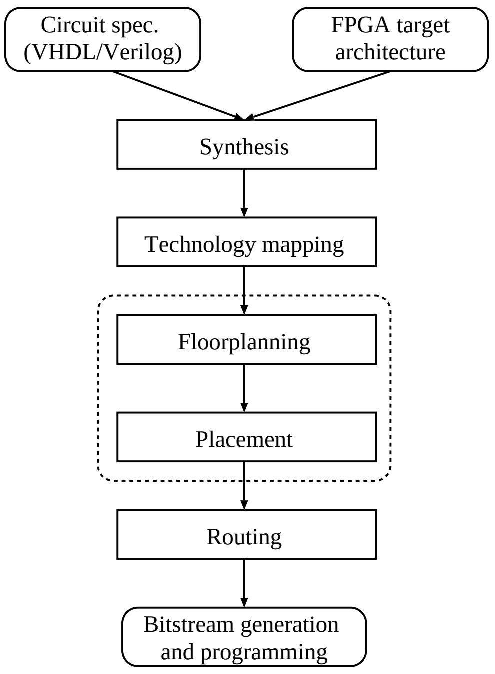


Figure 1.4: CAD flow for FPGAs; the rectangle with dashed lines shows the scope of this thesis.

high performance. In order to overcome these issues, a given design is partitioned into modules and then mapped on to a target architecture. Each module in the technology mapped *netlist of modules* requires a number of different logic resources available on the chip. A floorplanning step has become essential for modern FPGAs to allocate required logic resources to each of the module. The floorplanning problem for FPGAs with pre-placed heterogeneous resources and fixed-die size is the problem of allocating logic resources to each module, such that there is no overlap of modules while the optimization criteria such as total wirelength and area of the floorplan are minimized. One of the main focus of this thesis is this floorplanning step of physical design cycle for heterogeneous FPGAs.

Placement: After the technology mapping step, a netlist of *logic blocks* is obtained. The CLBs are then assigned to specific locations on the target chip such that certain objective criteria such as minimization of wirelength and *critical path delay*, are achieved. In order to obtain high performance circuit, the delay due to interconnection has to be minimized. This requires the connected CLBs and IOBs to sit as close as possible to each other. The exact delay of a connection can not be computed until and unless the path of the connection is routed through the channels and switch boxes on a particular FPGA chip. For large designs, this step is preceded by floorplanning of the modules, and then the netlist of CLBs within each module are placed.

The quality of a placement solution is typically judged by a *cost metric* called *half-perimeter wirelength* (HPWL) cost [Sarrafzadeh 1996] of a placement. For a *signal net* (i.e., a set of terminals of certain CLBs which are to be inter-connected) the minimum rectangle enclosing all the CLBs to be connected is called the *bounding box* of the net. The HPWL of a placement is the sum of the half-perimeters of the *bounding boxes* of all the signal nets in the given circuit, and hence called the *bounding box cost* (*BB cost*). The *critical path delay* of a circuit is the delay due to the longest path from an input pad to an output pad. Thus, it decides the performance of a circuit and needs to be minimized to obtain a high performance circuit. Since the exact delay can not be computed until and unless the signal nets of the design are routed, HPWL is used to evaluate the quality of the placement solution. Thus, HPWL or *BB cost* metric gives an estimate of the total routing wires required for a placement and this needs to be minimized for a high quality placement. HPWL is the standard and most popular estimation metric for evaluating the placement solution [Sherwani 1993, Sarrafzadeh 1996].

In this thesis, several variants of fast and efficient methods of placing the netlist of CLBs and IOBs have been proposed with the objective of minimizing the HPWL cost.

Routing: In this step, global and detailed routing are performed to connect all signals or nets in the netlist using the available programmable interconnects on the chip. If the solution is not routable, then the placement of logic blocks has to be altered and the place-then-route steps have to be iterated to obtain a high performance routable design.

Bitstream generation and programming: This step takes the placed and routed design to generate the necessary bitstream to program the logic and interconnects implementing the intended logic design on the target device through a serial or parallel port interface.

1.5 Scope of the thesis

This thesis focuses on the *placement* and *floorplanning* steps of the FPGA physical design flow. The rectangle with dashed lines in Figure 1.4 indicates the scope of this thesis. Most of the CAD algorithms usually take long time to map, place and route circuits with millions of gates on a state-of-the-art FPGA chip. This may nullify its advantage of very short time-to-market, and in particular the capability of reconfiguration by the users. With increasing emphasis on reconfigurable computing, there is a pressing need for very fast CAD tools. Both the placement and floorplanning problems for FPGAs are NP-hard [Shahookar 1991, Sherwani 1993]. Thus, optimal solution can not be obtained in reasonable time. Efficient heuristics are needed that quickly produce solutions with good quality. Current FPGA architectures such as Xilinx Virtex series FPGA, consist of CLBs along with BRAMs and MULs.

The placement and floorplanning methods described in this thesis are developed for such heterogeneous FPGAs, although the basic methodology may be applied to other architectures as well with some modifications. All the methods in this thesis aim at achieving quick solution in the context of FPGA's reconfigurability with minimal compromise in quality of the solution. The quality of the placement and floorplanning is measured using the standard metric of half-perimeter wirelength (HPWL) over all the *signal nets* in a placed circuit. Most of the modern commercial FPGAs being of island-style, placement problems for this architecture has been addressed in this thesis. Further, as most of the modern FPGAs have BRAMs and MULs embedded in the chip, Xilinx Spartan like architecture is assumed for the floorplanning methods discussed in this thesis. However, the proposed methods can be adapted to the similar FPGA architectures appropriately.

1.5.1 Placement of CLBs on island-style homogeneous FPGAs

As stated earlier, placement is the process of assigning position to each CLB and IOB in the netlist to a particular location on a target FPGA chip such that certain objective functions are optimized. The optimization criteria can be minimization of total HPWL of all nets, or *critical path delay*, or *congestion*, or *power dissipation*, or *crosstalk*, or a combination of the above. Figure 1.5 shows the placement problem for FPGAs discussed in this thesis. The precise problem formulation is given in Chapter 3.

In order to solve the placement problem, one group of researchers have aimed at obtaining accurate estimation metrics to capture the exact details of the chip architecture and the netlist characteristics. They have applied stochastic iterative methods such as *simulated annealing* (SA) to obtain high quality solution using cost metric for capturing the architectural and placement details during every iteration. The placement configuration is perturbed by swapping CLBs or IOBs and the schedule of annealing is updated depending on the new value of the cost. This process continues until there is no significant change in the cost for a few consecutive iterations. The first and the most popular tool developed using this concept is the VPR (*Versatile Place and Route*) [Betz 1999, Marquardt 2000]. VPR produced

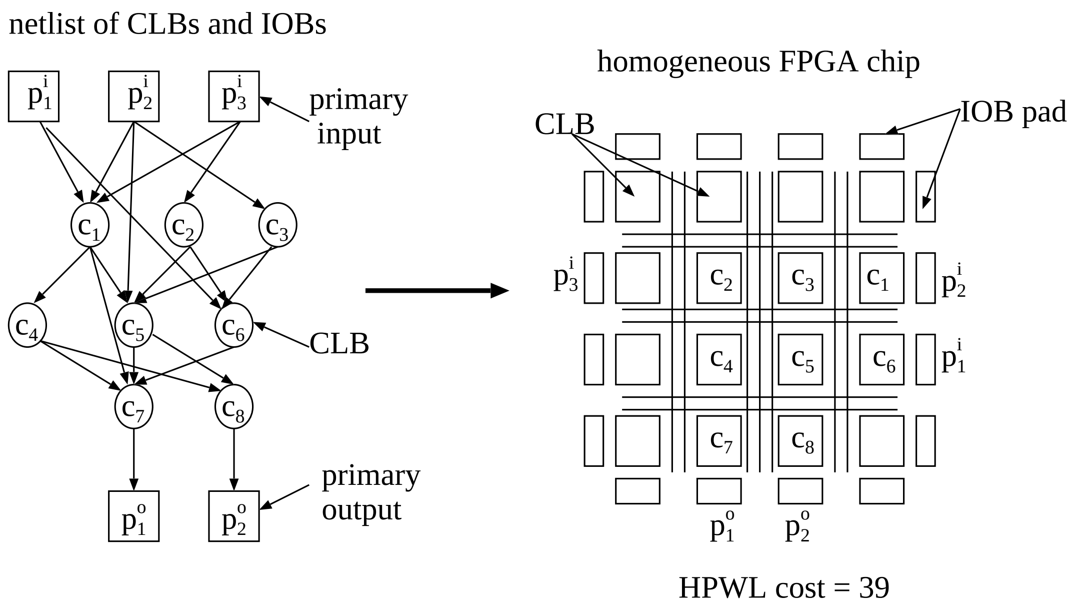


Figure 1.5: The placement problem for FPGAs with CLBs and IOBs.

high quality of solution at the cost of fairly long execution time. But in the context of reconfigurability, the placement method has to be very fast. Moreover, real time applications and smaller applications may not even require high quality solution at the cost of longer place-and-route time. Thus, another group of researchers aimed at obtaining faster solution with minimal compromise in the quality using *tabu-search based* technique [Emmert 1998], *partition-based approaches* [Maidee 2005], *analytical approaches* [Xu 2005, Gopalakrishnan 2006]. It was observed that, all of the above methods produced initial placement solutions, which were further improved by a low temperature simulated annealing, thus reducing the execution time of simulated annealing based placer.

The quality of the simulated annealing based placement depends largely on the initial placement configuration [Vorwerk 2009]. Many trials have to be performed with various initial solutions. This motivated the development of fast placement methods that produce quick solutions with minimal sacrifice in the quality. Although different types of heuristics such as simulated annealing based method, partitioning based method, analytical methods, meta-heuristic methods such as tabu search and many other heuristic methods have been tried to produce high quality solution quickly, none of them report the quality of the initial solution produced by these methods. Moreover, there is no theoretical bound on the quality of the solution thus obtained. This salient issue for FPGA placement methods is addressed in this thesis. The major contributions in FPGA placement are:

- New deterministic yet fast methods using both bottom-up and top-down paradigms for placing a flattened CLB netlist on to an island-style FPGA. Observation of the quality of placement before application of low-temperature simulated annealing (Chapters 3 and 4) and,

- Deriving a theoretical bound on the quality of the placement produced by the top-down approach (Chapter 5).

First, a bottom-up *cone based* greedy heuristic for placement of both CLBs and IOBs is proposed, and the quality of the solution before and after the execution of a low-temperature simulated annealing has been observed. Next, experimental results on new top-down deterministic approaches for placement are presented, and again the quality of the solution produced both before and after the execution of a low temperature simulated annealing are scrutinized. Finally, a theoretical bound on the quality of the solution is derived for a special case of the top-down deterministic method. Given a set of choices for placement tools with bounds on the quality of the solution, a user can choose a specific placement tool for a particular design requirement. For example, to place a small design without very stringent performance requirement, one can opt for a placement tool which produces solutions 33× faster with 1.31× degradation in the quality of the best known solution.

Current FPGAs having multiple pre-placed resources, segregate the CLBs into more than one smaller islands of CLBs as shown in Figure 1.3. Once the floorplan of the modules are obtained, the CLBs pertaining to a module needs to be placed in a small array of CLBs in a given region. This generates a set of independent placement subproblems which place smaller designs on to a set of smaller 2D arrays of CLBs. Application of highly sophisticated but complex placement methods to solve such subproblems may add overhead on the overall run-time of the FPGA physical design cycle. In order to achieve faster place-and-route time in such a scenario, our fast, deterministic placement methods can play an important role.

1.5.2 Floorplanning for heterogeneous FPGAs

A large design with millions of gates is typically partitioned into a smaller number of functional modules to reduce the compile time for place-and-route and to obtain better quality of solutions. Moreover, FPGAs with pre-placed resources on it has necessitated a floorplanning step for hierarchical designs in the physical design flow of FPGAs. Although a large volume of work exists for ASIC floorplanning, these were generally not employed while mapping designs on to the earlier island-style

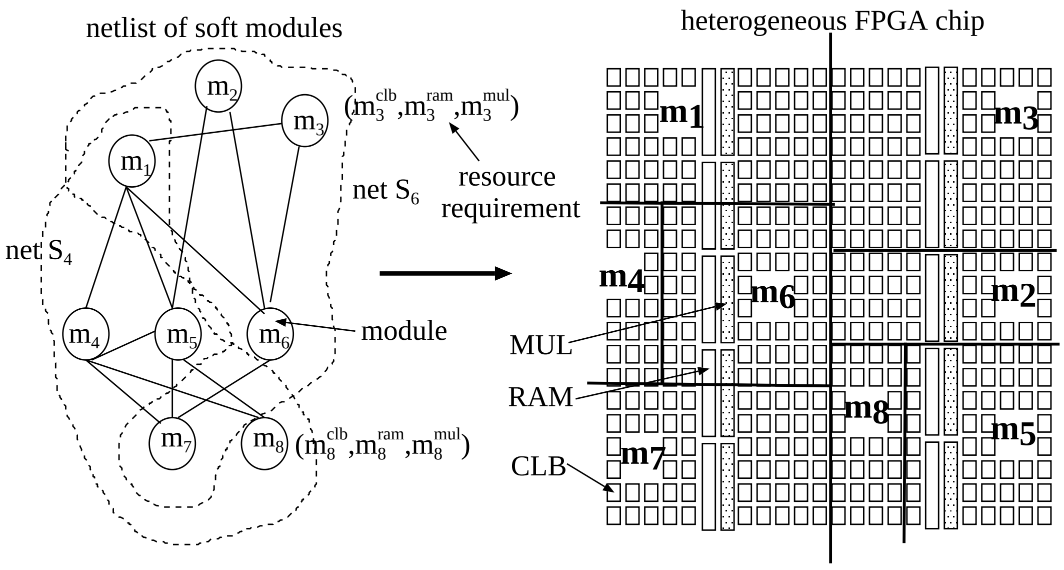
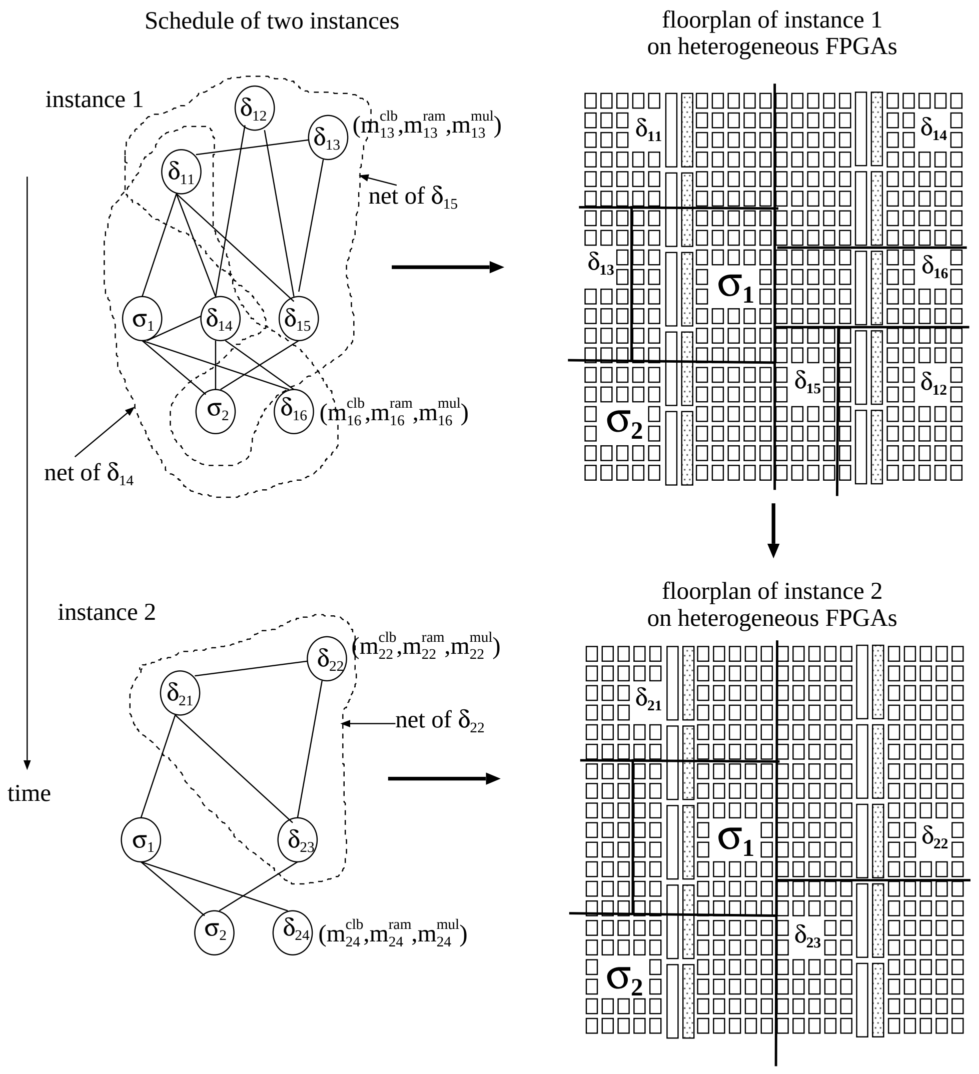


Figure 1.6: The floorplanning problem for FPGAs with *heterogeneous* resources; net S_4 and S_6 are the *nets* consisting of modules $\{m_1,\ m_4,\ m_5,\ m_7\ m_8\}$ and modules $\{m_1,\ m_2,\ m_3,\ m_6\ m_7\}$ respectively.

FPGAs. In a typical FPGA physical design flow, after technology-mapping, a flattened CLB netlist is directly placed [Betz 1997, Maidee 2005] and routed without any floorplanning. Of course, for hierarchical designs, modules or macros consisting of CLBs only were floorplanned or placed using various *bin packing* techniques [Tessier 2002, Emmert 1998]. But, for modules with heterogeneous resource requirements, neither this technique nor the traditional floorplanners for ASICs adapted to FPGAs, are adequate [Wang 2003, Taghavi 2004]. Hence, there is a pressing need for fast floorplanning techniques that consider the heterogeneous logic and routing resources of modern FPGAs. The floorplanning problem for a netlist of *soft modules* (modules without fixed dimensions) is the problem of determining the size and position of each module on a chip such that the modules are non overlapping and certain optimization criteria such as wirelength, area of the floorplan are minimized. Figure 1.6 shows the floorplanning problem for FPGAs with heterogeneous resources. The precise problem formulation is given in Chapter 6.

The floorplanning problem is also NP-hard [Sherwani 1993] and thus solved with heuristic methods. Most of the earlier approaches of floorplanning for FPGAs [Cheng 2004, Cheng 2006, Feng 2006] with heterogeneous resources are based on simulated annealing, and are slow in the context of reconfiguration. Our focus is to develop fast floorplanning method which is deterministic, yet produces floorplans of high quality for FPGAs with heterogeneous resources. Once again, the standard



Schedule of two instances
floorplan of instance 1 on heterogeneous FPGAs
instance 1
$(m_{13}^{clb}, m_{13}^{ram}, m_{13}^{mul})$
net of δ_{15}
$(m_{16}^{clb}, m_{16}^{ram}, m_{16}^{mul})$
net of δ_{14}
floorplan of instance 2 on heterogeneous FPGAs
instance 2
$(m_{22}^{clb}, m_{22}^{ram}, m_{22}^{mul})$
net of δ_{22}
time
$(m_{24}^{clb}, m_{24}^{ram}, m_{24}^{mul})$

Figure 1.7: The floorplanning problem for partial reconfiguration in FPGAs with *heterogeneous* resources; σ: common (*static*) modules, δ: *dynamic* modules.

HPWL cost metric is chosen for evaluating the quality of the floorplans obtained by the proposed method.

1.5.3 Floorplanning for partial reconfiguration in heterogeneous FPGAs

FPGAs are programmable since its inception. However, in the past, the complete chip had to be configured whenever it required reconfiguration. This incurred reconfiguration overhead. Also, chunks of FPGA resources remained unutilized, when small tasks were executed on FPGA. To better utilize the FPGA resources, modern day FPGAs provide partial reconfiguration capability on FPGAs, i.e., a part of the chip can be reconfigured while the other part is in operation. In order to effectively use this feature, new CAD tools are required to map the given set of tasks to the FPGA chip subject to a set of conflicting optimization parameters. The objective is to place and route the tasks very quickly on to the FPGA chip such that the resource utilization is maximized, reconfiguration overhead is minimized while the performances of the tasks are maximized. Given a *schedule of instances*, each *instance* having a netlist of modules and a set of common modules across all instances, the floorplanning problem is to determine *globally* the sizes and positions of the common modules, invariant over all the instances in the schedule, so that the partial reconfiguration overhead is minimized, while the total wirelength of all the floorplans for the entire schedule is minimized. With heterogeneous FPGA chips having pre-placed resources, the global floorplanning problem becomes a very difficult task and is NP-hard [Garey 1979]. Figure 1.7 shows the floorplanning problem in the context of partial reconfiguration in heterogeneous FPGAs. Chapter 7 has the precise problem formulation of floorplanning for partial reconfiguration.

The research in this new area of partial reconfiguration is merely a handful. Moreover, the existing methods are based on simulated annealing based approaches [Singhal 2006] as in the case of floorplanning. Our objective is to develop a fast yet effective *global* floorplanning method for the set of instances in a given schedule such that the partial reconfiguration overhead is minimized without compromising the quality (measured as HPWL) of the floorplan.

1.6 Organization

Figure 1.8 gives a pictorial representation of the organization of this thesis. The earlier works on FPGA placement, floorplanning and partial reconfigurations are detailed in Chapter 2. The two variants of our fast deterministic placement methods for netlist of CLBs on an island-style FPGA, namely (i) bottom-up greedy *cone* based approach and (ii) top-down partition-based approach, have been described in chapters 3 and 4. In Chapter 5, a theoretical bound on the quality of placement obtained by the most promising method, described in Chapter 4, is derived. Chapter 6 explains our deterministic unified FPGA floorplan topology generation and sizing method for floorplanning a netlist of modules with heterogeneous resources. A global floorplanning method in the context of partial reconfiguration is proposed in Chapter 7. Finally, the concluding remarks and potential future research directions appear in Chapter 8.

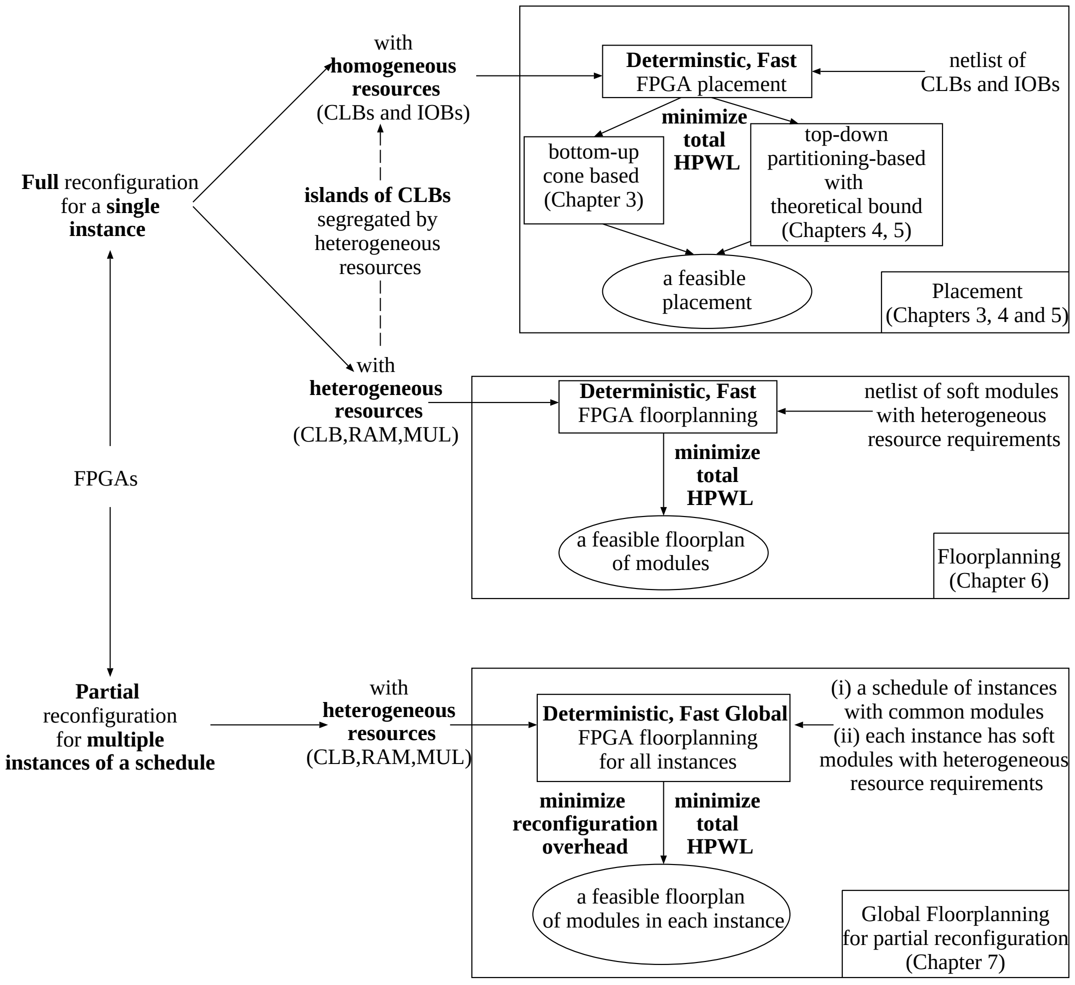


Figure 1.8: The organization of this thesis.

Chapter 2

Previous Works

Contents

In this chapter, a survey of different approaches for placement and floorplanning is reviewed with respect to the placement tools for earlier generation small scale FPGAs with CLBs to the recent partially reconfigurable FPGAs of the present day. Section 2.1 has the review of different placement approaches for FPGA placement. Section 2.2 provides the earlier works on FPGA floorplanning for both homogeneous and heterogeneous FPGAs. The techniques of floorplanning for partial reconfiguration on FPGAs with heterogeneous resources have been described in Section 2.3. The chapter closes with concluding remarks in Section 2.4

2.1 Placement on island-style FPGAs

FPGA placement is an NP-hard combinatorial optimization problem [Shahookar 1991]. So, there is no known algorithm that produces optimal solution in reasonable CPU (guaranteed polynomial) time. Thus, many heuristic techniques have been developed to obtain solutions. The heuristics are broadly divided into the following three categories:

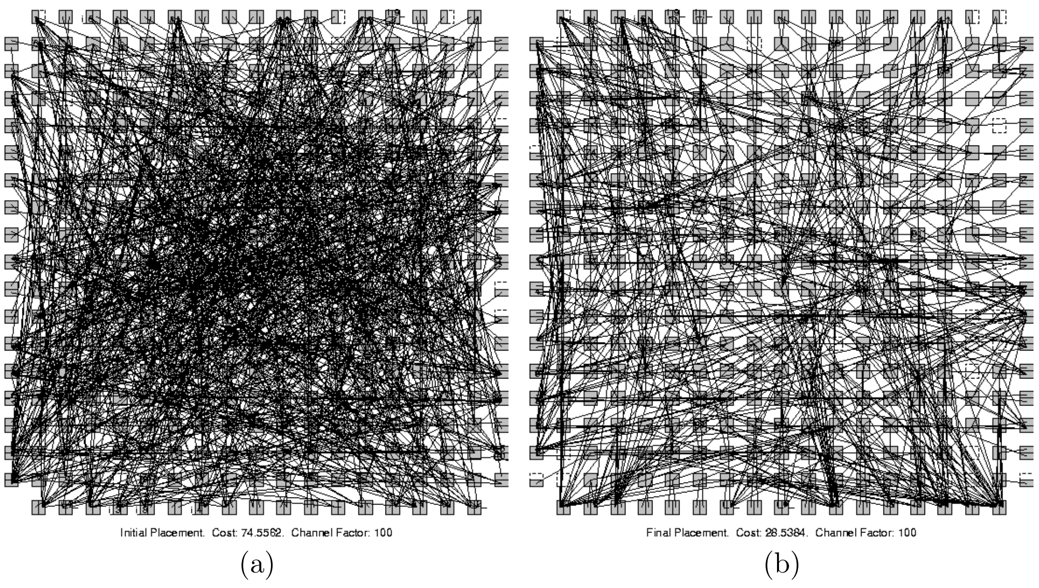

Initial Placement. Cost: 74.5562. Channel Factor: 100
(a)
Final Placement. Cost: 28.5384. Channel Factor: 100
(b)

Figure 2.1: An example of placement [Betz 1999]: (a) random initial placement of netlist of CLBs and IOBs, (b) final placement minimizing *linear congestion cost* (note the reduction in congestion); (Courtesy: [Betz 1999]).

- stochastic methods
- partitioning based placement
- analytical placement

2.1.1 Stochastic methods

In stochastic methods such as simulated annealing based placement, an initial placement configuration of CLBs and IOBs is obtained by a random assignment of blocks to legal positions on the FPGA target array. Each placement configuration usually has a multi-objective cost function such as wirelength, timing, power, to be optimized through several iterations. The initial placement is perturbed by swapping the positions of two blocks. The move is either accepted or rejected depending on an *acceptance criteria* based on the *annealing schedule* defined for simulated annealing. The iteration halts when the given *exit criteria* is met, or there is no more change in the cost function due to moves. Thus, simulated annealing is a stochastic heuristic that randomizes the iterative improvement procedure, even allowing moves that worsen the current solution in order to prevent the search from getting stuck at a locally optimal solution. The moves are controlled probabilistically by an annealing temperature. Theoretical analysis shows that this class of algorithms converges to a global optimum asymptotically with a probability 1, provided certain conditions are met [Wong 1988, Sechen 1988]. In reality, it is almost equivalent to searching the entire feasible solution space [Vygen 2007]. So, the research in VLSI placement using this method, has boiled down to empirically finding suitably well tuned parameters in the objective function and an appropriate annealing schedule by performing several trials, such that near-optimal solutions are obtained.

Among these methods, the place and route tool called VPR (*Versatile Place and Route*) by Betz and Rose [Betz 1997, Betz 1999, Marquardt 2000] was the first FPGA place-and-route tool and it has become the most popular one. The results obtained by VPR has become the benchmark for the entire research community working in FPGA placement and routing. A placement result obtained by VPR [Betz 1999] on a target FPGA chip is shown in Figure 2.1(b). VPR employs a *congestion aware half-perimeter bounding box metric* in *wirelength driven* mode, with an *adaptive simulated annealing schedule* for island-style FPGAs. The *linear congestion cost* function takes into account the *channel width* along with the half-perimeter wirelength of all nets. In the *timing driven* mode, the timing optimization is done by the addition of a *timing cost* term to the objective function. It minimizes the *weighted delays* of all connections, where the weight of a connection depends on its *slack*. This does not consider the impact of exponential number of paths going through a particular connection. The normalized difference in costs of two placement configuration has two components: the *linear congestion cost* for the *wirelength driven* mode, and the *timing cost* for the *timing driven* mode as follows [Marquardt 2000]:

$$\triangle C = \lambda \cdot \frac{\triangle timing\_cost}{previous\_timing\_cost} + (1-\lambda) \cdot \frac{\triangle linear\_congestion\_cost}{previous\_linear\_congestion\_cost}$$

where $linear\_congestion\_cost$ is

$$\sum_{i=1}^{N_{nets}} q(i) \left[\frac{bb_x(i)}{C_{av,x}(i)^{\beta}} + \frac{bb_y(i)}{C_{av,y}(i)^{\beta}} \right]$$

and $timing\_cost$ is

$$\sum_{\forall i,j \in circuit} Delay(i,j) \cdot Criticality(i,j)^{crit\_exp}$$

$\triangle$ in the above equation is the change in cost function. For each net i, $bb_x(i)$ and $bb_y(i)$ denote the horizontal and vertical spans respectively of the *bounding box* of i^{th} net. The factor $q(i)$ compensates for the fact that the *bounding box* wire length model underestimates the wiring necessary to connect *nets* with more than three terminals. $C_{av,x}$ and $C_{av,y}$ are the average channel capacities in the x and y directions respectively, over the *bounding box* of *net i*. The exponent β in the cost function allows the relative cost of using narrow and wide channels to be adjusted. $C_{av,x}$ and $C_{av,y}$ are constants in case of *bounding box cost* function. β is set to 0 to revert the *linear congestion cost* function to the *bounding box cost* function (pp. 56 of [Betz 1999]). $Delay(i, j)$ is the delay defined in the *delay lookup matrix* corresponding to the placement of two blocks i and j. $Criticality(i, j)$ defines the criticality of a *source-sink* connection between i and j [Marquardt 2000]. The *crit_exp* biases the critical connections more than the non critical ones [Marquardt 2000]. The performance of VPR has been validated on a set of MCNC (Microelectronics Center of North Carolina) benchmark circuits, where the size of the largest benchmark is about 9000 CLBs.

Kong [Kong 2002] has proposed an algorithm *PATH*, which can scale the impact of all paths by the relative timing criticality measured by their slacks. The work shows that in certain cases, enumeration of all paths in the circuit, counting their weights and then distributing the weights to all edges can be done in linear time. Compared to VPR, *PATH* reduces the longest path delay by 15.6% on average with no run-time overhead and only 4.1% increase in total wirelength.

Although simulated annealing based methods, such as VPR, produce "good" quality of solution with suitably tuned parameters in terms of congestion-aware HPWL and critical path delay during routing, it takes long hours [Mulpuri 2001, Chen 2006] to execute. As it is an iterative move-based stochastic method, the execution time increases as the size of the netlist increases. This drawback affects the run-time reconfigurability advantages of FPGAs, and hence the scalability.

Emmert et al. [Emmert 1999b] have proposed meta-heuristic search techniques such as *tabu search* optimization for the placement step in order to reduce the execution time while providing high quality solutions. The proposed method employing *tabu search* minimizes the total wirelength and the length of critical path edges for placed circuits on FPGAs. The moves per iteration are similar to *force-directed* placement methods for gate arrays [Eisenmann 1998]. The results showed dramatic improvement in placement time relative to the commercially available CAD tools (20×) for a set of small benchmark circuits, and the quality of placement was comparable to that of the commercial tools. The benchmark circuits were smaller in size, having less than 1000 CLBs and about 3000 nets. The *tabu search* based method did as good as VPR for the larger MCNC benchmarks but did not show significant speedup or quality gain.

Vicente et al. [Vicente 2004] proposed a method called *Thermodynamic Com-*

binatorial Optimization (TCO) for the FPGA placement problem. The method is derived from both thermodynamics and information theory. In TCO, two kinds of processes, termed as *micro state* and *macro state* transformations, are considered. Applying Shannon's definition of entropy to reversible micro state transformations, a probability of acceptance based on Fermi-Dirac statistics is derived. Applying the laws of thermodynamics to macro state transformations, an efficient annealing schedule is provided. Comparison with simulated annealing (SA) based VPR showed that TCO produces high quality results of SA, while inheriting the adaptive properties of *natural optimization.* The number of moves required and the CPU time taken by TCO is less than VPR in sixteen out of eighteen cases. The quality of the placement is comparable to VPR, and the speedup on the average is 1.36$\times$ over VPR. However, the method is quite complex overall. Further, this being a stochastic method, the run-time of TCO increases with design size, and hence is not likely to provide high scalability.

2.1.2 Partitioning based placement

In recursive partitioning based method, on one hand, the rectangular FPGA chip is divided into sub-rectangles by horizontal or vertical cuts, and on the other, the circuit is simultaneously partitioned such that each partition fits into the corresponding sub-rectangles and the number of nets going across partitions is minimized. This is the *minimum bisection problem* [Garey 1979] where the objective is to partition the set of vertices into equal sized subsets such that the number of edges having its two end points in the two different partitions is minimized. The problem being NP-hard, mostly heuristics [Fiduccia 1982, Kernighan 1970a, Alpert 1995] are employed.

The recursive partitioning based method such as *Partitioning based Placement For FPGAs* (PPFF) [Maidee 2003, Maidee 2005] has been proposed to build an effective coupling between the placement and routing stages. The method incorporates an accurate delay model and employs effective heuristic that minimizes *critical path delay.* It performs recursive bi-partitioning in a *breadth-first* manner. At each level of partitioning hierarchy, it considers *terminal alignment* by incorporating an *alignment_cost* in the VPR flow. For this, it uses the routing information of already routed circuits. PPFF also optimizes the partitioning order of the regions at the same level of hierarchy by a simple greedy algorithm. Finally, it goes through a legalization step and post-optimization step of low-temperature simulated annealing. The overall flow of the proposed approach is shown in Figure 2.2. First, a set of selected circuits are placed and routed using VPR. Then, the *routing pro-*

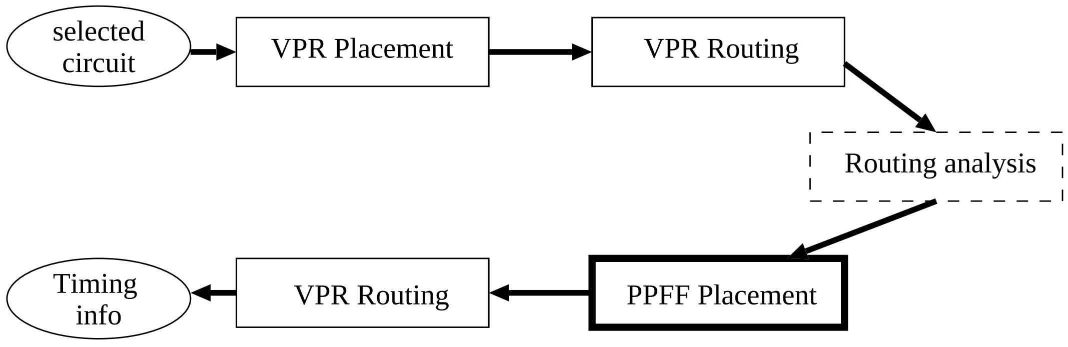


Figure 2.2: Flow of PPFF [Maidee 2003, Maidee 2005].

file information obtained from the already placed and routed circuit is used in the PPFF placement. Finally, the circuits are routed using VPR router. The results produced by PPFF show 2% improvement in delay over VPR with a penalty of 30% increase in run-time when the *routing profile* of individual circuit is used. When the *routing profiles* of only three representative circuits are used, the speedup is 3.6×. The regression analysis of up to 8000 blocks shows better scalability of the method compared to VPR.

Although the objective function is global in partitioning based approaches, a "good" cut at a certain level does not guarantee "good" cuts at subsequent levels. Their method includes a local *net terminal alignment* heuristic during each level of partitioning. PPFF is based on a reverse engineering technique where the *routing profile* statistics of a few already routed circuits are used. This requires additional pre-processing time. Moreover, to the best of our knowledge, no theoretical analysis has been provided for the key step of using routing profile statistics of a few already routed circuits for any given circuit and obtaining speedup. The method is especially effective for multiple placement runs on the same circuit.

2.1.3 Analytical placement

In *analytical placement*, the objective is to minimize the netlength, but block overlap is allowed by relaxing certain constraints. This leads to an easier placement problem which is formulated as certain variant of a mathematical program such as quadratic placement [Vygen 2007]. But, this entails a major bottleneck. Because of the constraint relaxation, block overlaps remain. Removing such overlaps, while maintaining the objective criteria of wirelength minimization, remains a daunting task and again does not scale up well to large circuits.

Quadratic placement algorithms belonging to the class of analytic placement, attempt to minimize total squared wire length by solving linear equations. The resulting placement tends to locate all cells near the center of the chip with a large amount of overlap. As the sum of the square of the wirelengths is only an indirect measure of linear wire length, the resulting total wire length may not be minimized. Xu et al. in their work *Quadratic Placement for FPGAs* (QPF) [Xu 2005] have proposed a placement algorithm based on quadratic placement. It builds and solves *linear equations* repeatedly to produce the placement. The placements generated in each iteration might not be legal and thus requires additional heuristic to obtain a legal placement by pulling nodes out of the dense area while minimizing linear wirelength. QPF also incorporates a low-temperature simulated annealing step to improve the quality of the solution. Experimental results indicate that, on an average, QPF is 5.8$\times$ faster compared to the well known FPGA placement tool VPR, while providing almost comparable estimated total wirelength. However, QPF does not report the *critical path length* obtained for the placement. Also, the channel width increases during routing.

Gopalakrishnan et al. in their work on *Convex Assigned Placement for Regular ICs* (CAPRI) [Gopalakrishnan 2006], have proposed an architecture aware analytical placement using *graph embedding* and *metric geometry* [Matousek 2002] that models the relationship between performance and the routing grid. The placement problem is modeled as an embedding of a graph representing the netlist into a metric space that is representative of the FPGA architecture. An analytic metric of distance that models delays along the FPGA routing grid has been developed. They use a metric space that accurately captures the delays on the FPGA chip, rather than the *Manhattan space*. Then the netlist is embedded into the defined metric space using matrix projections [Golub 1983] which produces an illegal placement with overlapping CLBs. The placement is legalized using a *online bipartite matching* [Gopalakrishnan 2006] formulation. Finally, a low-temperature simulated annealing step is performed to improve the solution further. The overall flow of CAPRI is given in Figure 2.3. Experimental comparisons with the popular FPGA tool VPR, show that with CAPRI's initial solution, the resulting placements have median improvements of 10% in *critical path delays* for the larger MCNC benchmarks. The total placement run-time improved by 2$\times$ on the average but the average or the range of run-times were not available in the published literature. It appears that the step of *online bipartite matching* is used iteratively and not once. Hence, it is likely to affect the scalability for large arrays. Analysis on the number of iterations required by the legalization step would have added strength of this method.

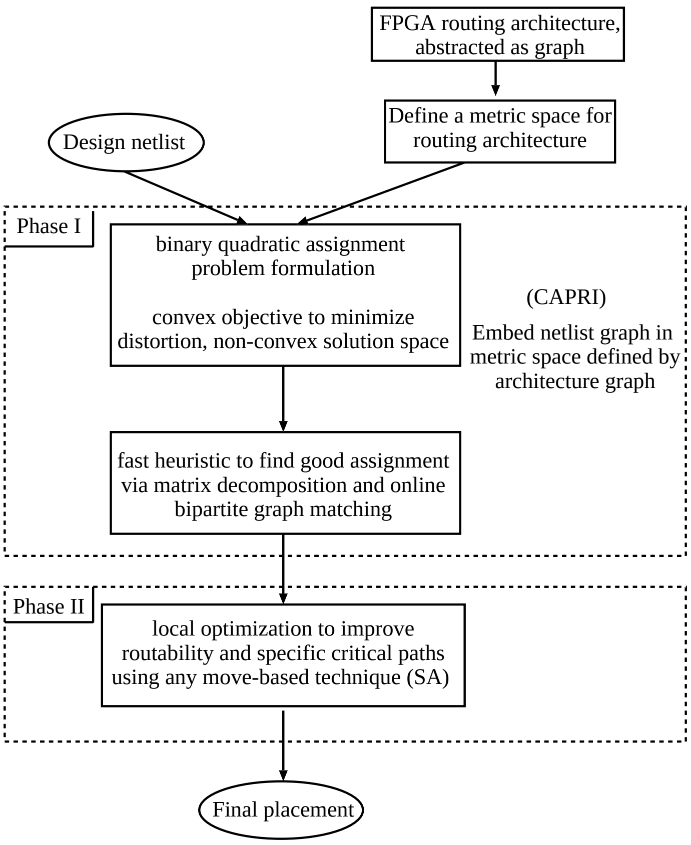

FPGA routing architecture, abstracted as graph
Define a metric space for routing architecture
Design netlist
Phase I
binary quadratic assignment problem formulation
convex objective to minimize distortion, non-convex solution space
(CAPRI)
Embed netlist graph in metric space defined by architecture graph
fast heuristic to find good assignment via matrix decomposition and online bipartite graph matching
Phase II
local optimization to improve routability and specific critical paths using any move-based technique (SA)
Final placement

Figure 2.3: Overview of CAPRI (Courtesy: [Gopalakrishnan 2006]).

2.1.4 Other placement techniques

Sankar et al. have proposed *Ultra-Fast Placement (UFP)* in [Sankar 1999] aiming to improve the run-time of the VPR placement tool, which produces good placement results but is not very scalable due to the use of simulated annealing. It employs *multi-level* optimization and *multi-level* clustering. The algorithm grows a cluster of blocks based on a connectivity based scoring function. The function includes (i) the strength of connection between a block and a cluster, and (ii) the number of nets that will get absorbed in a block if merged with the cluster. As a result, a netlist of clusters is obtained. A hierarchy of clusters is built in the same way by merging clusters. Finally, a low temperature simulated annealing is performed at each level of clustering hierarchy. It achieved 50× speed-up over VPR whereas the wirelength overhead is 33%.

2.1.4.1 IOB placement

There are a few works on the placement of IOBs in the literature. Khaled and Rose have experimentally studied the impact of fixing the positions of IOBs in a random fashion [Khalid 1995]. The experiments show that fixing the assignment of signals to pins in a random fashion can cause an increase in delay up to 19% in worst case and significantly impact the routing resources needed to complete the routing. For the Xilinx XC4000 architecture, random pin constraints caused an increase of up to 20% more single length interconnect segments, 11% more double length interconnect segments, and 49% more long length interconnect segments, although no routing failures occurred.

Anderson et al. have proposed [Anderson 2000] a placement algorithm that uses a combination of *simulated annealing*, *weighted bipartite matching* and constructive *packing* to produce a feasible IOB placement. Results show that the proposed algorithm produces placements with wirelength characteristics that are similar to the placements produced when pad placement is unconstrained.

Mak [Mak 2004, Mak 2005] has proposed the first exact approach to solve the constrained IOB placement problem for FPGAs that support multiple I/O standards. The author derives a compact *integer linear program* (ILP) formulation for the *constrained IOB placement* problem. The size of the ILP derived depends on the classification of the IOB type rather than the number of IOBs to be placed, and hence scalable to very large design instances. For a Xilinx Virtex-E FPGA, the number of integer variables required is never more than 32 and is much smaller for practical design instances. Extensive experimental results using a non-commercial ILP solver show that it takes only seconds to solve the resultant integer linear program in practice. In addition, the author also proposes a placement flow to place both core logic and IOBs. The method places the IOBs respecting a set of I/O banking rules corresponding to the multiple I/O standards. Then, the core logic, the CLBs are placed using simulated annealing. Thus, IOBs are placed without considering the connectivity of the IOBs and the CLBs. In this thesis, we address the issue of IOB placement with respect to the *netlist* and not the rules of I/O banks.

2.1.4.2 Combined placement and routing

Nag et al. have proposed a combined approach [Nag 1998] where a fast router is embedded inside the inner loop of a simulated annealing based placement engine. Although, there is performance improvement of 8% to 15% over the commercially available tools at that time, the run-time overhead was very high. It took 6$\times$ to

11× more time than the commercial Xilinx tool XACT [Xilinx].

Alexander et al. [Alexander 1998] have proposed an integrated approach, where a global router was embedded in a partitioning based placement algorithm. This was more scalable, but there was no result demonstrating the superiority of this combined approach.

A more recent work proposed by Chang et al. [Chang 2000] combines a cluster growth placer with maze router, where nets are placed and routed one by one using a cost function. However, this does not report any comparison with the commonly used flows. Hence, nothing could be concluded on the effect of this combined approach.

2.1.5 Summary

All of the above works, specially the CLB placement, aim at faster method for routable placement with better or comparable critical path delay and/or wirelength as in VPR. Almost all the works except VPR, utilize various heuristic techniques to generate a good initial solution optimizing delay, and then employ a low-temperature simulated annealing to produce the final solution. This indicates that the solution produced by those approaches are not close to the quality of the solution produced by the simulated annealing based method VPR. Nevertheless, the solution by these methods are better than a random initial placement, and so the simulated annealing converges faster in all the three types of heuristics detailed above. Further, the methods which produced better solution than VPR, had to sacrifice in run-time.

Although all of the above methods attempted at generating high quality initial solution, no work addresses the quality of the initial placement obtained by their deterministic method, either theoretically or experimentally. All the works report the quality of the final solution after execution of the low temperature simulated annealing. The HPWL, routability, delay, congestion of the initial placement are not available in the literature. Moreover, none of the above approaches give a theoretical guarantee on how close their initial placement solution is to the near-optimal solution. In this thesis, we propose both bottom-up and top-down deterministic placement approaches for island-style FPGAs, and derive theoretical bound on the quality of the solution produced by our partition based top-down approach.

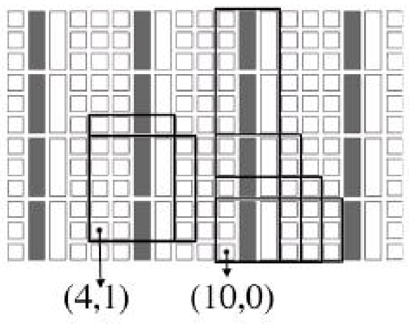
2.2 Floorplanning for heterogeneous FPGAs

With the advent of technology, current FPGAs not only have the array of CLBs and IOBs but also RAM, Multiplier blocks, DSP cores and even processor cores pre-placed at specific locations on the chip. Thus, the netlist of CLBs and other resources need to be placed in appropriate positions on the target chip such that the performance of the design is optimized. Modern FPGAs with million of gates are now capable of implementing large designs on to it. The heterogeneity of FPGAs and the large size of designs, have compelled researchers to investigate floorplanning for FPGAs.

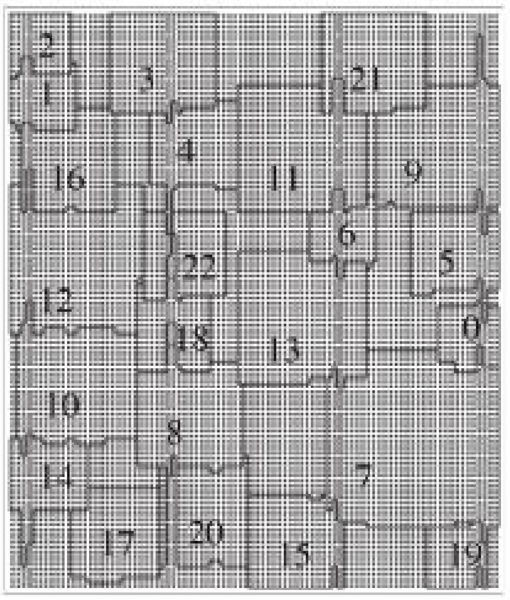
Emmert et al. [Emmert 1998, Emmert 1999a] devised a macro based floorplanning methodology for earlier generation island-style FPGAs. It uses *clustering* techniques to combine *macros* into *clusters*. Then the clusters are placed using *tabu search* based heuristic enhancing the circuit routability, reducing total wirelength by means of *terminal propagation*. Using the Xilinx XC4000 series of FPGAs as the target architecture, the work demonstrated effectiveness of this fast floorplanning method on a collection of designs. However, the clustering strategy needs extensive modification for handling modern FPGAs with heterogeneous resources.

Tessier [Tessier 2002] developed a timing-driven FPGA placement system, *Frontier*, that uses macro blocks in conjunction with a series of placement algorithms to achieve highly routable and high-performance layouts quickly. In the first stage of placement, a macro-based floorplanner is used to quickly identify an initial layout based on inter macro connectivity of *soft* and *hard macros*. A given FPGA architecture is decomposed into an array of placement bins with matching dimensions. Macros are clustered and each cluster is placed in a *bin*. If the available bins are insufficient to place all clusters, the bin size is increased and the process is repeated. Finally, after allocation of clusters to bins, entire bins are swapped using simulated annealing, minimizing inter-bin placement cost. For a collection of large reconfigurable computing benchmark circuits, *Frontier* exhibits a 2.6$\times$ speedup in combined place and route time with improved design performance compared to the commercial FPGA CAD tool Xilinx PAR for most designs. The work shows that floorplanning, placement evaluation, and back-end optimization are all necessary to achieve high-performance placement solutions. However, the work dealt with netlist of CLBs only. With multiple types of pre-placed resources on the modern FPGAs, this method seems unlikely to be adapted.

Cheng and Wong [Cheng 2004, Cheng 2006] proposed the first floorplanning algorithm targeted for *heterogeneous* FPGAs with multiple types of resources which



(a)



(b)

Figure 2.4: Example of floorplan obtained by [Cheng 2004, Cheng 2006]. (a) Possible *irreducible realization list* (IRL)s for two modules (b) a rectilinear floorplan after compaction (Courtesy: [Cheng 2004, Cheng 2006]).

produced feasible solution employing simulated annealing, optimizing the area, half-perimeter wirelength (HPWL) and the *aspect ratios* of modules. The authors began with a *slicing topology* [Otten 1982] and used simulated annealing to iteratively perturb and improve it. They used the concept of *irreducible realization list* (IRL) which is a set of realizations of distinct (width, height) pairs for each node in a *slicing tree*. Given an IRL, they devise efficient means to compute IRLs as nodes are merged from leaf level up to root, thereby finding optimal realizations of a particular slicing tree. An example of possible IRLs for two modules is shown in Figure 2.4(a). The rectangles with darker border lines show the possible IRLs for two modules, whose bottom-left corners are anchored at positions $(4, 1)$ and $(10, 0)$. The method is an extension of Stockmeyer's floorplan optimization algorithm [Stockmeyer 1983]. They also proposed a post-processing step of compaction, generating modules with rectilinear realization as shown in Figure 2.4(b). The proposed method may not scale well, being simulated annealing based.

Yuan et al. [Yuan 2005] have proposed a constructive packing-based method using *Less Flexibility First* (LFF) principle. A set of all *atomic realizations* (ARL) of a module is generated from the resource requirement and a starting corner position on the chip. The realizations are sorted based on the value of a *flexibility function* defined for the current configuration of packed modules on the chip. Thereafter, all realizations of different modules are collected into a list. The realization with the highest *fitness value* is selected. As to the worst case time complexity, the authors point out that their method takes $O(W^2 n^5 \log n)$ where n is the number of modules and W is the width of the chip. The authors claim that the $\log n$ factor in the time complexity comes from *range searching* with the help of a *kd-tree.* But, *kd-tree* takes $O(\sqrt{n})$ ($O(n^{1-\frac{1}{d}})$, where d is the dimension) for range searching in 2D [Berg 2000]. Time complexity proportional to $\log n$ is achievable if fractional cascading [Berg 2000] is used, but then space complexity would go up to $O(n \log n)$. The quality of the floorplans can not be compared as the benchmarks are synthetically generated.

Recently, Feng and Mehta [Feng 2006] presented a two step approach based on *resource-aware fixed-outline* simulated annealing starting from a given topology, followed by *maximum-flow* based *constrained floorplanning* to optimize wirelength. As FPGA is a bounded rectangle, the authors [Feng 2006] propose a fixed outline simulated annealing algorithm in contrast to the area minimization approach of [Cheng 2004]. The authors use a penalty term in their simulated annealing cost function so that modules are placed as close as possible to their resources. The shortcoming of each module in meeting its resource requirement is then taken care of by a constrained floorplanning which is based on a *minimum-cost maximum-flow* network formulation. This constrained floorplanning is a generalization of the method in [Feng 2004] that was developed for ASICs.

In [Singhal 2007b, Singhal 2007a], Singhal et al. have proposed another SA based floorplanner with an adaptive placer algorithm. The floorplanner handles each resource type individually. It allows for different placements of different resources of a module. Hence, it utilizes the available area better, optimizing the internal wirelength of the overall floorplan. The experiments show that the proposed floorplanner can reduce the area by as much as 50% of the area of a disjoint floorplanner. This *multi-layer* floorplanning algorithm reports better floorplan area for heterogeneous resources of statistically large variations. However, the run-time is 1.4× of the traditional non-heterogeneous floorplanner.

2.2.1 Summary

In summary, excepting the LFF method, all the floorplanning methods for heterogeneous FPGAs are simulated annealing based and thus incurs significant run-time. This raises concern about the scalability of all these methods for large designs. In

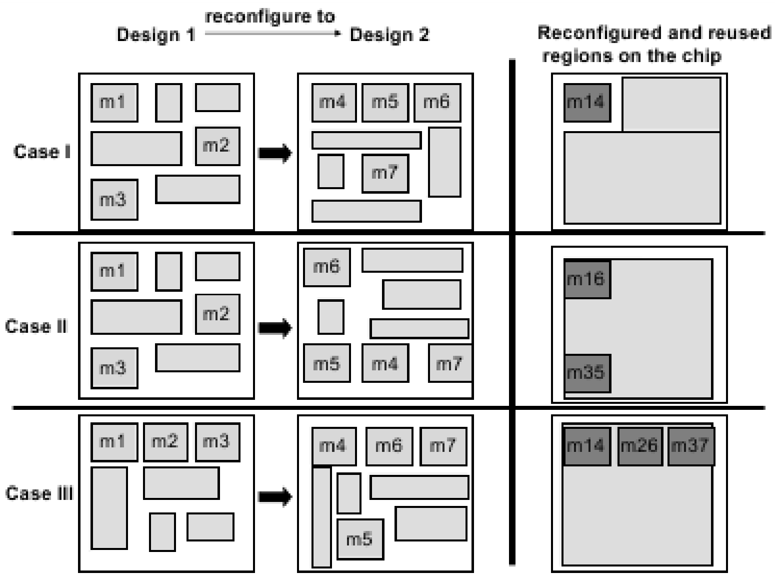


Figure 2.5: Partial reconfiguration proposed by Singhal and Bozorgzadeh [Singhal 2006]; case III shows the maximum overlap of common modules. (Courtesy: [Singhal 2006]).

this thesis, a fast, deterministic unified floorplan topology generation and sizing method is proposed which improves the run-time as well as the quality significantly compared to the earlier approaches such as [Feng 2006, Cheng 2006].

2.3 Floorplanning for partial reconfiguration

Partial dynamic reconfiguration is an emerging area in FPGA designs [Xilinx] which is used for saving device area and cost. In order to reduce the reconfiguration overhead, two consecutive similar sub-designs should be placed in the same location to get the maximum reuse of common components. This requires that the entire *sequence* (schedule) of tasks for an application be taken into consideration while floorplanning for the design of any single *instance* in the *schedule*. In the FPGA literature, floorplanning a set of modules in multiple instances is only a handful.

The earliest work on floorplanning for partial reconfiguration was formulated as a 3D template placement problem in [Bazargan 2000], time being the third dimension. The work focuses on both online and offline placement algorithms. For the online problem, a fast but non optimal and a slow yet high quality placement algorithm have been proposed. For the 3D placement of *reconfigurable units*, simulated annealing based and greedy placement methods have been devised.

Singhal and Bozorgzadeh [Singhal 2006] have introduced a new *multi-layer sequence-pair* representation based floorplanner which allows overlap of common (*static*) and dynamic (*non-static*) components of multiple designs of a *schedule*, and guarantees feasible, overlap-free floorplans with minimal area packing. Figure 2.5 shows this approach. It maximizes the overlap of common components of multiple designs thereby reducing reconfiguration overhead. Multi-layer sequence pair is an efficient representation and helps in reducing the total floorplanning run-time. Experimental results showed that the proposed floorplanner removes infeasibility in designs, achieves an improvement of clock period by 12% on an average and reduces the place-and-route time by as much as $3\times$ compared to a traditional sequential floorplanner. It also reduces the average wirelength by 50% in the designs. However, owing to the use of simulated annealing, the execution time is significantly high.

Ahmadinia et al. [Ahmadinia 2007] have proposed an algorithm for online *optimal free space management* and *routing conscious dynamic placement* for reconfigurable devices. The work describes algorithmic results on two crucial aspects of allocating resources on computational hardware devices with partial reconfigurability. With computational geometric techniques, their algorithm allows correct maintenance of free and occupied space of a set of n rectangular modules in time $O(n \log n)$. Previous approaches needed a time of $O(n^2)$ for correct results and $O(n)$ for heuristic results. This work also gives a matching lower bound of $\Omega(n \log n)$. The authors also show that, finding an optimal feasible communication-conscious placement which minimizes the total weighted *Manhattan distance* between the new module and existing demand points can be computed in time $\Theta(n \log n)$. Both the resulting algorithms are practically easy to implement and demonstrate convincing experimental behavior.

2.3.1 Summary

The work of Ahmadinia et al. [Ahmadinia 2007] addresses the management of free spaces during the partial reconfiguration and not the issue of maximizing the overlap of common modules. The formulation of the floorplanning method for partial reconfiguration addressed in this thesis is similar to that in [Singhal 2006]. However, the challenge is to overcome the drawback of long execution time of their simulated annealing based approach. This thesis presents a deterministic, fast method for generating a *global floorplan* that minimizes the partial reconfiguration overhead while the HPWL of the overall floorplan is optimized.

2.4 Conclusion

In this chapter, a survey of different placement approaches for island style FPGAs is done. Almost all the works aim at obtaining faster solution minimizing the wiring cost. With modern FPGAs having heterogeneous resources, floorplanning a netlist of modules on such a target chip has gained importance in the CAD community. Most of the earlier methods of floorplanning on heterogeneous FPGA use *simulated annealing framework* as in the case of placement. Floorplanning a set of modules in different instances of a schedule minimizing the reconfiguration overhead, is a challenging task in the context of partial reconfiguration. The issue of generating faster placement and floorplanning methods for *full* as well as *partial reconfiguration* has been dealt with in detail in the remaining chapters.

CHAPTER 3

Bottom-up Cone based Placement for Island-style FPGAs

Contents

3.1 Introduction

In this chapter, deterministic greedy methods are proposed to place the netlist of CLBs and IOBs quickly on a two dimensional island-style FPGA array consisting primarily of CLBs and IOBs. The locations of each type of resources, the CLBs and IOBs are fixed on the FPGA array with the CLBs placed in the inner portion and the IOBs on the periphery of the two dimensional array of CLBs. Here, we present three different placement approaches. The first method, *ConeCLBPlace*, places the *primary output blocks* first at random, and then the CLBs and *primary input blocks* in a deterministic way. In the second method, *ConeIOBPlace*, the IOBs are first placed on the periphery of the FPGA chip using a deterministic heuristic, and then the CLBs are placed within the two dimensional array using a typical *simulated annealing* flow. Finally, the third method, *ConePlace*, combines the above two deterministic approaches by first placing the IOBs only as per the second method, and then the CLBs by the first method deterministically. For further improvement of quality of the solution, we employ a very low temperature simulated annealing.

The rest of the chapter is organized as follows. Section 3.2 has the formulation of the placement problem for an island of CLBs and a brief description of the proposed methods. Section 3.3 describes the first bottom-up *cone* based deterministic method, *ConeCLBPlace*, which places the CLBs keeping the *output blocks* at some fixed positions. Section 3.4 presents the second bottom-up *cone* based deterministic method, *ConeIOBPlace* for placing the IOBs. Finally, Section 3.5 describes the combined method of deterministic IOB and CLB placement. Experimental results for all the three methods have been presented together in Section 3.6 and the concluding remarks appear in Section 3.7.

3.2 Background

Problem 3.1 (FPGA Placement) *Given*

- *a technology mapped netlist of a set of n CLBs* $C = \{c_1,\ c_2,\ \cdots,\ c_n\}$, $|C| = n$,
- *a set of p IOBs* $P = \{p_1,\ p_2,\ \cdots,\ p_p\}$,
- *a set of k signal nets* $S = \{S_1,\ S_2,\ \cdots,\ S_k\}$, *where each* S_i *is a set of connected CLBs* $c_i \in C$,
- *a set of locations in a* $\sqrt{n} \times \sqrt{n}$ *two dimensional array,*

the placement problem is to assign each CLB $c_i \in C$ to a unique location on the $\sqrt{n} \times \sqrt{n}$ array and each IOB $p_i \in P$ on the peripheral location of the array, such that the objective function relating to total wirelength is minimized.

We assume the smallest possible 2D array of size $\sqrt{n} \times \sqrt{n}$ for placing the n CLBs. Since IOBs are placed along the periphery of $\sqrt{n} \times \sqrt{n}$ array, the number of IOBs is $|P| = p \approx O(\sqrt{n})$. Each location on the 2D array is represented by a unique location $(x_j,\, y_j)$ on the 2D array where x_j and y_j are positive integers. The *bounding box* of a net is the minimum enclosing rectangle on the FPGA array that encloses the positions of all the CLBs in the net. We consider the sum of the half-perimeter of *bounding boxes* enclosing the CLBs of each net, i.e, the *HPWL cost* or *BB cost*, as the objective function to be minimized. Here, to compare the quality of a placement, we consider an extension of the HPWL cost, namely, *linear congestion cost* function proposed in VPR [Betz 1999, Betz 1997]. We use this extended *BB cost* for all our experiments in this chapter. The extended *BB cost* function which is already given in Section 2.1.1 of Chapter 2 is given here once more for the sake of completeness.

$$BB\ cost = \sum_{i=1}^{k} q(i) \left[\frac{bb_x(i)}{C_{av,x}(i)^{\beta}} + \frac{bb_y(i)}{C_{av,y}(i)^{\beta}} \right] \tag{3.1}$$

Most of the earlier works focus on obtaining high quality placement by various stochastic methods for large CLB netlist at the expense of long placement time, which might affect the advantage of reconfigurability in FPGA. In contrast, we focus on obtaining a fast deterministic heuristic for placing already partitioned netlist of CLBs in small islands of CLB arrays present in both island-style, as well as modern day heterogeneous FPGAs, with comparable quality of solution.

3.2.1 Overview of our method

Our proposed approach divides the placement problem into two subsequent phases, (1) placement of IOBs on the periphery of the chip, followed by (2) the placement of CLBs within the chip array.

Given a placement of *output blocks*, a *cone* based method, *ConeCLBPlace* for placing CLBs, is proposed first. Second, we propose a method, *ConeIOBPlace*, for placement of only IOBs, and then placement of the CLBs by a simulated annealing method such as VPR. Finally, the above two methods are combined together into *ConePlace* by first placing the IOBs with *ConeIOBPlace* and then the CLBs with *ConeCLBPlace*. Figure 3.1 depicts the flow of our methods. With all the above three methods, the quality of the produced placement suffered vis-a-vis the state-of-the-art tool VPR. A post-processing step of low temperature simulated annealing produced the desired result quickly. In fact, this offers a choice to the designers to opt for the required trade off between time and quality.

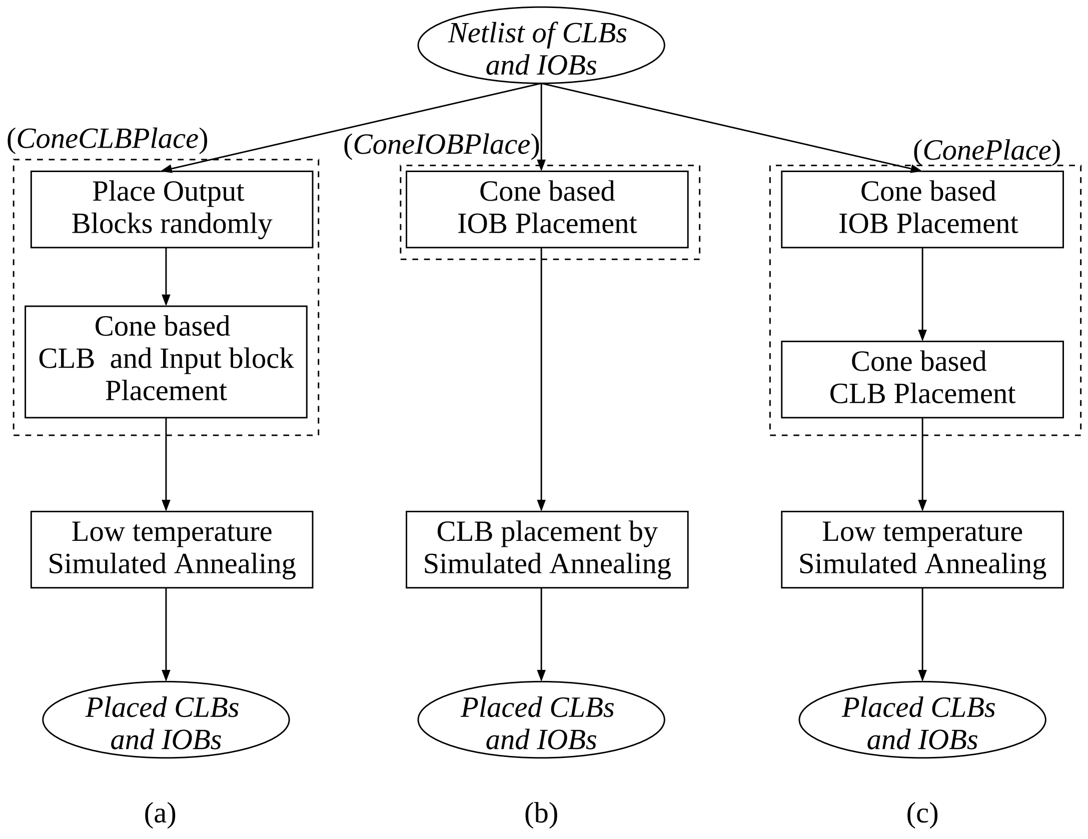


Figure 3.1: Flow of our methods: (a) *ConeCLBPlace*; (b) *ConeIOBPlace*; (c) *ConePlace.*

3.3 *ConeCLBPlace*: Cone based CLB placement

3.3.1 Placement of output blocks

The size of the minimum 2D square array required to place the n CLBs is $\sqrt{n} \times \sqrt{n}$. We place the *primary output blocks* randomly on the periphery of the minimum square array. Then we place the CLBs with respect to the positions of these placed *primary output blocks*.

3.3.2 Construction of an *output cone* for placement of CLBs

The netlist of CLBs and IOBs is defined as a *directed graph* as follows.

Definition 3.1 (Block netlist graph:) *A block netlist graph is a directed graph $D^b = (V, E)$, where $v_i \in V$ corresponds to a CLB or an IOB, and has an edge $e = (v_i, v_j) \in E$ if the CLB or output block corresponding to v_j receives its input signal from the CLB or input block corresponding to v_i. $|V| = n + p \approx O(n)$ and $|E| = O(n)$, where n and p are respectively the number of CLBs and IOBs. The number of edges are $O(n)$ due to the fact that a CLB has a bounded number of inputs and outputs, typically a small constant.*

The CLBs or *output blocks* (v_j) receiving input signals from a CLB or an *input block* (v_i) are the *fanout blocks* of v_i. The set of v_is, each corresponding to the *source* of a *signal net*, which are fed as input to a CLB or an *output block* v_j, constitute the *fanin* of the block v_j. Since CLBs in a given FPGA have fixed number of inputs, the number of *fanin blocks* of any CLB is assumed to be a constant or bounded (typically 4), and specified as an input to the placement problem. It is obvious that the number of *fanout blocks* of a CLB or an *input block* may vary. But the average number of *fanout blocks* has to be a constant to tally with constant number of *fanin blocks.*

Definition 3.2 (Predecessor:) *A predecessor of a vertex v_i in the directed graph D^b is a vertex v_j, such that there is a directed path from v_j to v_i in the graph D^b.*

Definition 3.3 (Output cone:) *An output cone of a primary output block $p^o \in P$ is the breadth-first search tree $\tau^o = (V_\tau, E_\tau)$ in D^b, with primary output block p^o as the root.*

Two such *output cones* are shown in Figure 3.2. The *predecessor* vertices of an *output block* belong to the *output cone* of the corresponding block. Thus, for ease of computation, the directed edges of D^b are reversed while obtaining the *output cones* by breadth-first search (BFS) traversal. Using the above definitions, an *output cone* is constructed as follows. A primary output (p^o) already placed on the periphery of the array, is chosen at random. The *output cone* τ^o of p^o is extracted from the directed graph D^b by BFS traversal in D^b starting from p^o. The *breadth-first traversal* of the graph generates a *tree* with root vertex corresponding to p^o and the CLBs and *input blocks* as its leaves. This process is repeated for all the *primary output blocks.*

3.3.3 Placement of CLBs and input blocks

Having constructed the *output cones* as above, we place the *CLBs* and *primary inputs* now. One CLB or *input block* at a time is chosen for placement in *breadth-first* order

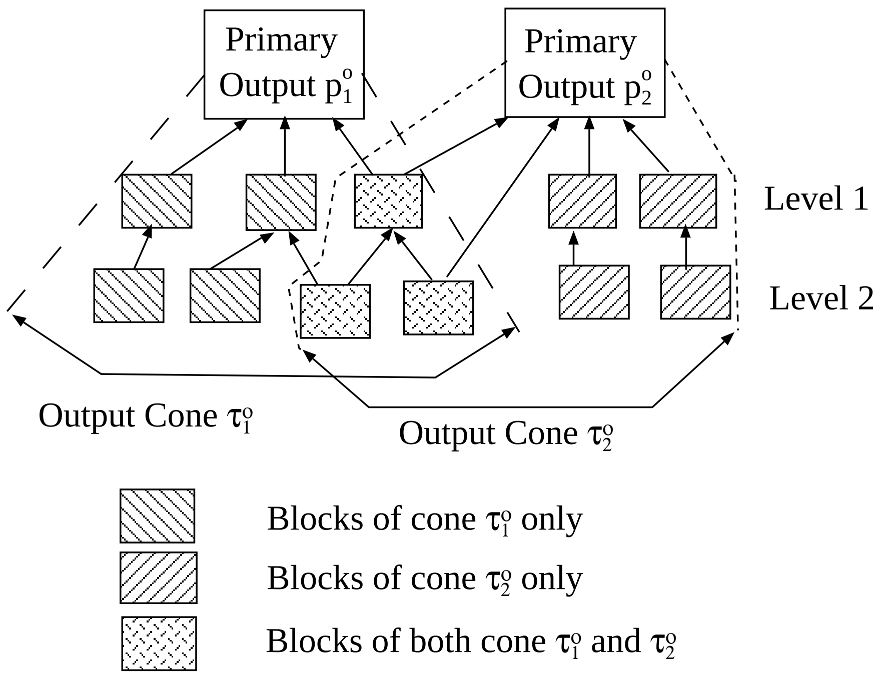

Primary Output p_1^o
Primary Output p_2^o
Level 1
Level 2
Output Cone τ_1^o
Output Cone τ_2^o
Blocks of cone τ_1^o only
Blocks of cone τ_2^o only
Blocks of both cone τ_1^o and τ_2^o

Figure 3.2: Two *output cones* with overlap corresponding to two *primary output blocks*, p_1^o and p_2^o.

from the *cone*. The chosen block is placed in a *minimum-cost position* with respect to the current placement configuration. Let b_i be the block chosen in the breadth-first order to be placed in the array. In order to determine the *minimum-cost position* with respect to the current placement configuration, we define a *bounding box* for the *block* b_i as follows.

Definition 3.4 (Bounding box of a block:) *The bounding box of a block b_i, denoted by BB_i, is the smallest rectangular region on the two dimension array, containing all blocks corresponding to $b_j \in fanout(b_i) \cup fanin(b_i)$ which have been already placed.*

Definition 3.5 (Net length:) *The net length of two connected blocks b_i and b_j already placed in the FPGA array, is the manhattan distance between the two blocks b_i and b_j.*

Thus, the current bounding box of b_i is a two dimensional sub array with already placed *fanout* and *fanin* CLBs or *input blocks* of b_i. All the *fanin* and *fanout* blocks of a CLB may not be placed at an intermediate step. Let b_j be the set of CLBs or *input blocks* that are adjacent to block b_i in D^b and already placed. We compute the *net length* of each pair (b_i, b_j) by placing b_i tentatively in all empty positions within

	p_0^i	p_1^i	p_2^i	p_3^i	p_4^o	p_5^o	p_6^o	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}
p_0^i	-	-	-	-	-	-	-	-	1	-	1	-	1	1	-	1	-
p_1^i	-	-	-	-	-	-	-	-	-	-	-	1	-	-	1	-	1
p_2^i	-	-	-	-	-	-	-	1	-	1	-	-	1	-	-	1	1
p_3^i	-	-	-	-	-	-	-	-	-	1	1	-	-	-	-	-	1
c_7	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-
c_8	-	-	-	-	-	-	-	1	-	-	-	-	1	-	1	1	-
c_9	-	-	-	-	-	-	-	1	-	-	-	-	-	-	1	-	1
c_{10}	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-
c_{11}	-	-	-	-	-	-	-	-	1	-	1	-	1	-	-	-	-
c_{12}	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-
c_{13}	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-
c_{14}	-	-	-	-	-	-	-	-	-	-	-	1	-	1	-	-	-
c_{15}	-	-	-	-	-	-	-	-	-	1	-	-	-	1	-	-	-
c_{16}	-	-	-	-	-	-	-	-	1	-	-	-	-	-	1	-	-

Table 3.1: An example: Adjacency matrix representing the block netlist graph D^b, the netlist of CLBs and IOBs; p^i, p^o and c denote the *primary input blocks*, *primary output blocks* and CLBs respectively.

the bounding box of block b_i. The block b_i is then assigned to a position within BB_i that results in minimum increase in *net length* of all the nets associated with the placed *fanin* and *fanout* blocks of b_i. If there are no empty slots left within BB_i, we extend the bounding box by either one row or one column, and place the *block* b_i in the new BB_i with the same objective of minimizing the increment in wirelength.

As stated earlier, only the *primary output blocks* are placed on the periphery of the chip initially. A single *output cone* corresponding to the *primary output block* p^o is chosen at random. The *output cone* τ^o is traversed in *breadth-first* order. The first CLB b_i chosen while traversing, is placed within a BB_i which contains only the placed *output block* p^o as no other *fanin* and *fanout* CLBs of b_i is placed yet. This process is repeated for each of the *CLBs* and *primary inputs* present in the *cone* τ^o in the *breadth-first* order. As the process progresses, it finds more and more placed blocks to compute the position of a new block accurately. Thus, our greedy heuristic attempts to place each block at a locally optimal position with respect to the current configuration. We process all the *output cones* of the given *netlist* one by one in an arbitrary order. This gives the initial placement configuration for the given *technology-mapped netlist.*

3.3.4 Illustration of CLB placement by *ConeCLBPlace*

Let us consider a *technology-mapped netlist* with ten *CLBs*, four *primary inputs* and three *primary outputs* as shown in Table 3.1. The table shows the adjacency matrix of the directed graph D^b. The vertices p_0^i to p_3^i corresponding to four primary inputs, p_4^o to p_6^o correspond to the three primary outputs, and the remaining c_7 to

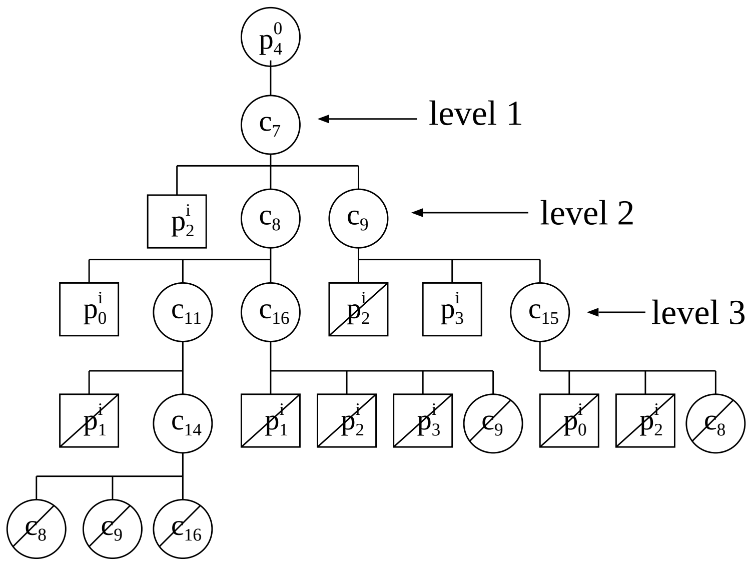


Figure 3.3: Illustration of *ConeCLBPlace*: An *output cone* τ_4^o of *primary output block* p_4^o of the netlist in Table 3.7.

c_{16} corresponds to the CLBs. The entries in the matrix corresponding to the *fanout* of *input blocks* and CLBs are also given. Since p_4^o to p_6^o are *primary output blocks*, they do not have any fanout blocks and hence the rows corresponding to p_4^o - p_6^o are omitted from the table.

The *output cone* τ_4^o, of *primary output block* p_4^o is shown in Figure 3.3. A *strikethrough* index in the *output cone* τ_4^o indicates that the tree need not be traversed any further. For example, all the children of c_{16} at level 3 in the *cone*, i.e., c_9, p_1^i, p_3^i and p_2^i are already placed in the previous levels. The indices that correspond to the *primary inputs* are not expanded at any level.

The execution of the algorithm *ConeCLBPlace* for all the three *output cones* is shown in Figure 3.4. The random placement of all those *primary outputs* as in VPR is shown in Figure 3.4(a). Next, we trace the *cone* of *primary output* p_4^o. By tracing the *cone* we find c_7 in the tree, whose *bounding box* and placement is shown in Figure 3.4(b). Next we find p_2^i in the tree and so on. Figures 3.4(c) shows the placement of all other *blocks* in the *cone* of p_4^o. Figure 3.4(d) shows the placement configuration after all the output *cones* τ_4^o, τ_5^o and τ_6^o of the given *netlist* are placed.

3.3.5 Time complexity of *ConeCLBPlace*

The proposed method is summarized in Algorithm 3.1.

Theorem 3.1 *The time complexity of ConeCLBPlace is* $O(n^2)$*, where n is the num-*

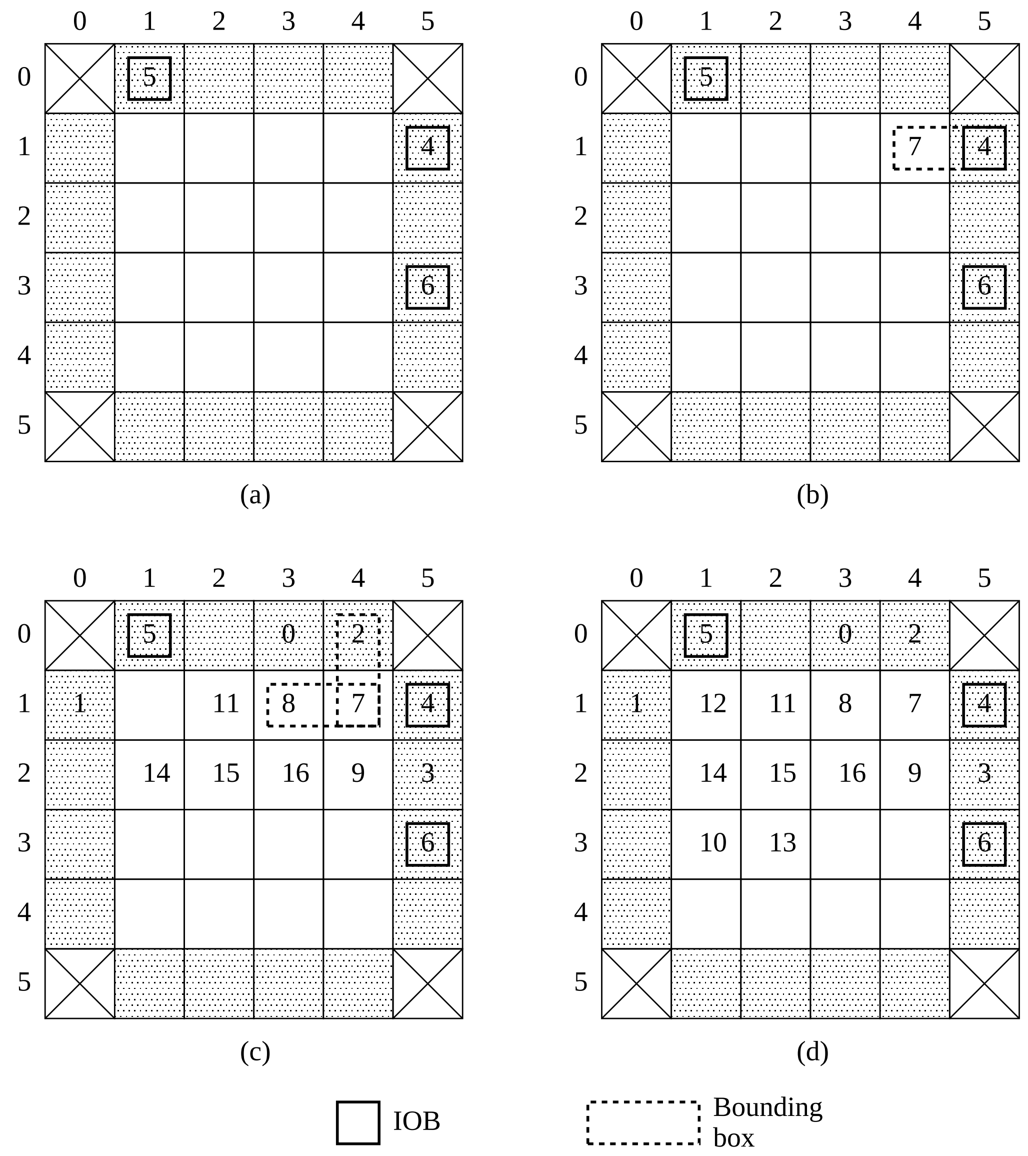


Figure 3.4: Execution of *ConeCLBPlace* on the example netlist of Table 3.1; (a) placement of IOBs; (b) placement of c_7 within the extended *bounding box* BB_7; (c) placement of all blocks in the *cone* τ_4^o; (d) placement of all blocks in the *cones* τ_4^o, τ_5^o and τ_6^o.

ber of CLBs.

Proof: Let the number of *primary input* and *output blocks* be n_{pi} and n_{po} respectively. The time complexity of placing the n_{po} *primary output blocks* on the periphery takes $O(n_{po})$ time. The time complexity of *breadth-first* traversal of a graph $G(V, E)$ is $O(|V| + |E|)$ [Alsuwaiyel 1999, Cormen 2003], $|V|$ and $|E|$ being the number of vertices and edges of the graph respectively. For the directed graph

Algorithm 3.1: ConeCLBPlace

```
Input  : Graph $D^b$ corresponding to the netlist of CLBs and IOBs
Output: Placement of CLBs and IOBs on the two dimensional array
1  Compute size of two dimensional array large enough to place all CLBs within
   array and IOBs on the periphery;
2  Place output blocks randomly on the periphery;
3  for each primary output block $p^o$ do
4      Extract the output cone by breadth-first traversal of the input graph $D^b$
       starting from $p^o$ ;
5      for each block $b_i$ to be placed do
6          Compute the bounding box $BB_i$ of $b_i$ ;
7          if there is an empty position in $BB_i$ then
8              Place $b_i$ at a position in the bounding box causing minimum
               increase in net length of all the nets associated with the fanin and
               fanout blocks of $b_i$ ;
9          else
10             Increase the size of the bounding box by either a row or column;
11             Place $b_i$ at a position with minimum increase in wirelength;
   /* Initial placement of CLBs and IOBs                                    */
12 Iteratively improve the quality of placement by execution of a low
   temperature simulated annealing in VPR framework;
   /* Final placement of CLBs and IOBs                                      */
13 Report placement;
```

D^b, $|V| = n + p$, and $|E| = O(n)$, where n and p are the number of CLBs and IOBs respectively. The time complexity of constructing a *cone* is $O(n+n) = O(n)$. Thus, to generate n_{po} *cones* corresponding to n_{po} *primary output blocks*, the time complexity is $O(n_{po} \cdot n) = O(n^{1.5})$, as $n_{po} \approx O(\sqrt{n})$ in our case. The size of the *bounding box of a block* can be at most $\sqrt{n} \times \sqrt{n} = n$. Finding the least cost location within $\sqrt{n} \times \sqrt{n}$ array takes at most n searches for each CLB. The time complexity of placing $n + n_{pi}$ blocks is $O(n(n+n_{pi})) = O(n^2 + n \cdot \sqrt{n}) = O(n^2)$. The total time complexity of *ConeCLBPlace* is $O(n^{1.5}) + O(n^2) = O(n^2)$. □

3.3.6 Iterative improvement of placement

The initial placement produced by our greedy *cone* based heuristic is typically twice as better as random placement but inferior to the solution obtained by simulated annealing based VPR. To improve the quality of the placement by *ConeCLBPlace* further, a low temperature *simulated annealing* is executed on it in the VPR framework. The initial temperature is empirically computed for the benchmark circuits

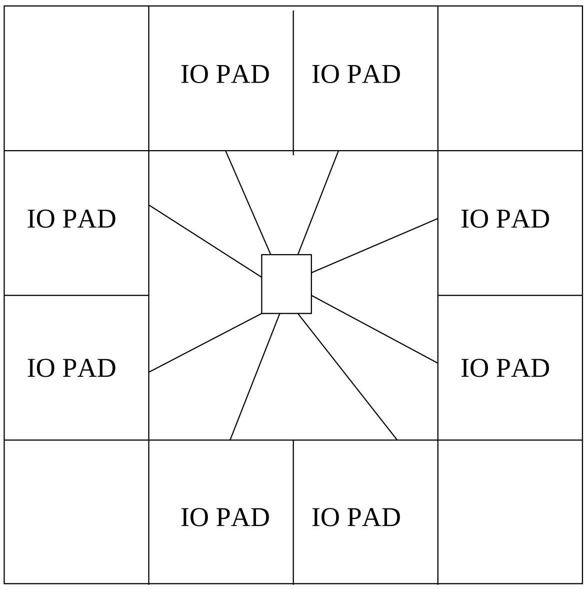


Figure 3.5: Position of padframe in an FPGA.

using appropriate values for input parameters *init_t* and *exit_t* of VPR. The experimental results in Section 3.6 show that the *simulated annealing* converges much faster with the initial placement obtained by *ConeCLBPlace*.

3.4 *ConeIOBPlace*: Cone based IOB placement

The *primary output blocks* were placed at random positions by our previous method *ConeCLBPlace*. The placement of IOBs is also significant to ensure quality placement [Mak 2004, Farbarik 1997, Anderson 2000, Mak 2005]. In this section, a deterministic and fast method of placing the IOBs is proposed. The random placement of IOBs on the chip periphery degrades the placement quality [Khalid 1995]. Coupling of the proposed deterministic IOB placement, *ConeIOBPlace* with standard stochastic block placement methods of VPR, produced placement solution comparable to VPR with a little speedup over VPR. In order to explain the proposed method *ConeIOBPlace*, we define the following terms.

Definition 3.6 (Pad frame:) *A pad frame is the area around the rectangular boundary of a chip, consisting of input and output pads.*

Definition 3.7 (Size of a pad frame:) *The size of a pad frame for an FPGA chip is the number of IOB locations in the pad frame along each side on its periphery.*

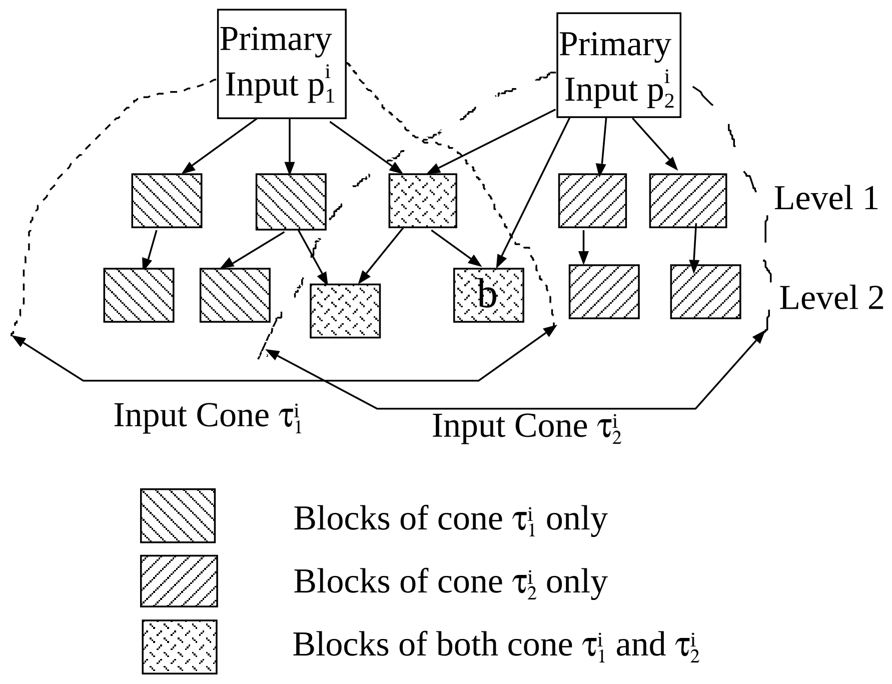

Primary Input p_1^i
Primary Input p_2^i
Level 1
Level 2
b
Input Cone τ_1^i
Input Cone τ_2^i
Blocks of cone τ_1^i only
Blocks of cone τ_2^i only
Blocks of both cone τ_1^i and τ_2^i

Figure 3.6: Two *input cones* with overlap corresponding to two *primary input blocks* p_1^i and p_2^i.

Definition 3.8 (Input cone:) *An input cone of a primary input block $p^i \in P$ is the breadth-first search tree $\tau^i = (V_\tau, E_\tau)$ in D^b, with primary input block p^i as the root.*

A typical pad frame surrounding the periphery of the FPGA chip, is shown in Figure 3.5. Figure 3.6 shows two overlapping *input cones* having common blocks. Now, the placement problem for IOBs is to find an assignment of IOBs to the positions on the *pad frame* such that the total wirelength is minimized after the placement of CLBs. Since the *pad frame* surrounds the 2D array of CLBs almost like a circle, we need to obtain a *circular arrangement* of IOBs such that a pair of IOBs is placed closer to each other if their *cones* have more common blocks than other pairs. Thus, the problem is defined as follows:

Problem 3.2 (Placement problem for IOBs) *Given the netlist of CLBs and IOBs, and the size of the pad frame for the target FPGA array, the IOB placement problem is to determine a circular arrangement of IOBs with a particular separation between adjacent IOBs such that an IOB pair with common CLBs in their respective cones are placed closer to each other than a pair of IOBs with no common CLBs in their respective cones.*

The proposed method *ConeIOBPlace* to solve the placement problem for IOBs, has two phases:

Phase I. Arrangement of p IOBs at p equidistant points on a circle (*circular arrangement*) which maximizes the *overlap* between adjacent *cones* corresponding to the adjacent IOBs.

Phase II. Computation of actual separation between adjacent IOBs in the arrangement obtained in Phase I for the target FPGA *padframe*.

3.4.1 Phase I: Generation of circular arrangement of IOBs

The problem of finding a *circular arrangement* is NP-hard [Galil 1977]. Thus, a heuristic is proposed. First, both the *input* and *output cones* are extracted from the graph D^b. The *input cones* and *output cones* are generated using *breadth-first* traversal on *fanout* blocks, and on *fanin* blocks of the graph D^b respectively. The extracted *cones* represent the level-wise connectivity of logic blocks connected either directly, or through a number of intermediate blocks to the corresponding IOB. The level of a block in a *cone* τ^i or τ^o is defined as one more than the number of intermediate blocks in the path from the block to that IOB. The *cones* for different IOBs may be overlapping, as shown in Figure 3.6. A block in a particular level of a *cone* corresponding to an IOB can be either present in the same or a different level of the *cones* for other IOBs. For example, in Figure 3.6, the logic block b is present in level 2 of cone τ_1^i, and in level 1 of cone τ_2^i. Next, we construct *cone adjacency graphs* in order to obtain a set of *linear arrangements*. Then, the *circular arrangement* is obtained by processing a set of *linear arrangements* in a bottom-up fashion as discussed in the following subsections.

3.4.1.1 Preprocessing: construction of *cone adjacency graph*

In order to obtain the *circular arrangement* of IOBs, a *cone adjacency graph* with vertices corresponding to the IOBs, is constructed first. The *cone adjacency graph* $G^a = (V, E)$ is an undirected and weighted graph, where each vertex $v_i \in V$ represents an IOB p_i, and there exists an undirected edge $(v_i, v_j) \in E$, if there exists at least one common block in the *cones* of p_i and p_j corresponding to vertices v_i and v_j. The weight w_{ij} of an edge $(v_i, v_j) \in E$ reflects the *closeness* of the IOB blocks p_i and p_j in the desired *circular arrangement*. The scheme for assigning weight is discussed next.

3.4.1.2 Preprocessing: computation of edge weights of *cone adjacency graph*

A CLB which is connected to two IOBs and is at a level nearer to both the IOBs, votes for placing the two IOBs closer to each other on the pad frame in order to minimize the total wirelength. Hence, a logic block b, if present in level l_i^b of the *cone* τ_i, and also in level l_j^b of the *cone* τ_j, contributes $w_{ij}(b)$ weight to the edge (v_i, v_j) and is given by

$$w_{ij}(b) = \frac{1}{l_i^b} + \frac{1}{l_j^b}$$

If there are b_{ij}^c common blocks in the *cones* τ_i and τ_j, then,

$$w_{ij} = \sum_{b=1}^{b_{ij}^c} w_{ij}(b)$$

3.4.1.3 The circular arrangement problem (CAP)

The cone adjacency graph G^a models the relative *closeness* of all possible IOB pairs with respect to the number of common blocks and their position in the respective *cones*. The aim is to obtain a *circular arrangement* of the vertices of G^a such that a pair of vertices connected with higher edge weights are adjacent in the arrangement than the vertices connected with smaller edge weights. Thus, the problem of finding a circular arrangement of IOBs is formulated as follows.

Problem 3.3 (Circular Arrangement Problem(CAP)) *Given a weighted undirected cone adjacency graph $G^a = (V, E)$, the problem is to find a circular arrangement $CA(G^a)$ of the vertices $h : V \rightarrow \{1, \ldots, |V|\}$ of G^a on evenly placed points on a circle such that the sum $\sum_{(i,j)\in E} w_{ij} \cdot L(h(i), h(j))$ is minimized, where w_{ij} is the weight of the edge (i, j) and $L(h(i), h(j)) = min\{|h(i) - h(j)|, |V| - |h(i) - h(j)|\}$ is the minimum distance between $h(i)$ and $h(j)$ along the perimeter of the circle.*

Liberatore [Liberatore 2002] has proposed an *approximation algorithm* for computation of the *circular arrangement*. But the algorithm is complex and has high time complexity. Hence, we propose a simple, yet effective heuristic method to obtain the *circular arrangement*. In order to place the vertices with less edge weights farther away from each other in the *circular arrangement*, a *balanced min-cut partitioning* [Kernighan 1970b, Fiduccia 1982] step is applied to the *cone adjacency graph* G^a first. G^a is recursively bi-partitioned up to r levels in order to obtain 2^r subgraphs, where $2^r \leq \sqrt{p}$. The recursive bi-partitioning generates a binary *partition* *tree* with each node corresponding to a subgraph as shown in Figure 3.7(a). Thus, a list of subgraphs $SG_i, i = 1 \cdots 2^r$ of approximately equal size ($\approx 2^r$) is generated at the leaves of the *partition tree*. For each subgraph SG_i at the leaves of the binary *partition tree*, a *linear arrangement* $O_i, i = 1 \cdots 2^r$ is obtained as discussed in Section 3.4.1.4. Next, the *partition tree* is processed bottom-up to generate the *circular arrangement* of IOBs at the root of the *partition tree*.

3.4.1.4 Heuristic for circular arrangement

As mentioned above, the *cone adjacency graph* is partitioned into subgraphs SG_i in order to place loosely connected IOBs farther away from each other. Now, for each of the subgraph SG_i, a *linear arrangement* O_i is obtained. The *linear arrangement* problem is the one, where the vertices are placed on an evenly spaced points on a line rather than a circle. The problem of *linear arrangement* [Garey 1979] also being NP-hard, a simple heuristic has been proposed.

Linear arrangement of vertices of a subgraph: First, a vertex v_h is chosen arbitrarily. Next, in the subgraph SG_i, we find the maximum weighted edge $e = (v_h, v_t)$ with one of the end points as v_h. The vertices v_h and v_t are considered adjacent to each other in the *linear arrangement* in the order $\{v_h, v_t\}$. Let v_h and v_t be the *head* and *tail* of the *linear arrangement* respectively. A single vertex is appended either at the *head* or at the *tail* of this *partial* list of *linear arrangement* during each step of the proposed heuristic. A vertex v_i having maximum weighted edge (v_i, v_h), and a vertex v_j having maximum weighted edge (v_t, v_j), are determined. If v_i and v_j are distinct, we append v_i before the current *head* v_h, and v_j after the *tail* v_t of the linearly arranged list respectively. Then the *head* and *tail* pointers are updated to v_i and v_j respectively. If v_i and v_j happen to be the same vertex, i.e., $v_i = v_j$, v_i is appended before the *head* of the list if $w_{ih} > w_{ti}$, else v_i is appended to the *tail* of the list. This process is repeated till all the vertices of the subgraph have been included in the list. This completes the generation of a *linear arrangement* O_i for the corresponding subgraph SG_i. The method is followed to generate *linear arrangement* for each subgraph SG_i at the leaves of the *partition tree*.

Circular arrangement of vertices of G^a: In order to generate the *circular arrangement* of all the IOBs, the *partition tree* is processed bottom-up as shown in Figure 3.7(c). Let v_i^h and v_i^t be the *head* and *tail* vertices of the *linear arrangement* O_i respectively. Similarly, let v_{i+1}^h and v_{i+1}^t be the *head* and *tail* vertices of the *linear arrangement* O_{i+1} respectively. The *linear arrangement* of two adjacent subgraphs among themselves is obtained by examining the four possible edge weights between the end vertices (v_i^h, v_{i+1}^h), (v_i^h, v_{i+1}^t), (v_i^t, v_{i+1}^h), (v_i^t, v_{i+1}^t) of the two adjacent arrangements O_i and O_{i+1}. These edge weights are termed as the *junction weights* of the two arrangements. The permutation with maximum *junction weight* is chosen as the *linear arrangement* of two adjacent *linear arrangements* O_i and O_{i+1} for the subgraph at their parent node in the *partition tree*. This process continues till it reaches one level below the root. To get the *circular arrangement* O^a at the root for the entire *cone adjacency graph* G^a, an additional *junction weight* due to the vertices at the extreme ends of each of the four possible permutation is also added to the *junction weights* of respective permutation. The permutation with maximum *junction weight* gives the final *circular arrangement* of IOBs.

The *circular arrangement of IOBs* assumes uniform distance between the adjacent IOBs. However, the shape of the IOB pad frame is rectangular rather than circular on the FPGA chip. Also, the *size* and *depth* of each *cone* in terms of number of blocks and levels respectively, is not uniform. This necessitates the IOBs to be placed at some distance apart from each other in order to avoid *congestion* during *routing*. Thus, the distance between adjacent IOBs are computed on the basis of the number of CLBs in each *cone*, the number of CLB overlaps between cones and the level of a block in different *cones*.

3.4.2 Phase II: Computation of separation between adjacent IOB positions

In order to compute the separation between adjacent IOBs, the *average width of a cone*, i.e, the average number of blocks per level of each *cone* is computed. Next, the average number of common blocks in each level of adjacent pair of *cones* in the *circular arrangement* is computed. This gives the *width of overlap* between the *cones* of adjacent IOBs. An estimation of the separation of two adjacent IOBs in the *circular arrangement* is computed as follows. Let W_i^c and W_j^c be the *average width of cones* τ_i and τ_j respectively corresponding to the adjacent IOBs p_i and p_j. Let W_{ij}^o be the *width of overlap* of the pair of *cones* τ_i and τ_j. Then the estimate of the separation d_{ij} between p_i and p_j is given by

$$d_{ij} = W_i^c/2 + W_j^c/2 - W_{ij}^o$$

The exact coordinate positions of the IOBs on the periphery of the FPGA array is computed by scaling the distance d_{ij} to the *pad frame* using the size of the rectangular *pad frame* and the number of IOBs to be placed.

3.4.3 Illustration of IOB placement by *ConeIOBPlace*

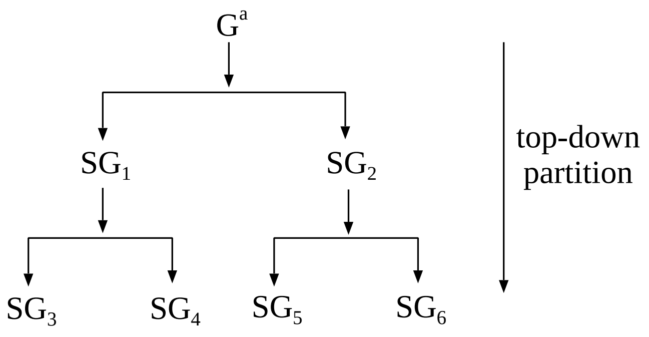
Figure 3.7 illustrates the proposed method with an example. Figure 3.7(a) shows the subgraphs SG_is generated by top-down recursive balanced bi-partitioning of the *cone adjacency graph* G^a.

Figure 3.7(b) shows a subgraph SG_i with five vertices, for which the *linear arrangement* is obtained. We start with an arbitrary vertex, say v_1. Vertex v_5 is connected to v_1 with maximum weighted edge. Hence we append v_5 to the arrangement $\{v_1,\ v_5\}$. Among the remaining vertices adjacent to v_1 and v_5, v_2 is a vertex adjacent to both v_1 and v_5 with weight 10 and 9 respectively. As $w_{12} > w_{25}$, we choose w_{12} as the max-weight edge and v_2 is appended to the left of v_1 generating the arrangement $\{v_2,\ v_1,\ v_5\}$. Proceeding thus, the *linear arrangement* of the vertices of subgraph SG_i is $O_i = \{v_3,\ v_4,\ v_2,\ v_1,\ v_5\}$. Figure 3.7(c) shows the order of processing the individual *linear arrangement* of each sungraph in order to obtain the *circular arrangement* of the vertices in the entire *cone adjacency graph* G^a. Let O_3, O_4, O_5, O_6 be the *linear arrangements* of the subgraphs SG_3, SG_4, SG_5, SG_6 respectively. Then the *linear arrangement* O_1 for the parent graph SG_1 of the subgraphs SG_3 and SG_4 are obtained by calculating the *junction weights* for the four possible permutations of O_3 and O_4. Similarly, the *linear arrangement* O_2 is computed from O_5 and O_6. Finally, the *circular arrangement* of IOBs O^a for the entire *cone adjacency graph* G^a is obtained from O_1 and O_2 with the additional *junction weights.*

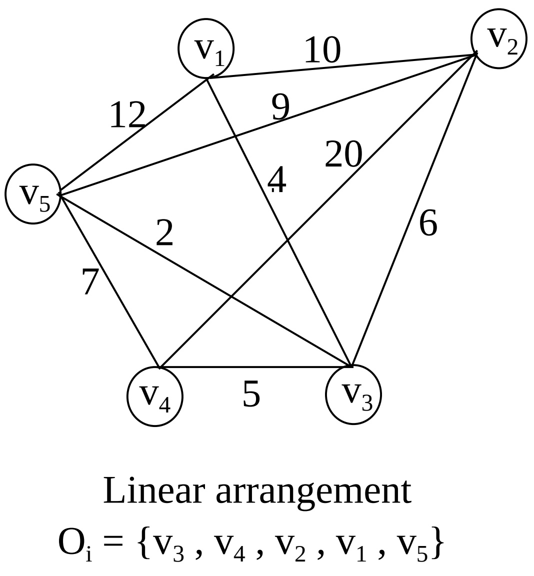
3.4.4 Time complexity of *ConeIOBPlace*

Theorem 3.2 *The time complexity of ConeIOBPlace is $O(n^2)$, where n is the number of CLBs.*

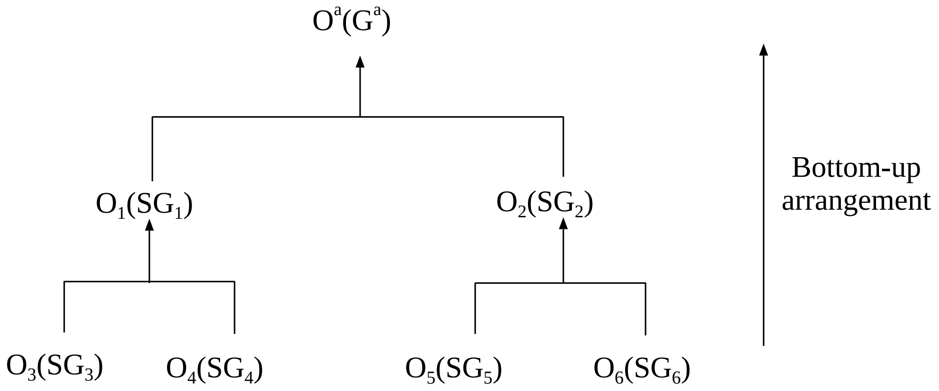
Proof: The two phases of *ConeIOBPlace* is given in Algorithm 3.2 The time complexity of generating *input* and *output cones* by BFS traversal is $O(pn)$, where p and n are the total number of IOBs and the number of CLBs respectively in the circuit. Since $p \approx O(\sqrt{n})$, the *cone* generation takes $O(n^{1.5})$ time. The time complexity of construction of weighted *cone adjacency graph* G^a is $O(p^2 n) = O(n^2)$, since edge weights are computed for all possible p^2 edges of the graph. The partitioning of G^a takes $O(n)$ time [Fiduccia 1982] as the number of edges in the graph is $O(n)$. The time complexity of finding the *linear arrangement* for all the subgraphs is $O(p^2) = O(n)$ and the time taken to generate the final *circular arrangement* hierarchically is $O(n)$. Thus, the time complexity of the first phase is $O(n^2)$.



(a)



(b)



(c)

Figure 3.7: Example: (a) Generation of subgraphs from the *cone adjacency graph*; (b) Subgraph SG_i for which *linear arrangement* O_i of vertices is obtained; (c) Generation of *circular arrangement* of IOBs from *linear arrangement* of subgraphs.

Algorithm 3.2: ConeIOBPlace

```
   Input  : Block netlist graph $D^b$ corresponding to the netlist
   Output: Placement of CLBs and IOBs on the two dimensional array
1  Phase I: Generation of circular arrangement of IOBs;
2  for each IOB do
3      Extract the input and output cone by BFS traversal of $D^b$ starting from
       each IOB ;
4  Generate cone adjacency graph $G^a$ with vertices corresponding to IOBs and
   edge weights representing the closeness between the IOBs corresponding to
   the end vertices of an edge;
5  Apply recursive min-cut bi-partition on $G^a$ generating a partition tree with
   $2^r \leq \sqrt{p}$ almost equal sized subgraphs $SG_i$ at the leaf;
6  for each subgraph $SG_i$ do
7      Compute linear arrangement $O_i$;
8  Process $O_i$s bottom-up in the partition tree to generate the circular
   arrangement at root based on juntion weight;
9  Phase II: Computation of separation between adjacent IOBs;
10 for each cone $\tau_i$ do
11     Compute the average width of cone $W_i^c$;
12 for each pair of adjacent IOBs $p_i$ and $p_j$ do
13     Compute $W_{ij}^o$, the average width of overlap of cones $\tau_i$ and $\tau_j$;
14     Estimate the separation $d_{ij}$ between IOBs $p_i$ and $p_j$ as
       $d_{ij} = W_i^c/2 + W_j^c/2 - W_{ij}^o$;
```

In the second phase, computation of the *average width* of p *cones* is done in $O(n)$ time, n being the number of CLBs in the netlist graph. The computation of the *average width of overlap* between adjacent *cones* takes $O(n)$ time, and the computation of separation between adjacent IOBs take $O(n)$ time. Hence, the total time complexity of the second phase is $O(n)$. Thus, the total time complexity of the method is $O(n^2)$. $\square$

3.4.5 Placement of CLBs

To complete the placement of all blocks and observe the effect of an IOB placement, we employ the simulated annealing based VPR to place the CLBs on to the FPGA array. The positions of IOBs obtained by *ConeIOBPlace* is given as an input to VPR. Then, VPR is executed only for placing the CLBs without changing the positions of IOBs. This completes the flow of *ConeIOBPlace*. The performance of the proposed method on a set of benchmark circuits is reported in Section 3.6.

Table 3.2: Characteristics of MCNC FPGA placement benchmark circuits

1	2	3	4	5
Ckt	# CLBs (n)	Number of inputs	Number of outputs	Size of 2D grid
ex5p	1064	8	63	33×33
misex3	1397	14	14	38×38
diffeq	1497	64	39	39×39
alu4	1522	14	8	40×40
des	1591	256	245	63×63
bigkey	1707	229	197	54×54
seq	1750	41	35	42×42
apex2	1878	38	3	44×44
s298	1931	4	6	44×44
spla	3690	16	46	61×61
pdc	4575	16	40	68×68
ex1010	4598	10	10	68×68

3.5 *ConePlace*: Cone based IOB and CLB placement

In our last method *ConePlace*, the ideas of *ConeCLBPlace* and *ConeIOBPlace* are combined in order to evaluate the overall performance of the proposed methods. Unlike random positioning of *primary output blocks*, all the IOBs are placed according to the *circular arrangement* obtained by *ConeIOBPlace*. Next, for each *output cone*, we place the CLBs by *ConeCLBPlace*. Finally, in order to improve the quality of the solution further, a low temperature simulated annealing is executed on the placement obtained.

3.6 Experimental results

In this section, the results obtained by the proposed methods *ConeCLBPlace*, *ConeIOBPlace* and *ConePlace* on a set of *technology-mapped* benchmarks have been reported. We use the most popular FPGA benchmarks from Microelectronics Center of North Carolina (MCNC) for all our experiments. We have implemented the proposed methods in C on 1.2GHz SunBlade 2000 workstation with SunOS Release 5.8. Table 3.2 shows the characteristics of the benchmark circuits in terms of number of CLBs, *primary inputs* and *output blocks*. The minimum square array required to place all the CLBs and IOBs for each circuit is given in the last column. Next, the performances of the proposed methods have been reported in terms of the *BB*

Table 3.3: Comparison of *wirelength driven BB cost*: *ConeCLBPlace* vs. VPR

1	2	3	4	5	6	7	8	9
	BB cost			Quality ($\frac{Ours}{VPR}$)		Number of SA moves		
Ckt	Initial (Ours)	Final (Ours)	Final (VPR)	w.r.t Initial	w.r.t Final	Ours ($\times 10^6$)	VPR ($\times 10^6$)	Speedup ($\frac{VPR}{Ours}$)
ex5p	291	165	162	1.79	1.02	3.7	13.6	3.7
apex4	316	181	180	1.75	1.01	5.8	16.4	2.8
alu4	361	194	192	1.88	1.01	6.0	22.1	3.7
seq	466	251	248	1.87	1.01	7.1	27.4	3.8
apex2	568	271	268	2.11	1.01	10.0	29.1	2.9
spla	1395	600	608	2.29	0.99	19.8	75.2	3.8
pdc	1725	903	871	1.98	1.04	37.0	97.2	2.6
ex1010	2252	659	654	3.44	1.01	28.4	101.5	3.5
Avg:				2.14	1.01			3.35

cost (computed as in Equation 3.1), *speedup* and *critical path delay* obtained after routing using VPR router. The *critical path delay* is reported in terms of 10^{-8} seconds and the CPU time is reported in *seconds*. In the Tables, the column headed *Initial* reports the *initial cost* obtained by the proposed methods *ConeCLBPlace* and *ConePlace* without the execution of low temperature simulated annealing. The columns headed *Final* reports the *final cost* after execution of low temperature simulated annealing in the *wirelength driven* VPR framework for the proposed methods *ConeCLBPlace* and *ConeIOBPlace*. For the method *ConePlace*, the final placement results of both *wirelength driven* and *timing driven* modes are given.

3.6.1 Placement obtained by *ConeCLBPlace*

Table 3.3 shows the comparison of *bounding box cost* (BB cost) as obtained by *ConeCLBPlace* with that of VPR. In both the cases, the proposed method and VPR are executed in *wirelength driven* mode with *-place_ algorithm* option as bounding_box. The second, third and fourth columns report the *BB cost* obtained by the *initial placement* of *ConeCLBPlace*, the *final cost* obtained after the execution of low temperature simulated annealing on *initial placement* of *ConeCLBPlace* and the *final cost* obtained by VPR respectively. The next two columns compare the *quality* of the *initial* and *final placement* respectively when compared to VPR. It shows that, on an average the *BB cost* of *initial placement* obtained by *ConeCLBPlace* is 2.14× of the placement obtained by VPR, although for smaller circuits, it is less than 2×.

Table 3.4: Comparison of *critical path delay*: *ConeCLBPlace* vs. VPR

1	2	3	4
	Critical path delay (10^{-8} sec.)		
Ckt	Ours	VPR	$\frac{Ours}{VPR}$
ex5p	15.9	11.2	1.41
apex4	20.6	12.8	1.61
alu4	19.1	12.1	1.58
seq	16.9	12.7	1.33
apex2	16.7	12.1	1.38
spla	23.3	17.1	1.36
pdc	39.5	21.9	1.80
ex1010	38.4	23.0	1.67
Average:			1.5

On execution of low temperature simulated annealing the quality of placement is as good as the placement of VPR (1.01× of VPR) as shown column 6. The *number of simulated annealing moves* required to obtain the *final placement* is reported in columns 8 and 9. Column 8 shows the *number of moves* required in execution of a low temperature annealing which is much less than the moves required in execution of VPR (column 9), thereby converging much faster than VPR and providing faster compilation time with almost equal quality of solution. The *speedup* over VPR is reported in the last column. On an average, the speedup is 3.3×. As the *initial placement* takes just a few seconds, the *speedup* is significant even if a low temperature simulated annealing is executed on the initial placement.

In order to validate the quality of the placement produced, the VPR router was executed on the *final placement* produced by our method. The results of *critical path delay* obtained by routing, is compared with that of routing a placement obtained by VPR in Table 3.4. The *critical path delay* reported here is of the order of 10^{-8} seconds. It is observed that although the *critical path delay* of the placements obtained by our method is 1.5× of VPR on the average, the placement was routable. This shows the suitability of the method for fast reconfiguration at the cost of a small compromise in the quality of the solution.

3.6.2 Placement obtained by *ConeIOBPlace*

The effect of IOB placement by *ConeIOBPlace* is reported next. Table 3.5 shows the *BB cost* obtained by pre-placing the IOBs by *ConeIOBPlace* against the random IOB placement of VPR. In both the cases, CLBs are placed by simulated annealing

Table 3.5: Comparison of *wirelength driven BB cost*: *ConeIOBPlace* vs. VPR

1	2	3	4	5	6	7
	BB cost			Number of SA moves ($\times 10^6$)		
Circuit	Ours	VPR	$\frac{Ours}{VPR}$	Ours	VPR	$\frac{Ours}{VPR}$
ex5p	158	162	0.98	11.4	13.6	0.83
apex4	182	186	0.98	15.2	16.4	0.92
alu4	190	192	0.99	21.5	22.1	0.92
des	360	361	1.00	34.4	36.9	0.93
bigkey	270	268	1.01	42.5	44.7	0.95
apex2	265	268	0.99	26.8	29.0	0.92
pdc	871	871	1.00	94.4	97.2	0.97
Average:			0.99			0.92

of VPR in the *wirelength driven* mode using the option *-place_ algorithm* as *bounding_ box*. The positions of the IOBs obtained by *ConeIOBPlace* on the periphery of the array is given as an input to VPR using *-fix_ pins* option. The *BB cost* obtained by pre-placing IOBs by the proposed method is reported in the second column. The third column shows the *BB cost* obtained by VPR. The fourth column shows that the quality of the placement is 0.99× of VPR implying a small improvement in the *BB cost* when IOBs are placed by *ConeIOBPlace*. As in the earlier method, we report the *number of simulated annealing moves* in the fifth and the sixth columns respectively for the proposed method and VPR. The *speedup* is given in the last column. It is observed that on an average the speedup is 0.92× of VPR implying a small gain in the execution time.

In order to compare the *critical path delay* obtained by *ConeIOBPlace* and VPR, we routed all the placements obtained using the default router of VPR. The results are reported in Table 3.6. The second column reports the *critical path delay* of each benchmark circuit obtained after routing the placement obtained by *ConeIOBPlace* followed by CLB placement using simulated annealing of VPR. The *critical path delay* obtained by VPR is reported in the third column. The last column compares the quality of *critical path delay*. It shows that, on an average, the *critical path delay* obtained is 0.99× of VPR implying a positive gain.

3.6.3 Placement obtained by *ConePlace*

Observing the positive performances of the proposed heuristics *ConeCLBPlace* and *ConeIOBPlace* in terms of *speedup* and *quality* independently, we observed the effect of the combined approach *ConePlace*. Here, we execute the low temperature sim-

Table 3.6: Comparison of *critical path delay*: *ConeIOBPlace* vs. VPR

1	2	3	4
Circuit	Ours $\times 10^{-8}s$	VPR $\times 10^{-8}s$	$\frac{Ours}{VPR}$
ex5p	6.5	6.7	0.97
apex4	7.0	7.2	0.97
alu4	8.4	8.6	0.98
des	14.7	14.8	0.99
bigkey	10.3	10.2	1.01
apex2	8.9	8.9	1.00
pdc	15.1	15.1	1.00
Average			0.99

ulated annealing in VPR framework with both *wirelength* and *timing driven* mode setting -*place_ algorithm* option of VPR to be *bounding_ box* and *net_ timing_ driven* respectively and compare the results of both the modes. While there was no significant improvement in *BB cost* by applying *ConePlace* on the set of benchmark circuits, the *critical path delay* obtained after routing was better in this combined approach.

Table 3.7 shows the *BB cost* obtained by the *initial placement* of *ConePlace* and the *final placement* after execution of low temperature simulated annealing in both modes. The *BB cost* obtained by VPR in both modes are also reported under the respective modes. It is observed that the *initial cost* obtained by *ConePlace* is much higher than the one obtained by *ConeCLBPlace* and hence we do not compare it with the *final cost* of VPR. The fifth and eighth columns show that, on an average, the quality of the *final placement* by the proposed method *ConePlace* are 1.15× and 1.11× of VPR in *wirelength* and *timing driven* mode respectively. The quality of the placement is better in *timing driven* mode than the *wirelength driven* one.

Next, we compare the total *CPU time* required in both *wirelength driven* and *timing driven* mode by the combined method *ConePlace*, which includes the deterministic heuristic for IOB placement *ConeIOBPlace*, CLB placement *ConeCLBPlace* and also the iterative improvement by low temperature simulated annealing. Table 3.8 shows the *CPU time* taken in seconds by the proposed method and VPR. The fourth and seventh columns show the *speedup* of our method for each circuit. On the average, the *speedup* is 2.01× and 2.09× of VPR in *wirelength* and *timing driven* mode respectively. Comparing with *ConeCLBPlace*, the speedup was more in *ConeCLBPlace* than the combined approach *ConePlace*.

Table 3.7: Comparison of *BB cost*: *ConePlace* vs. VPR

1	2	3	4	5	6	7	8
		Wirelength driven			Timing driven		
Ckt	Initial (Ours)	Final (Ours)	Final (VPR)	Final $\frac{Ours}{VPR}$	Final (Ours)	Final (VPR)	Final $\frac{Ours}{VPR}$
ex5p	310	173	162	1.07	184	180	1.02
misex3	402	191	189	1.01	200	200	1.00
diffeq	371	177	147	1.20	187	158	1.18
alu4	451	194	192	1.01	201	199	1.01
des	830	383	228	1.68	396	258	1.53
bigkey	584	306	186	1.65	313	209	1.50
seq	591	265	248	1.07	276	263	1.05
apex2	589	281	268	1.05	293	281	1.04
s298	463	205	205	1.00	228	229	1.00
spla	1713	637	608	1.05	667	633	1.05
pdc	2298	898	871	1.03	967	945	1.02
ex1010	2642	657	654	1.00	679	675	1.01
Average:				1.15			1.11

Table 3.8: Comparison of *speedup*: *ConePlace* vs. VPR

1	2	3	4	5	6	7
	Wirelength driven			Timing driven		
Ckt	Ours	VPR	$\frac{VPR}{Ours}$	Ours	VPR	$\frac{VPR}{Ours}$
ex5p	42	98	2.33	63	169	2.68
misex3	60	136	2.27	78	218	2.79
diffeq	54	161	2.98	100	247	2.47
alu4	79	150	1.90	90	242	2.69
des	122	224	1.84	169	312	1.85
bigkey	87	200	2.30	157	331	2.11
seq	103	198	1.92	146	312	2.14
apex2	100	215	2.15	122	339	2.78
s298	84	201	2.39	210	321	1.53
spla	397	600	1.51	685	961	1.40
pdc	575	770	1.34	941	1280	1.36
ex1010	597	775	1.30	839	1161	1.38
Average:			2.01			2.09

Finally we compare the *critical path delay* obtained by *ConePlace* with that of VPR. Table 3.9 shows the *critical path delay* obtained by *ConePlace* and VPR in both *wirelength* and *timing driven* mode respectively. The quality of the *critical*

Table 3.9: Comparison of *critical path delay*: *ConePlace* vs. VPR

1	2	3	4	5	6	7
	Wirelength driven			Timing driven		
Ckt	Ours $\times 10^{-8}s$	VPR $\times 10^{-8}s$	$\frac{Ours}{VPR}$	Ours $\times 10^{-8}s$	VPR $\times 10^{-8}s$	$\frac{Ours}{VPR}$
ex5p	12.4	11.2	1.11	8.95	8.11	1.10
misex3	11.8	11.2	1.05	10.1	10.5	0.96
diffeq	9.84	9.50	1.04	6.24	6.58	0.95
alu4	13.5	12.1	1.12	11.7	8.53	1.37
des	14.0	12.1	1.16	13.3	8.97	1.48
bigkey	9.59	10.5	0.91	6.85	6.26	1.09
seq	13.0	12.7	1.02	11.1	9.45	1.17
apex2	14.3	12.1	1.18	12.1	10.6	1.14
s298	20.8	19.9	1.05	13.6	16.2	0.84
spla	23.2	17.1	1.36	17.5	17.0	1.03
pdc	26.5	21.9	1.21	28.0	15.6	1.79
ex1010	19.2	23.0	0.83	22.7	17.5	1.30
Average:			1.08			1.18

path delay compared to VPR in both modes are shown in the fourth and seventh columns. It shows that *ConePlace* produces placement with *critical path delay* comparable to VPR, as it is 1.08× of VPR in *wirelength driven mode*, on an average. In case of *timing driven* mode, the *critical path delay* is 1.18× of VPR. However, the *critical path delay* is much better than the one produced by *ConeCLBPlace*, where the *critical path delay* is 1.5× of VPR. This shows that *ConePlace*, the combined approach provides significant improvement in the quality of the placement even if the *BB cost* did not show promising results.

3.7 Conclusion

In this chapter, we present three variants of constructive initial placement method for placement of *netlist* on FPGA to *accelerate* the final placement phase of a simulated annealing based approach with the objective of minimizing the standard *BB cost*. We formulate the placement of IOBs and CLBs as separate problems and propose fast deterministic heuristics to obtain placement for each of them. We observe the results produced by these two independent methods *ConeCLBPlace* and *ConeIOBPlace*. Finally, we combine both the approaches in *ConePlace* by first placing the IOBs by *ConeIOBPlace* and then the CLBs with *ConeCLBPlace*. The results produced by them are compared with the state-of-the-art tool VPR. The results show significant speedup and even improvement in quality of the solution for a few circuits. This emphasizes the suitability of these methods for fast compilation of smaller circuits on small islands of CLBs on FPGAs. The methods proposed in this chapter are greedy and hence falls into local optima. In order to overcome the limitations of the greedy heuristics, partitioning based approaches were tried out next.

Chapter 4

Top-Down Deterministic FPGA Placement

4.1 Introduction

Contents

In the previous chapter, we presented a new bottom-up, fast, deterministic greedy heuristic to place the netlist of CLBs, observed its performance in terms of speedup and quality and compared the results with the state-of-the-art popular tool VPR. In this chapter, we explore the top-down partitioning based approach for FPGA placement. The partitions are made either *Coarse grain* with more than one CLB in each partition, or *Fine grain* with exactly one CLB per partition. The effect of partitioning the netlist of CLBs on placement, both *Coarse* and *Fine grain*, is evaluated here.

The rest of this chapter is organized as follows. Section 4.2 discusses the overview of our approach. Section 4.3 has the details of our *Coarse grain* partitioning based placement method. The *Fine grain* partitioning based placement method is explained in Section 4.4. Experimental results are reported in Section 4.5, and the concluding remarks appear in Section 4.6.

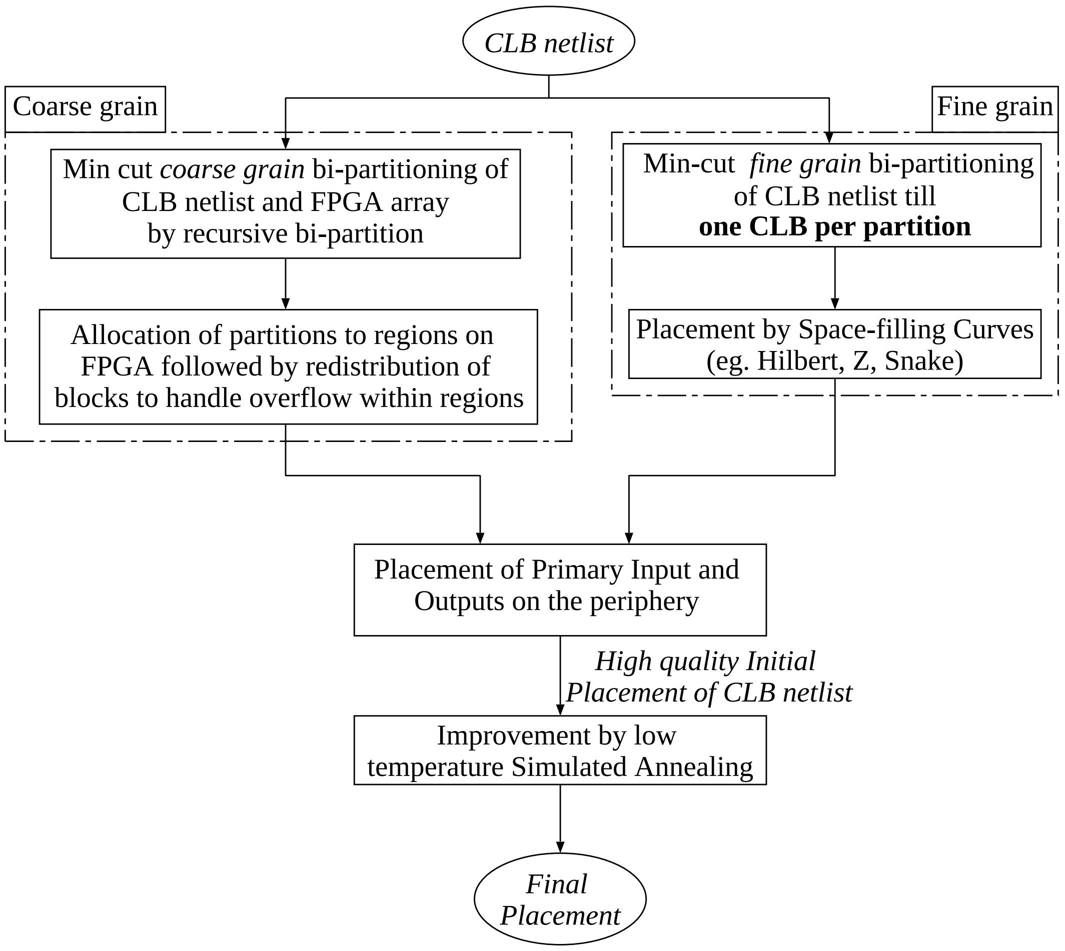

CLB netlist
Coarse grain
Fine grain
Min cut coarse grain bi-partitioning of CLB netlist and FPGA array by recursive bi-partition
Allocation of partitions to regions on FPGA followed by redistribution of blocks to handle overflow within regions
Min-cut fine grain bi-partitioning of CLB netlist till one CLB per partition
Placement by Space-filling Curves (eg. Hilbert, Z, Snake)
Placement of Primary Input and Outputs on the periphery
High quality Initial Placement of CLB netlist
Improvement by low temperature Simulated Annealing
Final Placement

Figure 4.1: Flow of our placement method: *Coarse grain* and *Fine grain*.

4.2 Overview

As in the preceding chapter, the aim of the partitioning based placement is to minimize the total HPWL cost of the placement quickly. We explore two variants of the partitioning based approaches. Our first technique, called the *Coarse grain* method comprises the following steps as shown on the left of Figure 4.1: (i) top-down min-cut partitioning of netlist of CLBs followed by partitioning of FPGA array into z nearly equal sub-arrays (*regions*) of specified *granularity* (i.e., the number of CLBs in a partition) (ii) mapping the z *netlist partitions* to the z equal-sized disjoint square-shaped *sub-arrays* or *regions* on an FPGA array, (iii) global reallocation of logic blocks from the *regions* corresponding to overloaded partitions to neighboring under-utilized *regions*, based on a *maximum flow* formulation to produce a valid placement, and finally (iv) obtaining the final placement solution by an appropriate cooling schedule of *simulated annealing* (SA). This method is termed as *Coarse grain* because each partition produced by partitioning has more than one logic block. Although this partitioning heuristic is very fast, it may generate partitions with more CLBs than the given *granularity*. This requires reallocation of logic blocks in certain *regions*. In order to circumvent the step of reallocation, a second method called *Fine grain*, is also proposed.

The *Fine grain* method consists of the following steps, as shown on the right of Figure 4.1: (i) transformation of the *netlist* of n CLBs to a one dimensional arrangement of CLBs, as opposed to partitions with more than one CLB in the *Coarse grain* method, using top-down bi-partitioning techniques, (ii) mapping of this one dimensional arrangement directly on to the $\sqrt{n} \times \sqrt{n}$ 2D grid using a recursive *space filling curve* [Sagan 1994] producing an *initial placement*, and finally, (iii) determination of the final placement by an appropriate cooling schedule of *simulated annealing* (SA). The technique focuses on generating a one dimensional placement which minimizes the total wirelength. We use min-cut bi-partitioning heuristic to generate the one dimensional (1D) arrangement. Having observed the efficacy of these methods in the experimental results (See Section 4.5), we explore the theoretical basis of this technique later on in Chapter 5. Next, we need to place the 1D arrangement of CLBs to the 2D FPGA array such that the wirelength remains minimized. The intuition behind using *space filling curves* for this purpose, is that on one hand the target location for each CLB in the 1D arrangement is computable in constant time and on the other, it retains the locality properties of the one dimensional arrangement.

Partitioning based placement traditionally has a recursive top-down phase followed by bottom-up construction. The novelty in our proposed methods lies in the fact that, instead of bottom-up construction, we take advantage of the arrangement of CLBs due to bi-partition and then apply the technique of *space filling curve* to obtain a high quality placement very fast.

4.3 Coarse grain placement

Typically, in recursive min-cut bi-partitioning based placement, the netlist and the FPGA array are partitioned independently keeping a correspondence between the partitioned netlist and the partitioned FPGA *sub-arrays*. The FPGA array can be bi-partitioned either horizontally or vertically, which is often chosen arbitrarily or based on some objective criteria such as the *terminal alignment* objective in [Maidee 2003, Maidee 2005]. This incurs additional timing overhead. In order to achieve faster mapping of *netlist partitions* to the FPGA *sub-array* or *region* preserving the relative closeness of two *netlist partitions*, the partitions are allocated to the *regions* in a *snake* like fashion.

4.3.1 Partitioning of CLB netlist

Unlike *block netlist graph* representation of a netlist as in Chapter 3, the netlist of CLBs is modeled as a *hypergraph* here. A *net* originates from a *source* CLB and sinks to the set of CLBs in the *fanout* of the *source*. In the previous chapter, a netlist of CLBs has been modeled as a *block netlist graph*, where there is an edge corresponding to every source-sink CLB pair in a *net*. However, the netlist of CLBs can be better represented as a *hypergraph* as follows.

Definition 4.1 (Netlist hypergraph:) *A netlist hypergraph $H(V, E)$ represents the technology-mapped netlist of CLBs. It has vertices $v \in V$ corresponding to each logic block or CLB that has to be assigned a physical location on the FPGA array. Each hyperedge $e \in E$ represents a net in the netlist corresponding to a subset of vertices in V that constitutes the net.*

The *netlist hypergraph H* is bi-partitioned recursively to obtain balanced min-cut partitions in order to group heavily connected CLBs. By employing a balanced min-cut bi-partition, we obtain two *netlist partitions* having almost the same number of CLBs. The CLBs in each partition are heavily connected as the partitioning is with respect to the minimum cut. We bi-partition the hypergraph recursively till the required *granularity* of *netlist partition* is achieved. The recursive partitioning process generates a *partition tree*, where the root corresponds to all the nodes in the hypergraph and the leaves correspond to the *netlist partitions* of given *granularity*. Without loss of generality, the order of the two partitions in the *partition tree* can be swapped.

We use the state-of-the-art hypergraph partitioner *hMetis* [hMetis , Karypis 1999a] to generate the *netlist partitions* of the given netlist hypergraph H. It first reduces the size of the hypergraph by collapsing vertices and edges in the coarsening phase, then partitions the reduced graph in the initial partitioning phase, and finally un-coarsens it to construct a bi-partition for the original graph in the un-coarsening and refinement phase.

The FPGA array is also partitioned into a number of square shaped *sub-arrays* or *regions* of same *granularity* as the *netlist partition.* In the *Coarse grain* method, without loss of generality, we choose the size of each *region* to be an array of 2×2 on the FPGA. So, each *region* has at most 4 target CLB locations thereby having a *granularity* of 4. This is termed as *region capacity.* The number of *netlist partitions* or *regions* is calculated as follows. The FPGA chip is represented as a $\sqrt{n} \times \sqrt{n}$ array of CLBs, where n is the number of CLBs. The number of 2×2 *regions* generated is $\frac{\sqrt{n}}{2}.\frac{\sqrt{n}}{2} = \frac{n}{4}$. We partition the given netlist hypergraph into z $(= \frac{n}{4})$ *netlist partitions*, so that these *netlist partitions* may be mapped on to $\frac{n}{4}$ *regions* on the chip. This implies that each *netlist partition* has at most 4 logic blocks and each *region* on the FPGA chip has 4 CLB locations within it. However, *hMetis* is based on min-cut, which may often generate many partitions with more than 4 CLBs in it. This causes overflow in some of the regions to which the partitions are mapped. The overflow is resolved in the last step of *reallocation.* Figure 4.2(a) shows the 16 partitions obtained at the leaves of a *partition tree* by min-cut recursive balanced bi-partitioning of a hypothetical circuit.

4.3.2 Allocation of *netlist partitions* to *regions*

The *partition tree* generated has *netlist partitions* at the leaves of the tree. By the process of recursive min-cut bi-partition, the *netlist partitions* of the left sub-tree of root are tightly connected than they are to the *netlist partitions* of the right sub-tree of the root. This implies that there is a *relative closeness* among the left and right children of the *partition tree* and they should be placed next to each other on the FPGA chip in order to minimize the total wirelength. It was observed that, if the *netlist partitions* at the leaves of the *partition tree* are chosen in left to right order of their position in the *partition tree*, and are assigned to the *regions* on the FPGA array in a *snake* like fashion, the relative positions of the *netlist partitions* are preserved. Figure 4.2(b) shows the allocation of the 16 partitions to a 4×4 array of *regions* on the FPGA chip following a simple order commonly known as the *snake.* Each square in Figure 4.2(b) corresponds to a *region* of size 2×2 array of CLBs.

A *snake curve* is a continuous traversal of the euclidean space in a specific man-

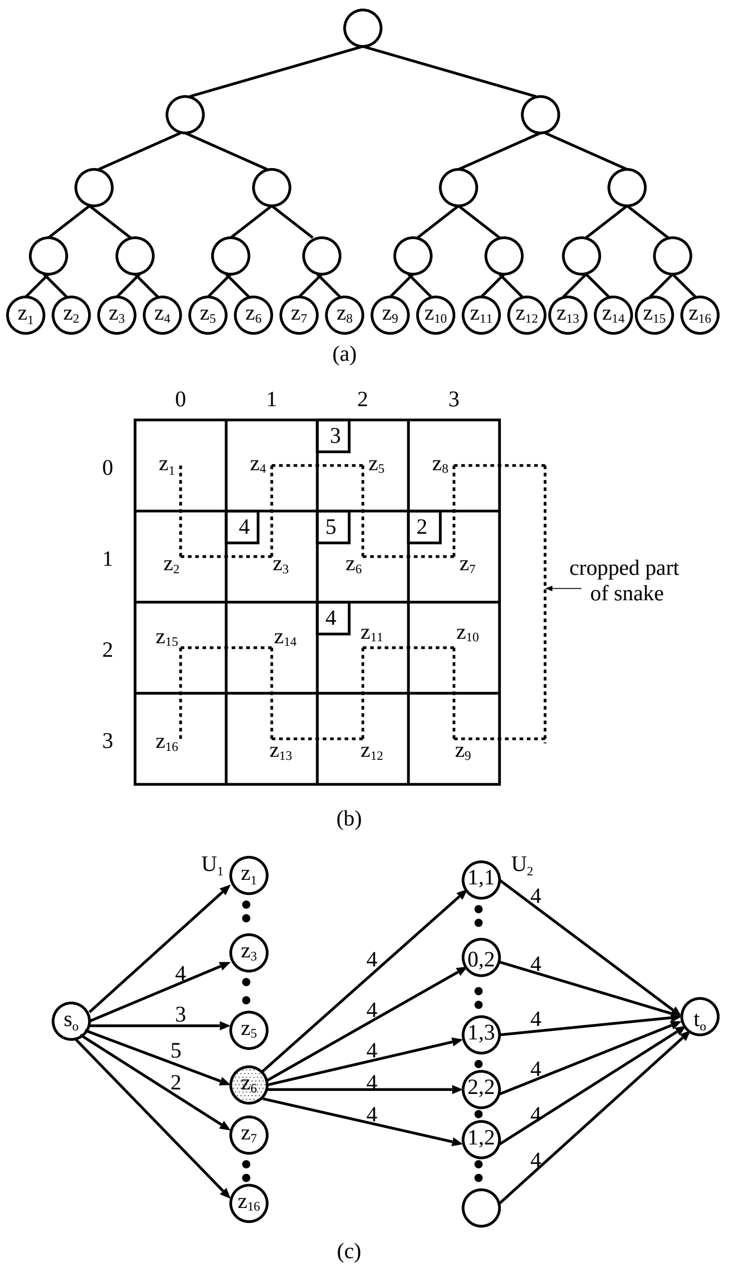


Figure 4.2: Steps of *Coarse grain* method: (a) balanced *partition tree* with *netlist partitions* at the leaves (b) assignment of *netlist partitions* to *regions* on an FPGA in a *snake* like fashion; each square corresponds to a *region* and is labeled by the *netlist partition* z_i assigned to it; the number of CLBs in each of the *netlist partitions* z_3, z_5, z_7, z_{12} and z_6, appears at the top left corner of the corresponding *region* (c) flow network for reallocation of CLBs from the overloaded *regions* to neighboring ones.

ner. A snake curve of height l partitions the 2D grid $a \times a$ into *horizontal stripes* of height l, if l divides a [Asano 1997]. Each of these stripes is covered by a snake-like curve as shown in Figure 4.2(b). Here, we use a snake curve of height 2, which implies it traverses the 2D array 2 units vertically, then two units horizontally and continues till it covers the entire 2D array. The snake curve of height 2 requires odd number of columns to be continuous in the 2D array. Thus, for arrays with even number of columns, an extra column is assumed for generating the curve, although no *netlist partition* is allocated to this column. The next *netlist partition* is allocated to that *region* where the snake curve re-enters the boundary of the given 2D array. The order of allocating a *netlist partition* to a *region* is shown in Figure 4.2(b) using the dotted lines, and the partition numbers are shown next to the *snake* curve in each *region*. Since the array has even number of columns, the cropped part of the snake curve is shown outside the 4×4 array in Figure 4.2(b).

4.3.3 Reallocation in overloaded regions

Ideally, had the number of CLBs in each partition not exceeded the *region* capacity of the region assigned to it, each logic block could be placed within its own *region*. However, several of the $\frac{n}{4}$ *netlist partitions* of the netlist hypergraph, may have more than 4 CLBs, whereas, as per our chosen *granularity* each *region* can accommodate only 4 CLBs. Thus, we need an additional step to reallocate them from *overloaded region* to its neighboring ones such that there is no *overloaded region* on the FPGA array. In order to achieve this, we formulate a *max-flow problem* as follows.

Let $G^o = \{\{s^o, U_1 \cup U_2, t^o\}, A\}$ $(U_1 \cap U_2 = \phi)$ be a bipartite network, where U_1 represents the partitions or the leaf nodes of the *partition tree*; U_2, the candidate *regions* on the FPGA array denoted by the array index (i, j), and s^o and t^o are the special nodes designated as the source and sink respectively. There are three types of arcs in A:

1) (s^o, u_i) $\forall u_i \in U_1$, with its capacity as the number of logic blocks in the partition corresponding to node u_i;

2) (u_i, u_j) where $u_j \in U_2$ is either the *region* assigned to the partition $u_i \in U_1$, or any of the four *regions* which is either a horizontal or a vertical neighbor of u_j. Thus, the *outdegree* of each node $u_i \in U_1$ is less than or equal to 5, and the capacity of arc (u_i, u_j) is the *region capacity* of u_j, namely 4.

3) (u_j, t^o) $\forall u_j \in U_2$, having capacity equal to that of the *region* $u_j \in U_2$, i.e., 4.

The *max-flow* formulation is illustrated in Figures 4.2(b) and 4.2(c). The *netlist partitions* z_6, z_3, z_5, z_7, z_{11} are assigned to the regions $(1, 2)$, and its 4 neighboring regions $(1, 1), (0, 2), (1, 3), (2, 2)$, respectively as shown in Figure 4.2(b). The number of CLBs in each of this *netlist partitions* is shown in a smaller square within the respective *regions*. The flow network corresponding to the *netlist partition* z_6 is shown in Figure 4.2(c) along with the arc capacities. There is an arc from s^o to the node z_6 with arc capacity 5, which is the number of CLBs in *netlist partition* z_6. There are five arcs from node z_6; one to the node corresponding to *region* $(1, 2)$, to which z_6 is initially assigned, and the remaining four to the nodes corresponding to its four neighboring *regions* $(1, 1), (0, 2), (1, 3), (2, 2)$. Each arc has capacity of 4 since the *region capacity* is 4 for each of these five *regions*. Finally, from each node corresponding to a *region*, there is an arc to the *sink* node t^o with capacity of 4, equal to the *region capacity* of the region corresponding to the tail node of the arc.

By solving the *max-flow problem* with integral flow, [Cormen 2003, Alsuwaiyel 1999] we obtain the number of blocks in a *netlist partition* u_i to be allocated to a particular *region* u_j. If the flow through the arc (u_i, u_j) is $f(i, j)$, then $f(i, j)$ is the number of CLBs that are chosen arbitrarily from u_i and reallocated to the *region* u_j. A *feasible flow* always indicate a feasible reallocation of the CLBs in the *regions* of the FPGA array. Within a *region*, the logic blocks are assigned to physical locations spirally. This completes the *Coarse grain* placement of CLBs on an FPGA array.

4.3.4 Placement of IOBs

After the CLBs are placed on to a 2D FPGA array, the IOBs (input/output blocks) are to be placed on the periphery of the array. We have formulated this problem as an instance of a *minimum weighted bi-partite matching problem* (MWBM) [Papadimitriou 2006, Alsuwaiyel 1999] as described below.

Let $G^b = \{B_1 \cup B_2, Q\}$ be a complete weighted bipartite graph, where $B_1 \cap B_2 = \phi$; the nodes in B_1 correspond to the primary inputs and primary outputs, and those in B_2 correspond to the IOB pad locations available on the periphery of the FPGA array of CLBs. Each pair of vertices (b_i, b_j), $b_i \in B_1$ and $b_j \in B_2$ contributes an edge $q = (b_i, b_j)$, and its weight is the *Manhattan distance* between the center of the bounding box of the net corresponding to the IOB $b_i \in B_1$ and the specific location corresponding to $b_j \in B_2$. For each matched edge (b_i, b_j) in the solution, we assign the IOB b_i to the location b_j. Figure 4.3 shows the MWBM formulation for placement of IOBs. The *bounding boxes* of two IOB *nets* are shown with shaded rectangles on the FPGA array after the CLB placement in Figure 4.3(a). The center of the bounding box is identified by a black circle. Each IOB *pad* is identified by the co-ordinate positions (x, y). Figure 4.3(b) shows the complete weighted bipartite graph. The edge weight which is the *Manhattan distance* between the center of the corresponding bounding box of an IOB net and the IOB location is shown next to each edge.

This completes the *initial placement* of the CLBs and IOBs on the FPGA array by our *Coarse grain* method.

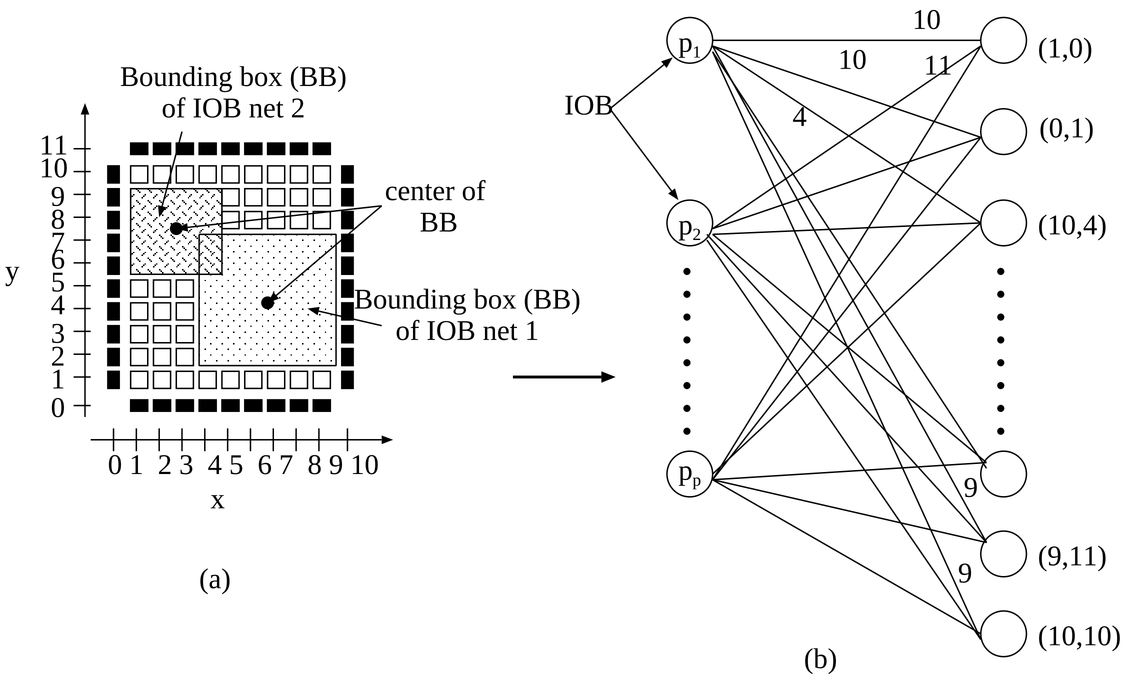


Figure 4.3: *Minimum weighted bi-partite matching* formulation for placement of IOBs: (a) the bounding box of two IOB nets; (b) the bi-partite graph for matching.

4.3.5 Time complexity of *Coarse grain* method

Theorem 4.1 *The time complexity of the Coarse grain method is* $O(n^2 \log n)$, *where* n *is the number of CLBs.*

Proof: The recursive min cut partitioning of CLB netlist by *hMetis* is an iterative process and the authors of *hMetis* [Karypis 1999a] claim that the time taken is almost linear in the number of hyperedges. As the number of hyperedges in the netlist hypergraph is $O(n)$, n being the number of CLBs, the time complexity of the partitioning step is $O(n)$.

After partitioning the CLB netlist by *hMetis*, the *Coarse grain* method allocates the partitions to *regions* in $O(n)$ time since the number of *regions* is $O(n)$. The CLBs in the *regions* are reallocated to neighboring *regions* using a *maximum flow* formulation. The time complexity of Goldberg-Tarjan max-flow algorithm [Goldberg 1988] is $O(|E||V|\log\frac{|V|^2}{|E|})$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in the flow graph. In our case, the number of nodes is $2\cdot\frac{n}{4}=\frac{n}{2}$, as there are $\frac{n}{4}$ *regions*. Thus, $|V|=\frac{n}{2}$ and $|E|=5n$. Therefore, the time complexity of the CLB placement by *Coarse grain* method is $O(n^2\log n)$.

For placement of IOBs on the periphery, we employ *MWBM*, which has a worst case time complexity of $O(|E|\sqrt{|V|})$ [Alsuwaiyel 1999], where $|V|$ and $|E|$ are the number of nodes and edges of the bipartite graph B with integer edge weights. In our case, $|V|$ is $4\sqrt{n}$ which is the number of IOB locations on the periphery of the $\sqrt{n}\times\sqrt{n}$ array of CLBs and $|E|=O(n)$. The time complexity of IOB placement is $O(n^{1.25})$. Hence, the time complexity of the *Coarse grain* method is $O(n^2\log n)$. $\square$

The *initial placement* generated thus far provides a fairly good quality solution. For further improvement in the placement quality, a very low temperature simulated annealing is executed on it to obtain the *final placement* as discussed in Section 3.3.6 of Chapter 3 for *cone* based placement approaches.

4.4 Fine grain placement

We observed that although the placement produced by *Coarse grain* method is better than our previous bottom-up greedy heuristics, the min-cut partitions get perturbed due to reallocation of blocks to neighboring *regions* thereby resulting in an increase in *BB cost*. Moreover, there is a run-time overhead due to the reallocation of CLBs from overloaded *regions* to its neighbors. In order to improve upon this drawback, we generate *netlist partitions* with finer *granularity*, and directly map the CLBs on to the 2D FPGA array using different types of *space filling curves*, which provide a way to transform a one dimensional arrangement to a two dimensional space.

4.4.1 Fine grain partitioning

In the *Fine grain* method, as opposed to the *Coarse grain* partitions where *region capacity* was set to 4, we set the *region capacity* to 1. Thus, the number of partitions z is nothing but the number of CLBs. However, for all practical purposes we have seen that $\frac{n}{2}$ partitions are sufficient for our purpose, and hence we partition the netlist hypergraph into $z=\frac{n}{2}$ partitions. We use the *hMetis* hypergraph partitioner to generate the balanced min-cut *partition tree*. As before, the leaves of the *partition tree* represent the *netlist partitions* and the arrangement of the leaf nodes from left to right represents a one dimensional arrangement of partitions. Before we delve into further details of our *Fine grain* placement method, a brief introduction to *space*

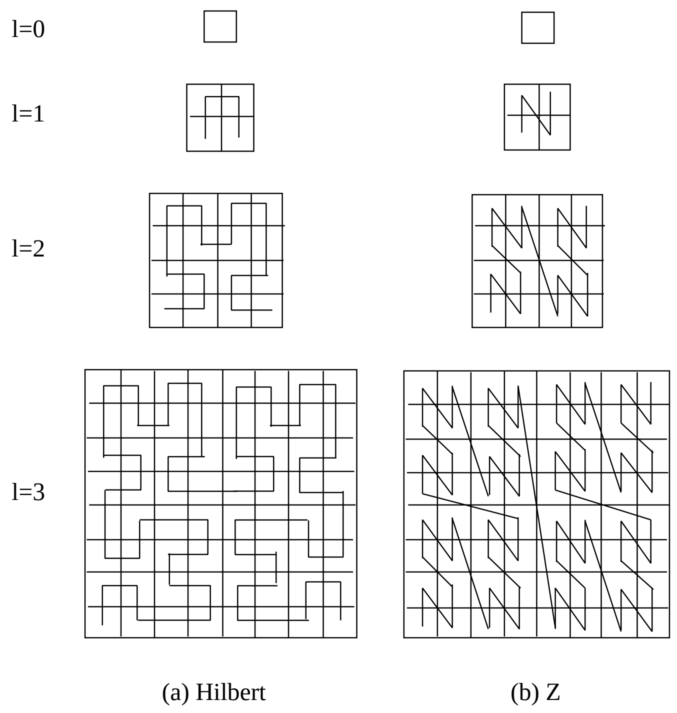


Figure 4.4: Generation of space filling curves for $l = 0, 1, 2, 3$.

filling curves is presented next.

4.4.2 Recursive space filling curve

Peano [Peano 1890] first defined and proved the existence of space filling curves. A *space filling curve* is defined to be a continuous map from the unit interval in 1D into the d-dimensional *euclidean space* that passes through every point of a d-dimensional region [Peano 1890, Hilbert 1891, Sagan 1994]. A discrete *space filling curve* provides a linear traversal or indexing of a multi-dimensional grid space. *Space filling curves* are commonly used to reduce a multi-dimensional problem to a 1-dimensional one [Asano 1997]. But our objective is the reverse. A given one dimensional arrangement is to be mapped on to a two dimensional array deterministically and by a closed form expression which can be computed in constant time. This is possible because a bijective mapping as defined next, is used.

Definition 4.2 (Space filling curve:) *For positive integers a and l, where $a = 2^l$, let us denote $[a] = \{1, 2, \ldots, a\}$. A 2-dimensional discrete space filling curve of length a^2 is a bijective mapping $C : [a^2] \to [a]^2$, that provides a linear indexing/traversal or total ordering of all grid points in $[a]^2$. This 2D grid is said to be of order l, and has sides of length $a = 2^l$.*

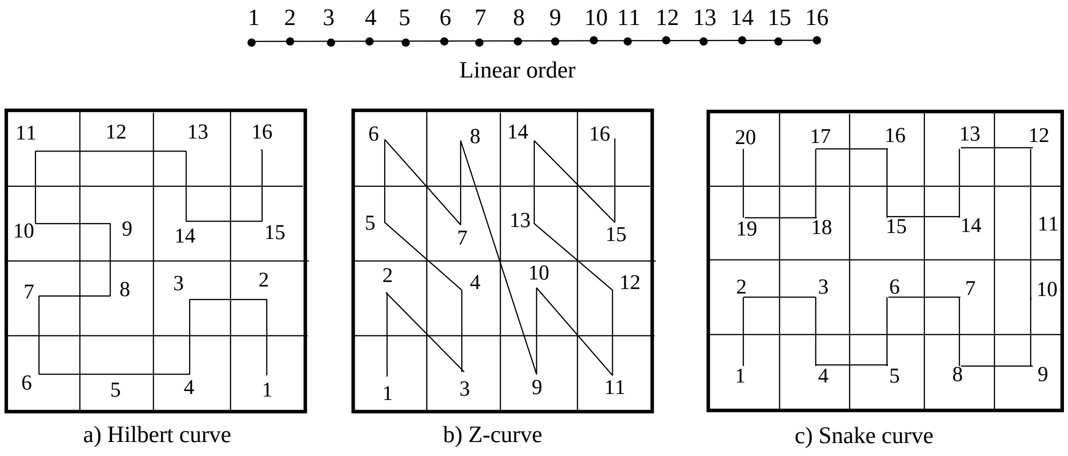


Figure 4.5: Placement of a one dimensional arrangement of 16 blocks on to a 4×4 array using *space filling curves.*

The generation of a 2D *space filling curve* of successive orders follows a recursive framework. Several *space filling curves* such as Hilbert, Z, gray code curves, are available in the literature [Sagan 1994] which are generated recursively. The *Hilbert* and *Z* space filling curve can be constructed from a basic unit shape as shown for $l = 1$ in Figure 4.4. The relative position and rotation of each unit shape is defined by its sequential position in the curve generation (see Figure 4.4). As the resolution of the curve increases, more unit shapes are required for its description, but the principle remains same as the original proposition of recursively dividing each part into smaller parts recursively. These curves can be generated using an EOL-type (extended zero-sided Lindenmayer) grammar [Wood 1987] that basically forces simultaneous rewriting at every cell of the grid partition. A more practical and easily implementable recursive procedure to generate *Hilbert* curve appears in [Breinholt 1998]. The *snake curve* used in the *Coarse grain* method, is also a *space filling curve.* However, this is generated non-recursively in a tail-recursive manner unlike *Hilbert* or *Z.*

In order to demonstrate the effectiveness of our proposed *Fine grain* placement method using *space filling curves*, we choose a set of representative *space filling curves* which are known to be easily constructable in linear time, using integer computations. As some of the *space filling curves* can be generated efficiently in a non-recursive manner, we selected the *Snake* curve to be one such. In the recursively constructable category, the simplest ones *Hilbert* and *Z* curves are taken. The reason is that another distinguishing feature of a *space filling curve* is whether it can be traced without long jumps, or discontiguities [Mokbel 2002]. *Hilbert* curve has no jumps, and Z curve has a jump to a non-contiguous cell at each level. The purpose was to observe the effect of this feature on the total wirelength of the placement obtained by applying the *space filling curve* mapping of the balanced min-cut based linear order to a 2D array. It should be noted that without loss of generality our proposed placement method may use other *space filling curves* as well, provided the associated mapping can be generated very quickly.

4.4.3 Placement using space filling curve

We select each of the *Hilbert*, Z and *snake space filling curves* as described in Section 4.4.2, to place the logic blocks on the $\sqrt{n} \times \sqrt{n}$ FPGA array. Essentially, this allocates a specific co-ordinate position for each of the CLBs in the one dimensional arrangement, using the sequence generated by the specific *space filling curve.* Figures 4.5(a), 4.5(b) and 4.5(c) respectively demonstrate the mapping of a one dimensional arrangement on to a 2D grid using *Hilbert*, Z and *snake* curves. For practical circuits, the size of the FPGA array may not necessarily be of the form $a = 2^l$. We draw the Hilbert curve for an arbitrary $R \times C$ array as follows. Let $N = max\{R, C\}$, and $a = 2^l$, where $l = \lceil \log_2 N \rceil$. Next, we find the *space filling curve* corresponding to a and crop the array of size $R \times C$ from the array $a \times a$ as in Figure 4.6.

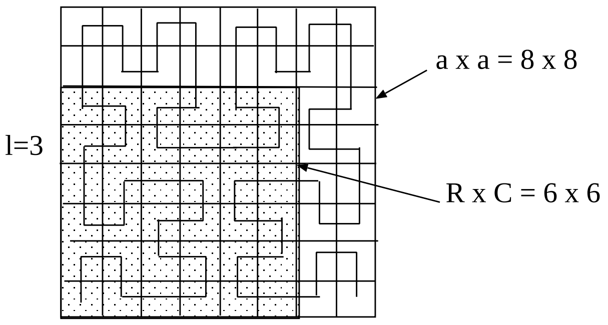


Figure 4.6: Cropping the curve for $R \times C$ array within $a \times a$.

In summary, we obtain a one dimensional arrangement of CLBs by recursive bi-partitioning of CLB netlist and map it to the two dimensional FPGA array by *space filling curve.* Finally, the IOBs are placed on the periphery of the array by formulation of a *minimum weighted bipartite matching* (MWBM) problem as described earlier for *Coarse grain* method in Section 4.3.4. This completes the *Fine grain* placement for FPGAs. In Chapter 5, we justify the success of this method. In order to improve the quality of the placement further, a low temperature simulated annealing step is executed as discussed in Section 3.3.6 of Chapter 3.

4.4.4 Time Complexity of *Fine grain* method

Theorem 4.2 *The time complexity of the Fine grain method is $O(n^{1.25})$, where n is the number of CLBs.*

Proof: The partitioning of the netlist hypergraph by *hMetis* takes $O(n)$ time, where n is the number of CLBs. The time complexity of placing the n CLBs using a *space filling curves* is $O(n)$ since n locations are visited only once during the construction of space filling curve. As explained in the proof of Theorem 4.3.5, the time complexity of IOB placement is $O(n^{1.25})$. Thus, the overall time complexity of *Fine grain* method is also $O(n^{1.25})$. □

4.5 Experimental results for top-down deterministic FPGA placement

In this section, we present the experimental results of our placement methodology and compare these with the placement and routing results produced by the popular FPGA placement tool VPR [Betz 1997]. The platform used is 1.2GHz Sun-Blade 2000 workstation. The *Library of Efficient Datatypes and Algorithms* (LEDA) [LEDA] is employed to solve the *max-flow* and MWBM formulations of Sections 4.3.3 and 4.3.4.

Table 4.1: Characteristics of MCNC FPGA placement benchmark circuits

1	2	3	4	5	6	7
Ckt	# CLBs	# Inputs	# Outputs	array size	# fine partitions	# coarse partitions
tseng	1047	52	122	33	523	256
ex5p	1064	8	63	33	532	256
apex4	1262	9	19	36	631	324
dsip	1370	229	197	54	685	729
misex3	1397	14	14	38	698	361
diffeq	1497	64	39	39	748	361
alu4	1522	14	8	40	761	400
des	1591	256	245	63	795	961
bigkey	1707	229	197	54	853	729
seq	1750	41	35	42	875	441
apex2	1878	38	3	44	939	484
s298	1931	4	6	44	965	484
frisc	3556	20	116	60	1778	900
elliptic	3604	131	114	61	1802	900
spla	3690	16	46	61	1845	900
pdc	4575	16	40	68	2287	1156
ex1010	4598	10	10	68	2299	1156
s38417	6406	29	106	81	3203	1600
s38584.1	6447	38	304	81	3223	1600
clma	8383	62	82	92	4191	2116

Table 4.2: Placement results for *wirelength driven* VPR

1	2	3	4	5	6	7
Ckt	Initial BB cost	Final BB cost	# of SA moves	Place time (s)	Delay ($\times 10^{-7} s$)	Channel width
tseng	418.41	92.05	15.5	98.49	0.90	7
ex5p	429.38	161.95	13.6	97.9	1.12	14
apex4	180.45	180.46	16.4	117.5	1.28	13
dsip	915.21	169.99	28.6	179.83	0.79	6
misex3	576.71	189.00	19.2	136.4	1.12	12
diffeq	674.62	146.86	22.1	160.83	0.95	8
alu4	612.34	191.66	22.1	149.7	1.21	10
des	1274.94	227.84	34.0	193.0	1.21	8
bigkey	1096.31	185.98	35.1	200.4	1.05	6
seq	791.10	247.74	27.4	198.1	1.27	12
apex2	905.76	268.14	29.1	215.1	1.21	13
s298	748.91	204.89	30.3	200.9	1.99	8
frisc	517.50	517.50	73.6	548.5	2.07	13
elliptic	2292.46	457.20	76.6	529.1	1.82	11
spla	2388.64	608.16	75.2	600.4	1.71	15
pdc	254.26	870.87	97.2	769.8	2.19	18
ex1010	3350.40	654.19	101.5	775.0	2.30	11
s38417	5816.43	670.56	167.5	1236.7	1.95	8
s38584.1	5597.42	657.87	172.2	1236.5	1.27	9
clma	8008.39	1395.24	233.4	1848.5	2.53	12

Table 4.1 shows the characteristics of twenty MCNC benchmark circuits, twelve of which have already been presented in Table 3.2. The columns 2 to 4 show the number of CLBs, primary inputs, and primary outputs respectively in the given circuit. Column 5 shows the minimum square array required to place all CLBs and IOBs. The number of coarse and fine grain partitions for each circuit are shown in columns 6 and 7 respectively.

As results of our proposed methods are compared with those of VPR, we first report the placement results obtained by VPR in *wirelength driven* and *timing driven* mode respectively with default parameters in Tables 4.2 and 4.3, in order to get a proper perspective of our method. Then in Tables 4.4 to 4.7, we compare our results with that of VPR with respect to various metrics, such as *BB cost*, *number of SA moves* and *critical path delay.* All these tables show the results obtained both in *wirelength driven* mode and the *timing driven* mode. Under each mode of operation, we report the results of the proposed *Coarse grain* method (column title Coarse), and the *Fine grain* method employing the three *space filling curves*, namely *Hilbert*, *Z* and *Snake.*

In Table 4.2, we have reported the *initial BB cost*, *final BB cost* after the execution of low temperature SA, the *number of SA moves* required, and the *time* taken by VPR to obtain the *final placement* in columns 2 to 5 respectively. The *critical path delay* and the *channel width* obtained after routing each circuit have

Table 4.3: Comparison of *GL%* between *timing driven* VPR and *wirelength driven* VPR

1	2	3	4	5
Ckt	GL%(BB Cost)	GL%(SA moves)	ratio of placement time	GL%(delay)
tseng	11	−18	2	−40
ex5p	11	−9	2	−28
apex4	8	−11	2	−33
dsip	18	−17	1	−22
misex3	6	−15	2	−6
diffeq	7	−14	2	−31
alu4	4	−15	2	−30
des	13	−13	2	−26
bigkey	13	−11	2	−40
seq	6	−17	2	−26
apex2	5	−16	2	−13
s298	12	−18	2	−19
frisc	14	−15	2	−37
Elliptic	9	−14	2	−26
spla	4	−15	2	−1
pdc	8	−13	2	−29
ex1010	3	−20	1	−24
s38417	2	−26	1	−46
S38584.1	4	−23	2	−25
clma	7	−16	2	−10
Avg:	8.3	−15.7	1.7	−25.5

been reported in the last two columns.

We define *GL%(metric)* as a percentage of gain or loss with respect to a certain metric of the *final placement* of VPR as follows:

$$GL\%(metric) = 100.\frac{Ours(metric) - VPR(metric)}{VPR(metric)} \tag{4.1}$$

where *metric* can be BB cost, number of SA moves, or *critical path delay.* A negative value means our method yields a better result than that by VPR.

Table 4.3 shows the trade off between *timing driven* VPR, and *wirelength driven* VPR. For *timing driven* VPR, the *final BB cost* increases (column 2), the number of SA moves decreases (column 3), while the *critical path delay* (column 5) improves significantly over those for *wirelength driven* VPR. However, the time to place each circuit (column 4) by *timing driven* VPR is 2× that by *wirelength driven* VPR.

In Table 4.4, we show how good our *initial placement* solution (without execution of low temperature SA) is with respect to BB cost, vis-a-vis the *final placement* obtained by VPR. The columns 2 to 5 show the GL% with respect to the *initial BB cost* by our method over *final BB cost* of VPR in *wirelength driven* mode. The columns 6 to 9 show the same in *timing driven* mode. The last row shows the averages. We observed that, on the average our *initial placement* solution is within

Table 4.4: Comparison of $GL\%$ (Initial BB Cost) by our method vs. Final BB Cost by VPR; [H:Hilbert, Z:Z, S:Snake]

1	2	3	4	5	6	7	8	9	10
	Wirelength driven BB cost				Timing driven BB cost				
	Coarse	Fine			Coarse	Fine			Speedup
Ckt		H	Z	S		H	Z	S	$\frac{VPR}{Ours}$
tseng	54	44	53	73	38	30	38	56	31.8
ex5p	44	53	46	50	30	38	31	35	39.2
apex4	43	49	45	54	29	35	30	39	36.7
dsip	64	65	60	67	39	40	36	42	47.2
misex3	43	56	45	55	35	47	37	46	44.0
diffeq	49	59	45	67	39	48	35	55	38.3
alu4	38	48	39	48	33	43	34	43	40.4
des	86	91	88	93	65	69	66	71	43.9
bigkey	42	45	43	46	26	29	27	30	39.3
seq	50	53	47	55	42	44	39	47	46.1
apex2	48	60	52	59	41	53	45	52	43.9
s298	26	39	26	39	12	25	12	24	45.7
frisc	48	52	48	59	44	48	45	56	56.0
Elliptic	47	48	44	57	35	36	32	44	53.4
spla	51	51	48	56	45	45	42	50	60.6
pdc	54	52	50	55	42	40	38	43	61.1
ex1010	80	70	74	93	74	65	69	87	53.1
s38417	64	56	68	82	60	53	64	78	58.6
S38584.1	72	67	73	89	68	64	69	85	58.6
clma	59	55	60	71	57	54	58	69	72.2
Avg:	53.1	55.8	52.7	63.5	41.4	43.7	41.2	51.1	48.5

41% to 63% of the final solution of VPR. However, the placement run-time is just a few seconds irrespective of the size of the circuits. The last column shows the average *speedup* of our *initial placement* solution by both *Coarse grain* and *Fine grain* method over VPR for each circuit. Our *space filling curve* based *initial placement* method is about 48× faster than VPR on an average, as reported in the last row of the last column. For both wirelength and *timing driven* modes, the *initial placement* obtained by *Z*, *Coarse*, *Hilbert* and *Snake* curve are in the increasing order of performance (BB cost). But, for bigger circuits (above 4000 CLBs), *Hilbert* curve performs the best.

The GL% with respect to final BB cost, after execution of low temperature SA is shown in Table 4.5. We observed that on the average the GL% (Final BB cost) lies between 1% to 3% for *wirelength driven* and 4% to 7% for *timing driven* mode, thus indicating that our *final placement* quality is comparable to VPR. The performance of *Hilbert* curve is the best for *wirelength driven* case.

The speedup for all four variations is significantly high, as shown in Table 4.6. In Table 4.6, we report the GL% (SA moves). This metric is based on the number of SA moves required during execution of low temperature SA on our *initial placement* solution. We observed that this GL% (SA moves) ranges between −47% to −67%

Table 4.5: Comparison of *GL*% (Final BB Cost) of our methods vs. VPR

1	2	3	4	5	6	7	8	9
	Wirelength driven BB cost				Timing driven BB cost			
	Coarse	Fine			Coarse	Fine		
Ckt		Hilbert	Z	Snake		Hilbert	Z	Snake
tseng	5	2	3	7	1	−1	−2	2
ex5p	2	2	2	3	4	5	2	6
apex4	3	1	1	0	9	8	8	5
dsip	1	1	0	1	3	14	16	18
misex3	1	1	0	1	4	8	4	8
diffeq	1	0	−2	−1	2	5	1	4
alu4	1	0	0	0	2	6	5	3
des	11	10	9	10	9	16	14	18
bigkey	2	1	2	3	3	8	10	15
seq	1	4	5	1	2	6	1	0
apex2	2	0	4	6	10	7	9	14
s298	1	0	0	1	3	3	−1	2
frisc	4	4	3	5	1	2	2	2
Elliptic	0	−1	−1	−1	7	2	3	10
spla	−1	−1	7	2	5	1	2	4
pdc	3	0	3	1	4	4	4	7
ex1010	5	2	2	5	28	13	11	17
s38417	1	1	1	−1	5	2	3	7
S38584.1	2	1	−1	2	0	1	1	1
clma	2	1	3	3	0	1	1	0
Avg:	2.3	1.4	2.2	2.4	5.1	5.5	4.7	7.1

implying that our placement solution converges much faster. In case of *wirelength driven* SA, the gain is the maximum for *Coarse*, whereas usage of any one of the space filling curves *Hilbert*, *Z* and *Snake* has similar gains in terms of SA moves. However, for the *timing driven* mode, *Hilbert* curve has the least gain in SA moves. We have also reported the actual average *speedup* (ratio of CPU times on the same platform) of our method over VPR in the last column. It shows that on the average, our placement with low temperature SA is about 1.6× faster than VPR.

Finally, Table 4.7 shows the GL% (critical path delay) obtained after routing our *final placement* using the router provided by VPR. Here also, we note that on the average the GL% lies between −0.3% to 5.1% implying the critical path delay of our placement after routing is comparable to VPR. For both wirelength and *timing driven* modes, *Hilbert* curve produces the best gain in *critical path delay* compared to VPR. Comparing the *BB cost*, *speedup* and *critical path delay* in both *wirelength* and *timing driven* mode, the solution produced by *Hilbert* seems to be the most promising one.

The results establish that our simple but fast placement technique generates placement solution very quickly with little or no sacrifice in the quality of placement solution. We have also noted that the channel width does not increase while routing our placement solution, excepting one or two cases where it increases by at most 2.

Table 4.6: Speedup of our methods over VPR in terms of $GL\%$ (SA moves) and ratio of CPU times; [H:Hilbert, Z:Z, S:Snake]

1	2	3	4	5	6	7	8	9	10
	Wirelength Driven				Timing Driven				
	Coarse	Fine			Coarse	Fine			Speedup
Ckt		H	Z	S		H	Z	S	$(\frac{VPR}{Ours})$
tseng	−62	−48	−48	−49	−57	−38	−51	−49	1.4
ex5p	−66	−50	−50	−50	−78	−51	−82	−58	2.0
apex4	−71	−51	−54	−47	−58	−58	−58	−58	1.6
dsip	−67	−54	−50	−51	−77	−45	−76	−72	1.7
misex3	−69	−52	−53	−51	−81	−40	−74	−53	1.6
diffeq	−66	−48	−50	−47	−55	−48	−70	−47	1.5
alu4	−65	−47	−44	−48	−62	−41	−64	−59	1.8
des	−66	−50	−50	−51	−78	−67	−79	−76	2.0
bigkey	−64	−50	−50	−49	−70	−46	−67	−72	1.8
seq	−64	−49	−46	−49	−79	−53	−61	−72	1.6
apex2	−61	−42	−43	−44	−59	−51	−63	−72	1.6
s298	−66	−48	−47	−49	−66	−33	−57	−54	1.3
frisc	−67	−50	−50	−50	−58	−58	−59	−58	1.8
elliptic	−67	−50	−50	−50	−49	−43	−43	−41	1.5
spla	−67	−40	−43	−40	−50	−69	−61	−58	1.3
pdc	−60	−39	−39	−37	−53	−37	−35	−38	1.6
ex1010	−61	−38	−41	−36	−41	−39	−32	−30	1.5
s38417	−67	−50	−50	−50	−43	−33	−35	−36	1.4
s38584.1	−67	−50	−50	−50	−58	−59	−57	−55	1.7
clma	−67	−50	−50	−50	−57	−56	−57	−57	1.4
Avg:	−65.4	−47.9	−47.9	−47.5	−61.6	−48.3	−59.1	−55.8	1.6

Table 4.7: Comparison of $GL\%$ (Critical path delay) of our methods over VPR

1	2	3	4	5	6	7	8	9
	Wirelength Driven				Timing Driven			
	Coarse	Fine			Coarse	Fine		
Ckt		Hilbert	Z	Snake		Hilbert	Z	Snake
tseng	13	17	14	8	4	2	9	11
ex5p	8	2	14	9	3	8	13	9
apex4	5	−4	27	−7	−1	0	0	0
dsip	0	−1	7	2	5	−13	1	5
misex3	5	7	0	2	−11	−5	−4	−3
diffeq	4	3	2	1	4	9	−2	−12
alu4	−10	−1	−9	−10	19	−1	23	10
des	31	2	20	3	15	16	11	9
bigkey	3	4	5	2	15	15	−5	0
seq	1	−6	−4	6	−6	−6	11	6
apex2	3	1	1	13	9	−11	−6	−6
s298	5	2	0	1	−15	−10	−6	−15
frisc	4	2	1	0	1	−3	2	−2
Elliptic	3	2	1	2	18	−12	10	−1
spla	−8	−2	−11	21	−10	−10	−4	−5
pdc	32	4	−3	2	1	11	11	40
ex1010	0	−15	−12	−13	9	2	0	−3
s38417	2	3	2	−1	8	−4	17	11
S38584.1	−2	2	4	1	1	2	4	0
clma	2	2	3	0	2	6	3	0
Avg:	5.1	1.2	3.1	2.0	3.7	−0.3	4.5	2.8

4.6 Conclusion

In this chapter, we have proposed a very fast technique to place a netlist of CLBs on FPGAs with a small sacrifice in the quality of solution. The novel contribution is in using space filling curves to generate a good quality *initial placement* very quickly which ultimately reduces the convergence time of an iterative improvement method. Here, three representative *space filling curves*, namely *Hilbert*, *Z* and *snake* have been used to demonstrate the efficacy of the proposed *Fine grain* placement method; as a matter of fact, any *space filling curve* that is easily constructable in linear time can be used.

The experimental results on benchmark circuits indicate that our *initial placement* method produces a solution about 47% and 52% faster than VPR by employing *Coarse grain* and *Fine grain* method respectively. As the BB cost of our *initial placement* is within 47% and 53% of *final BB cost* of VPR for *Coarse grain* and *Fine grain* partition respectively, we employed a low temperature SA on our *initial placement* solution and compared the BB cost, critical path length and the overall runtime with that of VPR. We observed that for our *Coarse grain* method, the *final BB cost* and the *critical path delay* is within 4% of *final BB cost* and *critical path delay* obtained by VPR. However, the gain in number of SA moves is about 63% and it is 1.6× faster than VPR on the average. In case of our *Fine grain* method, the *final BB cost* and the *critical path delay* is within 2% and 4% of VPR respectively, whereas, it is 1.6× faster and requires 53% fewer moves on the average.

The results also show that the *Coarse grain* method produces the final solution faster than *Fine grain* method but of slightly lower quality than that of *Fine grain* method. Moreover, while the partitioning-based placement method PPFF [Maidee 2003], achieves an improvement of 2% in *critical path delay* at the cost of additional 0.28% runtime (in case the routing profile statistics are not available or ineffective for the circuit under consideration) and channel width over VPR, our method produces a solution of nearly comparable quality in much less time. The overall performance of *Fine grain* method using *Hilbert* space filling curve showed interesting results. This led to a deeper investigation in the theory behind the success of this method described in the next chapter. As stated earlier, our fast methods can also be applied to small islands of CLBs in *heterogeneous* FPGAs in the context of fast reconfiguration.

CHAPTER 5

Efficient FPGA Placement using Space Filling Curves: Theory Meets Practice

Contents

5.1 Introduction

In Chapter 4, we proposed a fast top-down placement method, where a one dimensional placement is obtained first, which is then arranged in two dimensions using several *space filling curves*. It was observed that a particular *space filling curve*, namely, *Hilbert*, performed the best in terms of quality of placement. This chapter presents an analytical viewpoint of this strategy for FPGA placement and gives bounds to the quality of solution achieved by this method.

5.1.1 FPGA placement: theory and practice

VLSI placement, be it for ASIC or FPGA, is a computationally difficult problem that challenges both theoreticians and practitioners alike. The existing approaches to solve the placement problem have branched into two different directions. One uses stochastic iterative heuristics with many tunable parameters and tries to reach the optimal solution, but no rigorous theoretical analysis can be given about the quality of the solution. These methods suffer from all or some of the following drawbacks - no theoretical guarantee, immense running time for good quality solution, not scalable for all practical purposes. The problem under study being NP-hard [Shahookar 1991], the other line of study mainly concerns researchers dealing with approximation algorithms where the aim is to give theoretical bounds on the deviation of the obtained solution from the optimal one. But, as literature on placement shows, there has been almost no effort at bridging the gap between theory and practice.

On the theoretical front, the FPGA placement problem can be modeled as a *graph embedding* problem on a two dimensional grid that minimizes a cost resembling the wirelength [Even 2000, Vempala 1998]. The model is formally defined in Section 5.3. The best known approximation ratios for the *graph embedding* problem are polylogarithmic [Even 2000, Feige 2007, Rao 1998]. This implies worsening theoretical guarantees with increase in the problem size. On the other hand, inapproximability results (results that prove that the design of an approximation algorithm with a small approximation ratio is impossible unless $\mathsf{P}=\mathsf{NP}$) [Vazirani 2001] have also eluded researchers. The best theoretical result for this problem is due to Even et al. [Even 2000] where the authors propose an $O(\log n \log\log n)$ approximation result. As it turns out, the one dimensional version of this problem is also NP-hard [Garey 1979]. Rao and Richa [Rao 1998] showed that the approximation ratio of the one dimensional version is $O(\log n)$. The current approximation result stands at $O(\sqrt{\log n}\log\log n)$ [Feige 2007].

These algorithms are of polynomial time complexity, use advanced concepts, and are theoretically very elegant. But the running time of these algorithms are prohibitively high for VLSI practitioners since they involve solving linear programs with exponential number of constraints by the ellipsoid method or semidefinite program [Schrijver 1998]. Moreover, the theory developed thus far is tuned for VLSI placement modeled as *graph embedding.* But in FPGA placement problem, the model of a hypergraph is more accurate for representing a given circuit netlist. To the best of our knowledge, there have been no efforts on the part of VLSI practitioners to bring this theory into practice. Practical algorithms should be fast, give reasonably acceptable solutions and should scale up well for very large circuits [Sarrafzadeh 2001]. In effect, as Vygen [Vygen 2007] points out, one has to accept that no approximation guarantee can be given for practical algorithms. This has led practitioners to look for heuristics for global placement that can be broadly classified into three categories: (i) stochastic iterative search, most notably, using simulated annealing (SA), (ii) recursive partitioning, and (iii) analytical placement.

All of the above works aim at faster method for routable placement with better or comparable critical path delay as in VPR. However, no work addresses the quality of the initial placement as obtained by their deterministic method in terms of half-perimeter bounding box cost, the routability of the placement or the critical path delay of this placement after routing. Also, there is no theoretical guarantee on how close their placement solution is to the optimal solution.

Thus, we observe that there is a wide gap between theory and practice regarding VLSI placement. Our effort, in this chapter, is to bridge this gap. We propose a very simple placement method with theoretical bound on the quality of the placement. Further, to emphasize the suitability of this fast yet effective method in the context of reconfigurability, we route the placement and show that all the placements are routable with reasonable critical path delay when compared to VPR.

The rest of the chapter is organized as follows. An overview of our motivation and approach for FPGA placement is presented in Section 5.2. Section 5.3 defines the FPGA placement problem as a *graph embedding problem.* Section 5.4 forms the theoretical backbone of our work wherein we discuss approximation algorithms for FPGA placement. We discuss the ways of implementing the theoretical algorithmic results for real FPGAs in Section 5.5. Section 5.6 reports experimental results. Finally, the concluding remarks appear in Section 5.7. Although, routing is not the main focus of this chapter, in order to justify the quality of our fast placement method, we report the routability of the placed design in the experimental results.

5.2 Our contribution

We first extend the existing *graph embedding* approximation algorithms [Feige 2007, Even 2000] to hypergraphs in order to prove an $O(d\sqrt{\log n}\log\log n)$ bound for 1D (i.e., *linear arrangement*), and an $O(d\log n\log\log n)$ bound for the 2D case. Next, we obtain a bound on the quality of placement obtained by the *fine grain* method proposed in Section 4.4.1 of Chapter 4. The bound is derived for the HPWL, the *half-perimeter bounding box* cost only, without considering the congestion and routability in particular.

We obtain a *linear arrangement* of the vertices of the hypergraph modeling the circuit netlist using an approximation algorithm, and then employ a recursive *space filling curve* [Sagan 1994] for deterministically mapping this *linear arrangement* to a two dimensional grid. We establish an approximation bound of $O(\sqrt[4]{\log n}\sqrt{kd}\log\log n)$ for this method. However, the theoretical approximation bound is not tight and the time complexity is still high. This leads us to the pertinent issue of how to adapt the algorithms with theoretical bound into practice.

We answer this by replacing the complex approximation algorithm for determining optimal *linear arrangement*, with the left to right order of the leaves of the recursive bi-partition tree produced by a top-down min-cut hypergraph partitioning heuristic (e.g. hMetis [hMetis , Karypis 1999a]). This method runs in just a few seconds for standard benchmark circuits to yield reasonably good solutions. We report that the solution obtained is routable even without any iterative improvement or legalization heuristic. In the existing literature, we have not found reports on routability for the deterministic heuristics. To sum up, our technique (i) transforms the netlist hypergraph to a *linear arrangement* of nodes of the hypergraph using top-down hypergraph bi-partitioning techniques, (ii) maps this *linear arrangement* directly on to the 2D grid using a recursive *space filling curve.* The intuition behind using *space filling curves* is that on one hand the target location for each node in the *linear arrangement* is computable in constant time and on the other, it retains the locality properties of the 1D *linear arrangement.* Although it is not the main focus of our work, we also show that our placement can be improved by low temperature simulated annealing schedule with fast convergence, just as placements obtained by other methods [Maidee 2003, Xu 2005, Gopalakrishnan 2006] have been improved using local search or simulated annealing.

5.3 Graph embedding and approximation algorithms

In contrast to the heuristic methods developed by VLSI practitioners for placement, there has been a different line of study where researchers look at designing approximation algorithms for producing solutions that lie within certain bounds of the optimal solution [Even 2000, Vempala 1998]. These works mainly formulate the VLSI layout problem as embedding a graph in a d-dimensional grid. For $d = 1$, the *graph embedding* problem is basically the *graph optimal linear arrangement* (GOLA) problem, which is known to be NP-hard [Garey 1979]. Most of these works deal with laying out a graph on a grid optimizing a cost, which in a way resembles an estimate of the total wirelength. But, a more accurate model of a circuit represented as netlists of CLBs, is a hypergraph. Before presenting a brief review of existing results, a series of problem definitions [Bhasker 1987, Even 2000, Rao 1998, Feige 2007] related to embedding of graphs and hypergraphs, are presented below.

Problem 5.1 (Graph Optimal Linear Arrangement (GOLA):) *Given an undirected graph $G = (V, E)$, the problem is to find a linear arrangement of the vertices $h : V \to \{1, \dots, |V|\}$, that minimizes the cost function $\sum_{(i,j)\in E} |h(i) - h(j)|$, i.e., the sum of the lengths of the edges in the arrangement.*

Problem 5.2 (Graph Placement on Grid (GPG):) *Given an undirected graph $G = (V, E)$, the problem is to find an embedding of G in a two dimensional grid, or equivalently, a one-to-one mapping, h of G to a subgraph containing $|V|$ vertices of the two dimensional grid, such that the cost function $\sum_{(i,j)\in E} d(h(i), h(j))$ is minimized. Here $d(x, y)$ is the number of grid edges in the shortest path between x and y along the grid.*

For FPGAs, the circuit represented by the CLB netlist is more realistically modeled as a hypergraph $H = (V, S)$, where $V = \{1, 2, \dots, n\}$ are the n CLBs, and $S = \{S_1, S_2, \dots, S_k\}$ are the k hyperedges or nets with each hyperedge S_i being a subset of V. Let a net i $(1 \leq i \leq k)$ have n_i number of CLBs, i.e., $|S_i| = n_i$. Let $d = \max_{i=1}^{k}\{n_i\}$ and certainly $d \leq n$. Now, we progress to the following problems.

Problem 5.3 (Hypergraph Optimal Linear Arrangement (HOLA):) *Given a hypergraph $H = (V, S)$, the problem is to find a linear arrangement of the vertices $h : V \to \{1, \dots, |V|\}$ such that the cost $\sum_{i=1}^{k} max_{q,l\in S_i}\{|h(q) - h(l)|\}$ is minimum.*

Problem 5.4 (Hypergraph Placement on Grid (HPG):) *Given a hypergraph $H = (V, S)$, the problem is to find an embedding of H in a two dimensional grid, or equivalently, a one-to-one mapping, h of H to a subgraph containing $|V|$ vertices of the two dimensional grid, such that the cost function*

$$HPWL = \sum_{i=1}^{k} (bb_x(i) + bb_y(i))$$

is minimized. Here, $bb_x(i)$ is the x span of net i, $bb_y(i)$ is the y span of net i, and $bb_x(i) + bb_y(i)$ is termed as the half-perimeter bounding box of net i.

Note that the shortest path between vertices i and j in the 2D grid, as defined in GPG, is similar to the $HPWL$-cost as defined in HPG. Both measure the half-perimeter of the enclosing rectilinear bounding box, i.e., HPWL cost.

Bhasker and Sahni [Bhasker 1987] showed that the problem of computing the ε-approximation solution for GOLA and HOLA are NP-hard. They provide branch-and-bound and dynamic programming algorithms for optimal solutions for HOLA, but obviously the algorithms are not of polynomial time complexity. Approximation algorithms for GOLA and GPG were designed by Even et al. [Even 2000]. Their approximation algorithms work in cases where divide-and-conquer is applicable and a fractional spreading metric can be computed in polynomial time. A spreading metric on a graph is an assignment of rational lengths to edges such that subgraphs for which the optimization problem is non-trivial, are spread apart in the associated metric space. The sum of the lengths of these edges multiplied by the corresponding weights gives a lower bound on the cost of solving the optimization problem. The crux of the strategy by Even et al. is a novel divide-and-conquer that divides not according to the sizes of the subproblems, but on the cost of solving the optimization problem which is bounded from below by the volume of the spreading metric. Their approximation bounds for both GOLA and GPG are $O(\log n \log\log n)$. Following on the work of Even et al. [Even 2000], Rao and Richa [Rao 1998] improved the bound to $O(\log n)$ for GOLA by using better graph separators. This has been further tightened to $O(\sqrt{\log n}\log\log n)$ [Feige 2007]. So, the current approximation bound for GOLA stands at $O(\sqrt{\log n}\log\log n)$ [Feige 2007], and for GPG, $O(\log n \log\log n)$ [Even 2000]. Apart from minimizing the sum of distances as done in the above reviewed works, there has been effort in minimizing the maximum edge length. Vempala [Vempala 1998] obtained an $O(\log^{3.5} n)$ approximation algorithm for this problem.

The above algorithms though polynomial in nature, have high time complexity as they need to solve a linear program by the ellipsoid method or semidefinite program as a subpart. This is understandable from a theoretical standpoint since the challenge is to reduce the approximation bound within polynomial time. Our goal on the contrary, is designing approximation algorithms with theoretical bounds for HOLA and HPG, that run in reasonable time and are easy to implement. The application under consideration is FPGA placement. We present two types of results. One that extends the work of [Rao 1998, Feige 2007] on GOLA to HOLA, and the work of [Even 2000] on GPG to HPG. These are mainly of theoretical interest. Next, we design an approximation algorithm for HPG based on a solution for HOLA followed by application of a *space filling curve.* We show that this algorithm can be

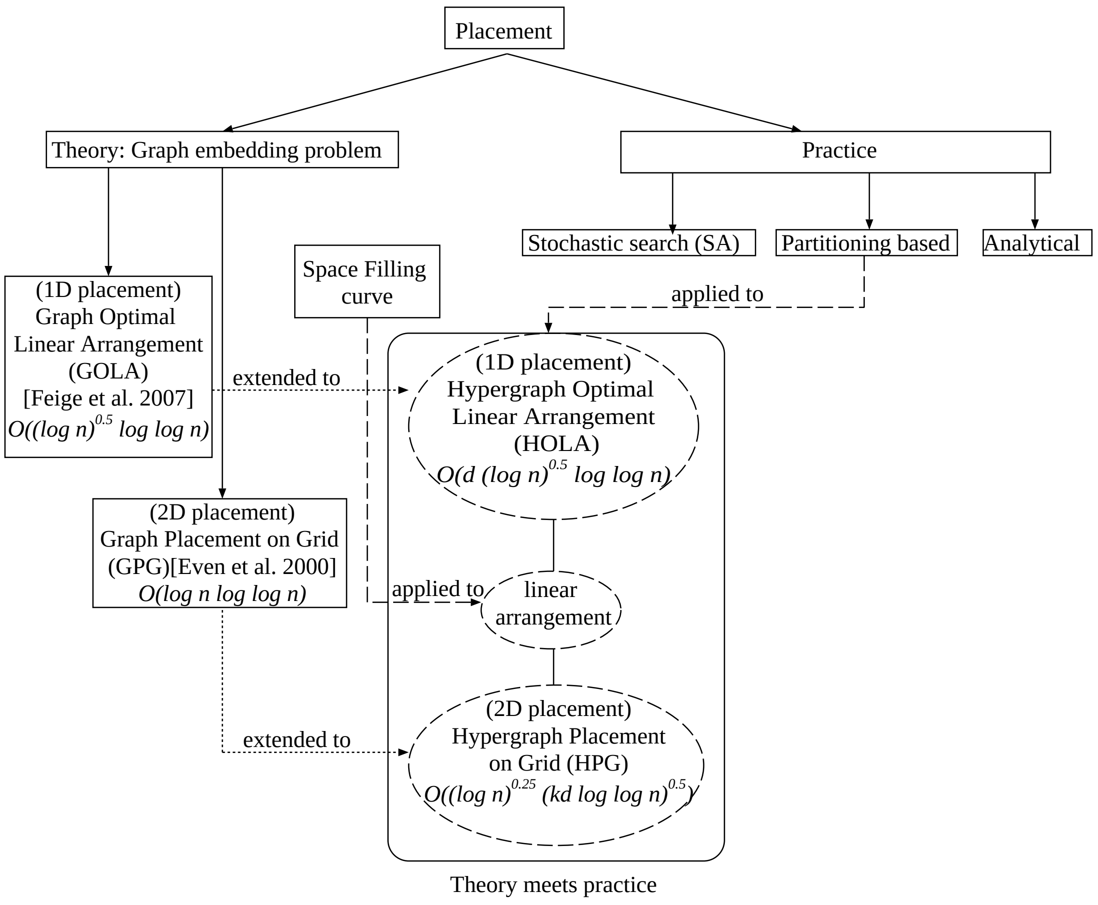

Placement
Theory: Graph embedding problem
Practice
Stochastic search (SA)
Partitioning based
Analytical
Space Filling curve
applied to
(1D placement) Graph Optimal Linear Arrangement (GOLA) [Feige et al. 2007] $O((\log n)^{0.5} \log\log n)$
extended to
(1D placement) Hypergraph Optimal Linear Arrangement (HOLA) $O(d (\log n)^{0.5} \log\log n)$
(2D placement) Graph Placement on Grid (GPG)[Even et al. 2000] $O(\log n \log\log n)$
applied to
linear arrangement
extended to
(2D placement) Hypergraph Placement on Grid (HPG) $O((\log n)^{0.25} (kd \log\log n)^{0.5})$
Theory meets practice

Figure 5.1: A schematic indicating the role of our work.

adapted for practical FPGA placement running in near-linear time if the *approximation algorithm* for HOLA is replaced by a *linear arrangement* obtained from a recursive min-cut partitioning of the CLB netlist. We provide experimental results on standard benchmarks that support our claim.

A schematic representation of our line of research is presented in Figure 5.1. The arrows show the steps/directions of the development of different approaches in both theory and practice. The dotted lines indicate the theoretical derivation of approximation ratio of our method from existing results. The dashed lines show the theoretically or practically available concepts or techniques used in our method. The steps of our method are marked within oval shapes. The rectangle in the center depicts how our approach is bridging the gap between theory and practice by providing a solution with an effective theoretical bound.

5.4 Approximation algorithms for FPGA placement

5.4.1 Extending approximation algorithms for graphs to hypergraphs

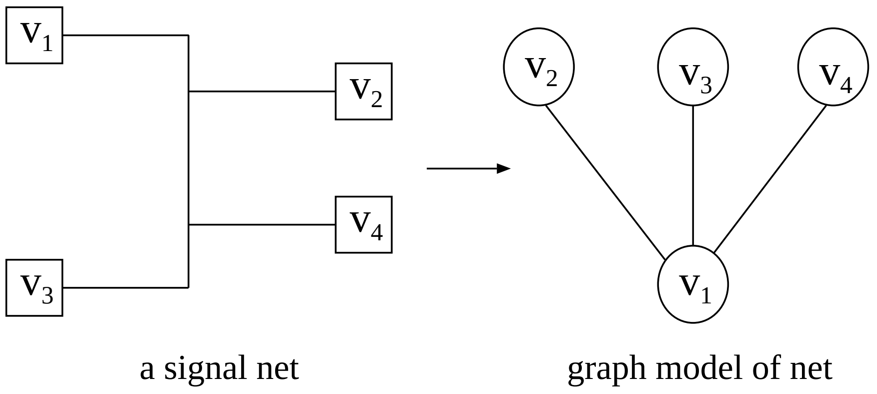


Figure 5.2: A net/hyperedge and its corresponding graph; v_1 is the source.

First, it is to be noted that GOLA is a special case of HOLA and similarly, GPG is a special case of HPG. We now deduce relations between costs of GOLA and HOLA.

Given a hypergraph $H(V, S)$, we construct a graph $G^* = (V, E)$ as follows. For each hyperedge $S_i \in S$, a vertex $v_i\,(\in V)$ is designated as source from which edges are connected to vertices $v_j\,(v_j \in V$ and $v_j \neq v_i)$, i.e., we add edges $(v_i,\, v_j)$ to E. So, for hyperedge $S_i \in S$, we add $(n_i - 1)$ edges. Figure 5.2 provides an illustration of how edges are added for a particular hyperedge.

Lemma 5.1 *$OPT_{GOLA} \leq d \cdot OPT_{HOLA}$, where OPT_{HOLA} is the optimum cost of HOLA for a hypergraph $H = (V, S)$, OPT_{GOLA} is the optimum cost of GOLA for the corresponding graph $G^* = (V, E)$, and $d = max_{i=1}^{k}\{n_i\}$.*

Proof: Let h be the permutation obtained by HOLA of the nodes in H, and

$$cost(h) = \sum_{i=1}^{k} \max_{q,l \in S_i}\{|h(q) - h(l)|\}$$

where the cost contribution of hyperedge S_i is $\max_{q,l \in S_i}\{|h(q) - h(l)|\}$. When the hyperedge S_i is transformed to edges of the graph G^*, $(n_i - 1)$ edges are added. Of them, the maximum contribution is from $\max_{q,l \in S_i}\{|h(q) - h(l)|\}$ which adds to the HOLA cost. So, the corresponding graph cost for the hyperedge S_i cannot exceed

$(n_i - 1) \cdot \max_{q,l \in S_i}\{|h(q) - h(l)|\}$. Thus,

$$\begin{aligned} d \cdot \mathrm{OPT}_{\mathrm{HOLA}} = d \cdot \mathrm{cost}_H(h) &= \max_{i=1}^{k}\{n_i\} \cdot \sum_{i=1}^{k} \max_{q,l \in S_i}|h(q) - h(l)| \\ &\geq \sum_{i=1}^{k} n_i \max_{q,l \in S_i}|h(q) - h(l)| \\ &\geq \mathrm{cost}_{G^*}(h) \\ &\geq \mathrm{OPT}_{\mathrm{GOLA}}. \end{aligned}$$

□

Theorem 5.1 *HOLA is $O(d\sqrt{\log n}\log\log n)$ approximable.*

Proof: Using the $O(\sqrt{\log n}\log\log n)$ approximation algorithm $\mathcal{A}(G)$ for GOLA [Feige 2007], which produces a solution with cost $A(G)$, we have

$$\frac{A(G)}{\mathrm{OPT}_{GOLA}} \leq c' \sqrt{\log n}\log\log n, \text{ where } c' \text{ is a constant.} \qquad (5.1)$$

By Lemma 5.1 and Equation 5.1, we have

$$d \cdot \mathrm{OPT}_{HOLA} \geq \mathrm{OPT}_{GOLA} \geq \frac{A(G)}{c'\sqrt{\log n}\log\log n} \qquad (5.2)$$

Thus, we have $\frac{A(G)}{\mathrm{OPT}_{HOLA}} \leq c' \; d \; \sqrt{\log n}\log\log n$.

In order to get an $O(d\sqrt{\log n}\log\log n)$ approximation algorithm for HOLA, we transform the hypergraph netlist to the corresponding graph G^* as stated above, and then run the $O(\sqrt{\log n}\log\log n)$ approximation algorithm for GOLA [Feige 2007] on G^*. □

As a matter of fact, results similar to that of Lemma 5.1, hold for the costs of GPG and HPG by replacing the cost functions of GOLA and HOLA with the half-perimeter bounding box cost. Using this observation, we can proceed along similar lines as Theorem 5.1, and use an approximation algorithm for GPG by Even at al. [Even 2000] to obtain an approximation algorithm for HPG as stated in the following theorem.

Theorem 5.2 *HPG is $O(d \; \log n \log\log n)$ approximable.*

These results are merely of theoretical interest since these are not implementable in reasonable time as mandated by VLSI practitioners. In the next subsection, we begin a line of study that would finally lead to an algorithm with reasonable running time.

Algorithm 5.1: SFCTheoreticalPlace: Placement using HOLA and SFC.

```
Input  : Hypergraph representing a given netlist of $n$ CLBs; a 2D array of
         FPGA chip
Output: Placement of the CLBs on a given 2D array
1 Call the $O(d\sqrt{\log n}\log\log n)$ approximation algorithm for HOLA as per
  Theorem 5.1;
2 Map the Hypergraph Optimal Linear Arrangement (HOLA) obtained directly
  on to the two dimensional grid using a recursive space filling curve, e.g.,
  Hilbert curve;
```

5.4.2 Approximation algorithm for HPG using space filling curve

Our algorithm for placement is very simple. We obtain a *hypergraph optimal linear arrangement* (HOLA) from the hypergraph and then use a *space filling curve* [Peano 1890, Sagan 1994] to embed this *linear arrangement* on the grid. Algorithm 5.1 gives the steps of the proposed method. A discrete *space filling curve* provides a linear traversal or indexing of a multi-dimensional grid space, as described in Section 4.4.2.

5.4.2.1 Approximation ratio for general hypergraphs

The approximation algorithm we discuss next needs a lower bound on OPT_{HPG}, the optimal solution for HPG.

Assume that the nets are disjoint. Then, the half-perimeter bounding box of each net i has to be greater than $2\sqrt{n_i}$. This follows from the fact that for a fixed area, say n_i, the perimeter is minimized when both sides are equal (i.e. $= \sqrt{n_i}$). So,

$$\mathrm{OPT}_{HPG} \geq 2\sum_{i=1}^{k}\sqrt{n_i}. \tag{5.3}$$

First consider the approximation algorithm $\mathcal{A}(H)$ for HOLA. As per Theorem 5.1, we have

$$\frac{A(H)}{\mathrm{OPT}_{\mathrm{HOLA}}} \leq c' \; d\sqrt{\log n}\log\log n$$

where $A(H)$ is the cost of the output of $\mathcal{A}(H)$.

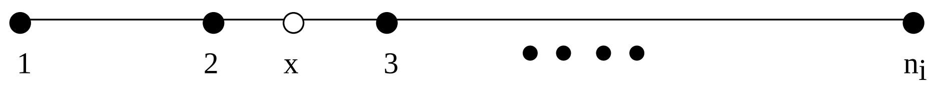


Figure 5.3: Span of a net (hyperedge) with n_i CLBs.

Lemma 5.2 *In the optimal solution for HOLA, the sum of the maximum span of the nets is bounded by $d\sum_{i=1}^{k} n_i$, i.e., $OPT_{HOLA} \leq d\sum_{i=1}^{k} n_i$, where k is the number of nets.*

Proof: As shown in Figure 5.3, consider a vertex $x \notin S_i$ but lying in the span of the net S_i. It is easy to observe that x has to share a net $S_j \neq S_i$ with some vertex $v_i \in S_i$, otherwise $\mathrm{OPT}_{\mathrm{HOLA}}$ can be reduced further. Now, if none of the nets share any vertices, then $\mathrm{OPT}_{\mathrm{HOLA}} = \sum_{i=1}^{k}(n_i - 1)$. However, each vertex of the net S_i in the worst case can belong to different nets S_j $(\neq S_i)$ in addition to S_i. Each S_j can be at most of size d. So, if we expand each net S_i to a span of $d \cdot n_i$ and take the sum, then the sum exceeds $\mathrm{OPT}_{\mathrm{HOLA}}$, making the total $\mathrm{OPT}_{\mathrm{HOLA}} \leq d\ \sum_{i=1}^{k} n_i$, where k is the total number of nets or hyperedges. □

As a consequence of Lemma 5.2 and Theorem 5.1 we have the following upper bound,

$$A(H) \leq c^{'} d^2 \sqrt{\log n} \log\log n \sum_{i=1}^{k} n_i \tag{5.4}$$

Let us now consider a hyperedge (net) S_i that spans a length r_i in the *linear arrangement* corresponding to $A(H)$. Therefore,

$$r_1 + \ldots + r_k \leq c^{'} d^2 \sqrt{\log n} \log\log n \sum_{i=1}^{k} n_i \tag{5.5}$$

Lemma 5.3 *[Sagan 1994, Even 2000] In a 2D embedding using space filling curve, each net spanning a length r_i in the linear arrangement has a perimeter bounded by $C^{'} \cdot \sqrt{r_i}$ where $C^{'}$ is a positive constant.*

Proof: Consider a Hilbert curve of order l [Sagan 1994, Even 2000, Gotsman 1996]. It is easy to observe that the Hilbert curve of the preceding lower order is obtained using a 4-fold reduction. As an example, the lower left 16 cells of $l = 3$ in Figure 4.4(a) are mapped to the lower left 4 cells of $l = 2$. Now, consider a net spanning a length of r_i in the *linear arrangement*, and consider its Hilbert filling. As we go down the order of Hilbert curves applying the 4-fold reduction, the net spanning a length r_i becomes successively smaller till it reaches a size of 4. If x is the number of such 4-fold reduction for a net of length r_i, then $\frac{r_i}{4^x} = 4$. So, $x = O(\log_2 \sqrt{r_i})$.

If we expand again 4-fold, the sides increase exponentially as a power of 2 (vide Figure 4.4(a)). So, the perimeter is bounded by $2^{O(\log_2 \sqrt{r_i})} = O(\sqrt{r_i})$. □

Next, we derive the approximation ratio of our algorithm for the grid embedding of hypergraph.

Theorem 5.3 *HPG is $O(d\sqrt[4]{\log n}\sqrt{k\log\log n})$ approximable.*

Proof: As a consequence of Lemmata 5.2 and 5.3, the upper bound on the half-perimeter bounding box cost is $HPWL \leq C^{'}(\sqrt{r_1} + \ldots + \sqrt{r_k})$.

Cauchy-Schwarz inequality [Abramowitz 1972] states that $\forall x_i, y_i \in I\!R$, $\left(\sum_{i=1}^{k} x_i y_i\right) \leq \sqrt{\left(\sum_{i=1}^{k} x_i^2\right)\left(\sum_{i=1}^{k} y_i^2\right)}$.
By setting $y_i = 1$ and $x_i = \sqrt{r_i}$ in the above, we get $\left(\sum_{i=1}^{k}\sqrt{r_i}\right) \leq \sqrt{k}\sqrt{\left(\sum_{i=1}^{k} r_i\right)}$.
Using the above and Equation 5.5, we arrive at

$$HPWL \leq C^{'}(\sqrt{r_1} + \cdots + \sqrt{r_k}) \leq C^{''}d\;\sqrt{k}\sqrt{\sqrt{\log n}\log\log n\;\sum_{i=1}^{k} n_i} \qquad (5.6)$$

where $C^{'}$ and $C^{''}$ are constants. Finally, using Equations 5.2, 5.3 and 5.6, and the fact that n_i's are all non-zero positive integers, the approximation ratio becomes

$$\begin{aligned}\frac{HPWL}{OPT_{HPG}} &\leq \frac{C^{''}d\;\sqrt{k\;\sqrt{\log n}\log\log n}\sqrt{\sum_{i=1}^{k} n_i}}{2\sum_{i=1}^{k}\sqrt{n_i}}\\ &\leq \frac{C^{''}d\sqrt{k\;\sqrt{\log n}\log\log n}}{2}\left(\frac{\sqrt{\sum_{i=1}^{k} n_i}}{\sum_{i=1}^{k}\sqrt{n_i}}\right)\end{aligned}$$

Thus, $\frac{HPWL}{OPT_{HPG}} \leq O(d\sqrt[4]{\log n}\sqrt{k\log\log n})$ because $\left(\frac{\sqrt{\sum_{i=1}^{k} n_i}}{\sum_{i=1}^{k}\sqrt{n_i}}\right) \leq 1$ with n_i's being all non-zero positive integers. □

The approximation ratio obtained here is dependent on the number of hyperedges/nets. If $k = o(\log^{\frac{3}{2}} n \log\log n)$, then the approximation ratio of Theorem 5.3 is better than that of Theorem 5.2.

5.4.2.2 Approximation ratio for hypergraphs with bounded degree

In the island style FPGAs, each CLB typically has one or two output terminals and a small but constant number (say four) input terminals. An output terminal can have *fanout* to all other CLBs, but it can have *fanin* from at most four CLBs. For example, in the CLB cited above, v_i can belong to at most five hyperedges as v_i can have *fanout* of one hyperedge and *fanin* of four hyperedges. This leads to the following observations.

Observation 5.1 *Each CLB (denoted as a vertex v_i) can belong to at most $O(1)$ hyperedges. This implies $\sum_{i=1}^{k}|S_i| = O(n)$, where $|S_i|$ denotes the number of vertices* *in the hyperedge S_i, k and n being the number of hyperedges and vertices respectively.*

Observation 5.2 *As each vertex belongs to at most $O(1)$ hyperedges, the number of pairs of i, j $(i \neq j)$ such that $S_i \cap S_j \neq \emptyset$ is $O(n)$. Further, $\sum_i \sum_j |S_i \cap S_j| = O(n)$.*

We already know that for an optimal *linear arrangement* of a hypergraph, a vertex $x \notin S_i$ but lying in the span of S_i has to share a net $S_j \neq S_i$ with some vertex $y \in S_i$, otherwise, $\mathrm{OPT}_{\mathrm{HOLA}}$ can be reduced further. The optimal cost of a hypergraph *linear arrangement* is $\sum_{i=1}^{k} \max_{q,l \in S_i}\{|h(q) - h(l)|\}$ where the cost contribution of hyperedge S_i is $\max_{q,l \in S_i}\{|h(q) - h(l)|\}$. Now, $\max_{q,l \in S_i}\{|h(q) - h(l)|\}$ is nothing but the number of vertices in the span of S_i in the *linear arrangement*. Some of these vertices belong to S_i and others do not. So, the cost is the sum of the number of vertices in the span of each S_i. It is non-trivial to derive an upper bound on $\mathrm{OPT}_{\mathrm{HOLA}}$ this way. Instead, we look at the number of hyperedges S_j that pass over v_i and sum it over all i, $1 \leq i \leq n$. This would also give us the same cost. For an amortized counting of the number of hyperedges that pass over v_i, we design a charging scheme as follows.

$c_1(v_i)$: type 1 charge to v_i: $\forall S_i$ such that $v_i \in S_i$, assign a charge of 1 unit to v_i. As v_i belongs to $O(1)$ hyperedges, so the number of such charges $c_1(v_i)$ is $O(1)$. Thus, $\sum_i c_1(v_i) = O(n)$ by Observation 5.1.

$c_2(v_i)$: type 2 charge to v_i: $\forall S_j$ such that $S_i \cap S_j \neq \emptyset$ and $v_i \in S_i$ but $v_i \notin S_j$, assign a charge of 1 unit to v_i. Owing to our characterization of the optimal solution, only these hyperedges S_j need to be charged to v_i. So, $c_2(v_i)$ the charge for this type of hyperedges on v_i is $\sum_j O(1).|S_i \cap S_j|$ as any vertex v ($\neq v_i$ and $v \in S_i$ and S_j) can belong to at most $O(1)$ hyperedges. In the worst case, $c_2(v_i)$ can be $O(n)$. But it can only happen for a constant number of cases because of Observations 5.1 and 5.2. It can also be computed as follows:

$$\sum_{pred(v_i)} (\#\text{ incoming and outgoing hyperedges of pred}(v_i)) +$$
$$\sum_{succ(v_i)} (\#\text{ incoming and outgoing hyperedges of succ}(v_i))$$

Any vertex v_i has bounded indegree and outdegree, i.e., $O(1)$; further while the number of predecessor vertices of any v_i is also $O(1)$, the number of successors is bounded by the cardinality abs(S_i) of its largest outgoing hyperedge, say S_i. This gives us $c_2(v_i) = O(1) \cdot O(1) + O(|S_i|) \cdot O(1) = O(|S_i|)$. So, we have the following for the total charge. The total charge summed over all vertices v_i is $\sum_i (c_1(v_i) + c_2(v_i))$. Now,

$$\begin{aligned}
\sum_i (c_1(v_i) + c_2(v_i)) &= \sum_i (O(1) + \sum_j O(1) \cdot |S_i \cap S_j|) \\
&= \sum_i O(1) + \sum_i \sum_j O(1) \cdot |S_i \cap S_j| \\
&= O(n) \text{ by Observation 5.2}
\end{aligned}$$

Alternately,

$$\begin{aligned}
\sum_i (c_1(v_i) + c_2(v_i)) &= \sum_i (O(1) + O(|S_i|) \\
&= \sum_i O(|S_i|) \\
&= O(n) \text{ by Observation 5.1}
\end{aligned}$$

The above discussion leads to the following lemma which is a special case of Lemma 5.2. Here, we get a reduction of d compared to Lemma 5.2.

Lemma 5.4 *In the optimal solution for HOLA with bounded degree for each vertex, the sum of the maximum span of the nets is bounded by $O(n)$, i.e., $OPT_{HOLA} \leq O(n)$.*

As a consequence of Lemma 5.4, we get the Corollary 5.1 of Theorem 5.3, where there is a reduction by $\sqrt{d}$ in the approximation ratio. The proof of the Corollary works out in the same way as the proof of Theorem 5.3, the only exception being that the proof uses Lemma 5.4 instead of Lemma 5.2.

Corollary 5.1 *HPG for degree bounded hypergraph is $O(\sqrt[4]{\log n}\sqrt{kd}\log\log n)$ approximable.*

Theorem 5.3 or Corollary 5.1 is not implementable straight away as per fast FPGA placement standards because of the time consuming $O(d\sqrt{\log n}\log\log n)$ approximation algorithm of HOLA. Hence, we focus on how we can modify this algorithm in order to make it run very fast.

Algorithm 5.2: SfcPlace: Our efficient placement method.

```
Input  : Hypergraph representing a given netlist of CLBs and IOBs; a 2D
         array
Output: Placement of CLBs and IOBs on a given 2D array
1 Transform the netlist hypergraph to a linear arrangement of CLBs of the
  hypergraph using top-down min-cut graph bi-partitioning technique;
2 Map this linear arrangement obtained directly on to the $\sqrt{n} \times \sqrt{n}$ 2D grid
  using a recursive space filling curve;
3 Place IOBs on the periphery by minimum weighted bipartite matching
  formulation
```

5.5 The algorithm in practice

Our placement technique is the *fine grain* method described in Section 4.4 of Chapter 4. The method consists of the three steps as shown in Algorithm 5.2. We replace the time consuming Step 1 of Algorithm 5.1 as follows. We apply a *min-cut* recursive bi-partitioner to partition the netlist of CLBs such that finally each partition has one element. The left to right order of the leaves of the partition tree is treated as the *linear arrangement* of CLBs. We justify this assumption in Section 5.5.1 below. Step 2 is described in Section 5.5.2. In Step 3, the IOBs are placed on the periphery using a *minimum weighted bi-partite matching* formulation.

5.5.1 Computation of linear arrangement

In the *fine grain* method of Chapter 4, first we generated a one dimensional arrangement of CLBs. This one dimensional arrangement has to be a *linear arrangement* such that the total HPWL cost is minimum, in order for the second step of placement, namely, the usage of *space filling curve* to be meaningful.

The problem of placing the nodes of a graph on a straight line with equal spacing such that the sum of edge lengths of the graph is minimum, is NP-complete [Garey 1979]. But, for some special classes of graphs, i.e., rooted directed trees, undirected trees, and series parallel graphs, this problem can be solved in polynomial time [Adolphson 1973, Shiloach 1979, Nandy 1997]. As a netlist hypergraph does not belong to these special classes, we adopt a heuristic procedure based on balanced min-cut bipartition to generate a *linear arrangement* of the nodes. In this recursive process, at each level we obtain two partitions having almost the same number of nodes, which are heavily connected intra-partition. The recursive partitioning process is represented as a *partition tree*, where the root corresponds to all the nodes in the hypergraph. The left and right child correspond to the two partitions. Without loss of generality, the arrangement of the two partitions in the *partition tree* can be swapped. We adopt the convention that the partition assigned to the left child is the immediate predecessor in the *linear arrangement* of that assigned to the right child. Even et al. [Even 2000] designed a divide-and-conquer based approximation algorithm for optimal *linear arrangement* using the *decomposition tree* of a graph. The root node corresponds to all the vertices of graph and each internal node of the tree corresponds to a partition of the vertices. The tree is fully decomposed when each leaf consists of a single node. The decomposition may be based on any "criteria" that partitions the set of vertices such that "related" vertices belong to the same partition. The authors [Even 2000] established that the partitions induced by a *decomposition tree* gives a qualitatively good *linear arrangement* of the vertices according to their order of appearance as the leaves of the decomposition tree. We use this idea to obtain a *linear arrangement* of blocks using a very fast hypergraph partitioner *hMetis* [hMetis , Karypis 1999a] as already explained in Chapter 4.

The authors of *hMetis* have not reported about the quality of the *linear arrangement* obtained by their method. But, we observed that, as established in [Even 2000], the *linear arrangement* produced by *hMetis* is very good and this can very well help in reducing the final wirelength, and hence the delay. Although the proposed method does not explicitly consider congestion, the recursive min-cut bipartitioning paradigm was chosen for its known byproduct of reducing wide spread congestion, and thereby to enhance success in routing.

5.5.2 Placement by space filling curves

We generate *Hilbert space filling curve* as described in Section 4.4.2, to place the logic blocks on the FPGA array. Essentially, this allocates a specific co-ordinate position for each of the logic blocks in the *linear arrangement*, using the sequence generated by *Hilbert curve*.

5.5.3 Placement of IOBs

After the CLBs are placed on to a 2D FPGA array, the IOBs are to be placed on the periphery of the array. We have formulated this problem as an instance of a *minimum weighted bi-partite matching problem* (MWBM) as explained in Section 4.3.4.

The time complexity of the proposed method is the time complexity of the *Fine grain* method discussed in Chapter 4. Thus, the time complexity is $O(n^{1.25})$ as explained in Section 4.4.4 of Chapter 4.

5.6 Experimental results

In this section, we present the experimental results of our placement methodology and compare these with the placement results produced by popular FPGA tool VPR [Betz 1997]. In order to demonstrate the suitability of our fast placement method in the FPGA CAD flow, we have also routed the placement produced by our method using the router of VPR and compared the critical path delay thus obtained with that of VPR. The platform used is 1.2GHz SunBlade 2000 workstation. The *Library of Efficient Datatypes and Algorithms* (LEDA) [LEDA] is employed to solve the MWBM problem described in Section 5.5.3.

Table 5.1: Characteristics of MCNC FPGA placement benchmark circuits

1	2	3	4	5	6
Ckt	# CLBs (n)	# Inputs	# Outputs	Max. # terminals (d)	Size of 2D grid
tseng	1047	52	122	246	33×33
ex5p	1064	8	63	75	33×33
apex4	1262	9	19	84	36×36
dsip	1370	229	197	450	54×54
misex3	1397	14	14	49	38×38
diffeq	1497	64	39	196	39×39
alu4	1522	14	8	24	40×40
des	1591	256	245	227	63×63
bigkey	1707	229	197	449	54×54
seq	1750	41	35	90	42×42
apex2	1878	38	3	86	44×44
s298	1931	4	6	244	44×44
frisc	3556	20	116	124	60×60
elliptic	3604	131	114	292	61×61
spla	3690	16	46	155	61×61
pdc	4575	16	40	238	68×68
ex1010	4598	10	10	260	68×68
s38417	6406	29	106	236	81×81
s38584.1	6447	38	304	188	81×81
clma	8383	62	82	457	92×92

5.6.1 Quality of placement

For completeness sake of this chapter, we reproduce the characteristics of twenty MCNC benchmark circuits given in Table 4.1 once again in Table 5.1 here. The columns 2 to 5 present the number of CLBs, primary inputs, primary outputs and the maximum degree (d) of a net respectively, in the given circuit. Column 6 gives the minimum square array required to place all CLBs and IOBs.

Table 5.2 shows the quality of placement and the speedup achieved by our method (called SFC in the tables) over the simulated annealing based method of VPR. We report the sum of half-perimeter of the bounding box of all nets, namely *HPWL cost*, by our method in column 2 and the same obtained by VPR in column 3. We ran VPR with all default parameters for *place_ only* option. Assuming the result obtained by VPR to be the optimum, the ratio $\frac{HPWL(SFC)}{HPWL(VPR)}$, is reported in column 6. The time taken in seconds by our method and the placement method of VPR are shown in columns 4 and 5. Column 7 shows the speedup achieved by our method. We observed that the HPWL placement cost of our method is 1.31× of VPR on an average and lies in the range of 0.96 to 1.65 in practice, whereas the theoretical upper bound on the approximation ratio ($O(\sqrt[4]{\log n}\sqrt{kd \log\log n})$) derived by us in Corollary 5.1 is not a constant factor. But, as can be observed from column 6 of Table 5.2, the quality of our practical method is almost independent of n, showing that the placement quality produced by our method is scalable, just as its run-time performance. It is to be noted that the ratio remains almost the same if *BB cost* is computed as reported in Table 4.4 of Chapter 4. The circuit $s298$ has a positive gain in placement cost, possibly because it has very few input and output terminals, and mostly two terminal nets which lead to short, local connections in the *linear arrangement*. The gain in speed is very significant. The speedup is about 33× on an average. Thus, our placement method can serve well in the cases where speed and scalability are of greater importance than optimal quality.

We have also observed the routability of the placement by routing its output using VPR router. The columns 8 and 9 of Table 5.2 report the critical path delay (in 10^{-8} secs.) obtained for our placement method and that for VPR. Our placement is routable with a fixed channel width, and on the average the critical path delay is about 1.97× of that obtained by VPR. To the best of our knowledge, no earlier work has reported the routability of benchmark circuits with deterministic heuristics.

The simulated annealing based methods (say VPR) do not scale up well for large circuits. In order to verify this fact, we carried out another set of experiments on the same set of benchmark circuits. First, we noted the CPU time t taken by our method (as shown in Algorithm 5.2). Next, we tuned VPR with appropriate values for parameters such as *init_ t*, *exit_ t* to let it naturally terminate within time very close to t (with a positive bias in favor of VPR) and noted the *HPWL cost* for each circuit. Let this said *HPWL cost* be denoted as $HPWL(VPR_t)$. The ratios

Table 5.2: Comparison of *HPWL cost*, *speedup* and *critical path delay*

1	2	3	4	5	6	7	8	9	10
	HPWL		CPU time (s)		SFC/VPR		Delay ($10^{-8}s$)		
Ckt	SFC	VPR	SFC	VPR	HPWL	Speedup	SFC	VPR	$\frac{SFC}{VPR}$
tseng	9653	7302	5	152	1.32	30.4	12.6	5.71	2.21
ex5p	17089	13395	5	169	1.27	33.8	14.0	6.55	2.13
apex4	18241	14217	6	222	1.28	37.0	15.6	7.85	1.99
dsip	18037	11666	15	248	1.54	16.5	7.6	6.55	1.16
misex3	17550	13430	7	218	1.30	31.1	12.4	7.53	1.65
diffeq	13714	10913	8	247	1.25	30.8	14.1	6.27	2.25
alu4	16669	12197	8	242	1.30	30.2	14.4	8.02	1.80
des	28288	18441	22	312	1.53	14.2	13.2	9.02	1.46
bigkey	18100	13167	16	331	1.37	20.7	8.4	6.79	1.24
seq	22905	17747	9	312	1.29	34.7	13.2	7.87	1.67
apex2	25527	18816	10	339	1.29	33.9	16.8	10.00	1.68
s298	11207	11635	10	321	0.96	32.1	27.2	13.10	2.07
frisc	40942	40655	23	969	1.00	42.1	35.1	12.70	2.77
elliptic	34492	29760	24	846	1.15	35.2	31.3	10.80	2.90
spla	49545	37006	23	961	1.33	41.8	22.3	12.90	1.74
pdc	74087	55269	31	1280	1.34	41.3	34.7	14.10	2.46
ex1010	71669	43327	32	1161	1.65	36.3	45.8	18.30	2.51
s38417	62404	47179	51	1700	1.32	33.3	20.0	9.40	2.13
s38584.1	64510	44597	52	1930	1.44	37.1	19.2	10.10	1.90
clma	114271	81412	69	3200	1.40	46.4	37.2	22.40	1.66
Avg:					1.31	32.9			1.97

$\frac{HPWL(SFC)}{HPWL(VPR)}$ and $\frac{HPWL(VPR_t)}{HPWL(VPR)}$ are plotted, as shown in Figure 5.4. It can be noted that our method always outperforms VPR.

5.6.2 Effect of low temperature SA

Although it is not the main focus of our work, for completeness sake, a low temperature simulated annealing was applied to the solutions obtained by our Algorithm 5.2 to overcome the local optima and compared the performance of the benchmarks with respect to the *HPWL cost*, *CPU time* and the *critical path delay*.

Table 5.3 reports the *HPWL cost* obtained after execution of the low temperature SA on our SFC based method ($SFC + SA$) and default VPR in columns 2 and 3. The CPU time taken by $SFC+SA$ and VPR are then reported in columns 4 and 5. The last column shows the ratio of CPU time taken by VPR to $SFC + SA$. Thus, $SFC+SA$ is about 2× faster than VPR as opposed to 33× when a post processing

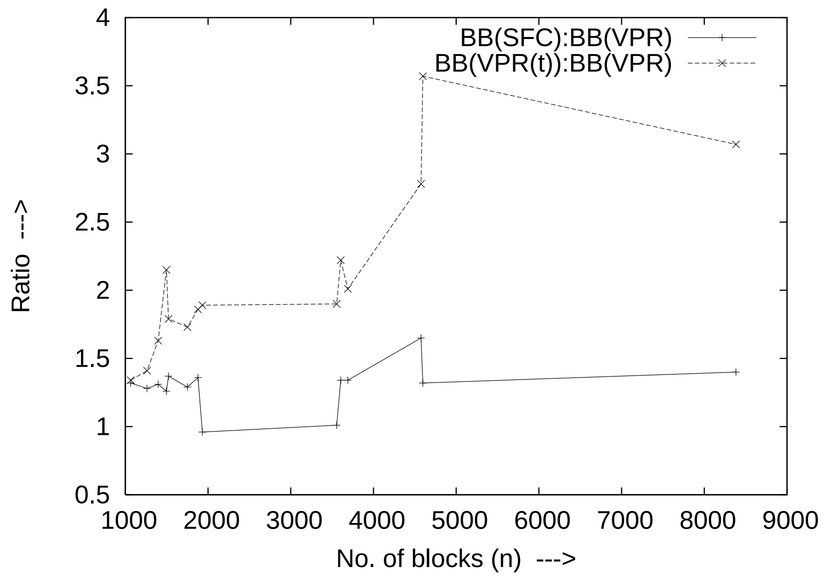


Figure 5.4: Effectiveness of our proposed Algorithm 5.2: comparison of the ratios $\frac{HPWL(SFC)}{HPWL(VPR)}$ and $\frac{HPWL(VPR_t)}{HPWL(VPR)}$, where t is the time taken by Algorithm 5.2, and $HPWL(VPR_t)$ is the *HPWL cost* of the solution produced by VPR run only for time t.

step of low temperature SA is executed along with the proposed deterministic island-style FPGA placement method.

Table 5.4 shows the routability of our method with respect to the critical path delay obtained. In columns 2 and 3 we report the critical path delay $CP(SFC)$ and $CP(SFC{+}SA)$ achieved by Algorithm 5.2 given in Section 5.5 (SFC), and the same method followed by low temperature simulated annealing $SFC + SA$ respectively. The critical path delay obtained by VPR $CP(VPR)$ is reported in column 4. The comparison of the critical path delay achieved by our proposed method, namely SFC, $SFC + SA$ with that of VPR are reported in columns 5 and 6 as ratios $\frac{CP(SFC)}{CP(VPR)}$ and $\frac{CP(SFC+SA)}{CP(VPR)}$ respectively. We observed that, on an average, the critical path delay obtained by SFC and $SFC + SA$ is 1.97× and 1.03× respectively of VPR. As stated earlier, due to execution of SA in $SFC + SA$, we achieve a much lower speedup of 2×, as opposed to 33× for our proposed method SFC (Algorithm 5.2) based on *linear arrangement* followed by *space filling curve*.

5.7 Conclusion

In this chapter, we presented a very simple and effective yet fast placement approach for island-style FPGAs with theoretical bounds on the quality of the so-

Table 5.3: Comparison of *HPWL cost* and *CPU time* for $SFC+SA$ and VPR

1	2	3	4	5	6
	HPWL cost		CPU time(s)		
Ckt	$SFC+SA$	VPR	$SFC+SA$	VPR	$\frac{VPR}{(SFC+SA)}$
tseng	7587	7302	87	152	1.74
ex5p	13265	13395	89	169	1.89
apex4	13470	14217	101	222	2.19
dsip	14589	11666	176	248	1.40
misex3	13020	13430	113	218	1.92
diffeq	12271	10913	122	247	2.02
alu4	12402	12197	121	242	2.00
des	18708	18441	187	312	1.66
bigkey	15612	13167	200	331	1.65
seq	17291	17747	160	312	1.95
apex2	18813	18816	167	339	2.02
s298	11812	11635	169	321	1.89
frisc	40476	40655	451	969	2.14
elliptic	32847	29760	474	846	1.78
spla	38836	37006	494	961	1.94
pdc	56319	55269	645	1280	1.98
ex1010	43409	43327	624	1161	1.86
s38417	50202	47179	985	1700	1.72
s38584.1	47047	44597	928	1930	2.07
clma	84214	81412	1337	3200	2.39
Avg:					1.91

lution. First we extended the theoretical results for *graph optimal linear arrangement* (GOLA) and *graph embedding on grid* (GPG) to *optimal linear arrangement* (HOLA) and *embedding* (HPG) for hypergraphs respectively. Next, we designed an $O(\sqrt[4]{\log n}\sqrt{kd}\log\log n)$ approximation algorithm. It is needless to say that this bound automatically improves along with tighter approximation bound on $GOLA$, as it is derived from that for $GOLA$. Our algorithm is easy to implement; it uses only an approximate *linear arrangement* and a recursive *space filling curve*. The theoretical algorithm proposed by us works for real-life benchmark circuits with practical implementations. The running time is near-linear in the number of CLBs and hence is scalable for large circuits. As per our knowledge, this is the first attempt in bringing into practice a method for placement that has theoretical bounds of approximation. Applying our method to a set of benchmark circuits, we observed that on the average, the quality of our solution is 1.31× of the popular simulated annealing based tool VPR while the speedup is 33×. The quality of solutions, as

Table 5.4: Comparison of *critical path delay*

1	2	3	4	5	6
	Critical Path(CP)(10^{-8} s)			Comparison of CP	
Ckt	$CP(SFC)$	$CP(SFC+SA)$	$CP(VPR)$	$\frac{CP(SFC)}{CP(VPR)}$	$\frac{CP(SFC+SA)}{CP(VPR)}$
tseng	12.6	5.83	5.71	2.21	1.02
ex5p	14.0	7.00	6.55	2.13	1.07
apex4	15.6	8.22	7.85	1.99	1.05
dsip	7.6	6.79	6.55	1.16	1.04
misex3	12.4	7.99	7.53	1.65	1.06
diffeq	14.1	6.84	6.27	2.25	1.09
alu4	14.4	7.96	8.02	1.80	0.99
des	13.2	9.32	9.02	1.46	1.03
bigkey	8.4	6.56	6.79	1.24	0.97
seq	13.2	8.95	7.87	1.67	1.14
apex2	16.8	10.10	10.00	1.68	1.00
s298	27.2	13.60	13.10	2.07	1.04
frisc	35.1	13.30	12.70	2.77	1.05
elliptic	31.3	11.70	10.80	2.90	1.08
spla	22.3	13.50	12.90	1.74	1.05
pdc	34.7	15.00	14.10	2.46	1.06
ex1010	45.8	16.50	18.30	2.51	0.90
s38417	20.0	11.10	9.40	2.13	1.18
s38584.1	19.2	9.41	10.10	1.90	0.93
clma	37.2	20.00	22.40	1.66	0.89
Avg:				1.97	1.03

observed from experiments on benchmark circuits, stays within a constant range and do not depend on the number of CLBs. The placements obtained are routable with fixed channel width. This justifies the applicability of our method for fast FPGA placement.

Chapter 6

Unified Floorplan Topology Generation and Sizing on Heterogeneous FPGAs

Contents

6.1 Introduction

Modern FPGA architectures are significantly different from those that were available in the last decade. Earlier, CLBs were a *homogeneous* resource and arranged in rows and columns uniformly, with *primary input and output blocks* (IOB) on the periphery of the chip. Recent FPGA architecture comprises not only the CLBs and IOBs, but also Multipliers (MUL), Block RAMs, DSP and microprocessor cores. Few columns of RAM-MUL pairs and even IOB are interspersed among CLB columns. Moreover, a large design with millions of gates is partitioned into a smaller number of functional modules to reduce the place-and-route time, and to achieve better quality of solution. This necessitates a floorplanning step for hierarchical designs in the physical design flow of FPGAs. Although a large volume of work on ASIC floorplanning exists, these are generally not employed while mapping designs even on to the earlier sea-of-gates style FPGAs. Hence there is a pressing need for fast floorplanning techniques that consider the *heterogeneous* logic and routing resources of modern FPGAs.

The literature on FPGA floorplanning (both *homogeneous* and *heterogeneous*) is merely a handful. Moreover, most of them are SA based and hence has large run-time. Unlike SA based methods [Cheng 2004], [Feng 2006], and [Singhal 2007b], we propose a deterministic method like [Yuan 2005], for unified floorplan topology generation and *sizing* to place the netlist of soft modules on to a target FPGA architecture with pre-placed *heterogeneous* resources. The proposed method *HeteroFloorplan* is further supported by various experimental results and theoretical analysis. The experimental results indicate that *HeteroFloorplan* is fast and can produce floorplans with improved *half-perimeter wirelength* (HPWL) when compared to very few existing methods.

In this chapter, first we briefly describe the *heterogeneous* FPGA architecture, define the floorplanning problem for such FPGAs, and give an outline of the proposed three phase floorplanning method *HeteroFloorplan* in Section 6.2. The three phases of *HeteroFloorplan* are detailed in Sections 6.3, 6.4 and 6.5 respectively. This method is then illustrated with an example in Section 6.6. Experimental results are reported in Section 6.7. Section 6.8 validates the greedy allocation of rectangular regions (GARR) of Phase III by comparing it with a network flow formulation for the floorplanning problem. Finally, the concluding remarks appear in Section 6.9.

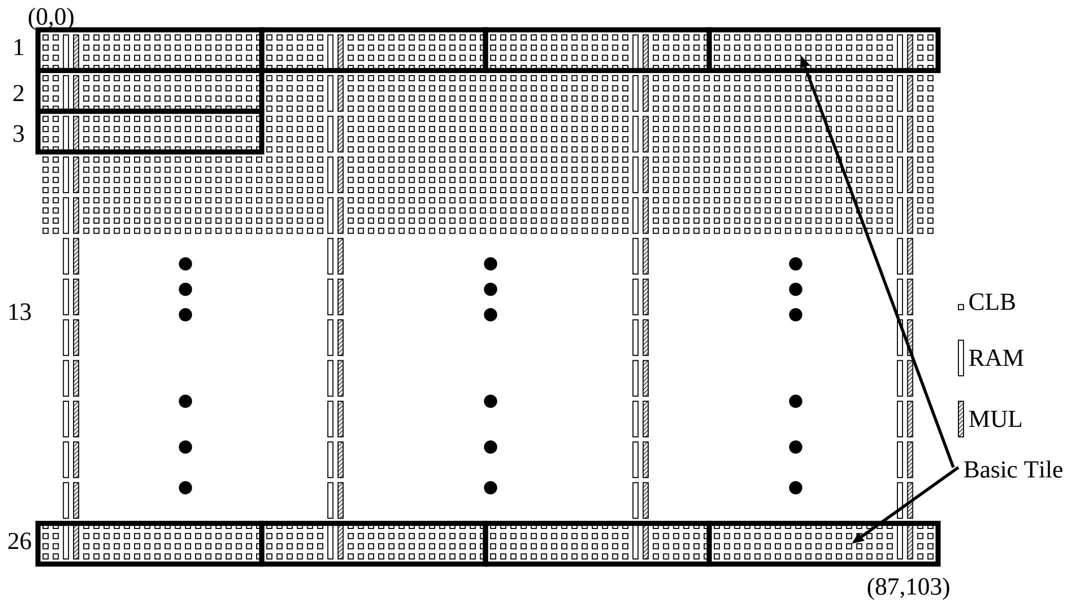


Figure 6.1: Spartan-3 XC3S5000 FPGA architecture, tessellated with *basic tiles* indicated by thick-lined rectangles, each having 4 rows × 20 columns of CLBs and 1 pair of RAM-MUL blocks; $(0, 0, W, H) = (0, 0, 87, 103)$.

6.2 Background

6.2.1 Architecture

In modern FPGAs, CLBs and routing resources are arranged in rows and columns as before, but there are also other types of resources placed in certain patterns, to satisfy a wider range of design requirements. Figure 6.1 shows a Xilinx Spartan-3 [Xilinx] FPGA where the CLBs are arranged in columns interleaved with columns of RAM-MUL pair at certain intervals. Each small square represents a CLB. A RAM block paired with a MUL block, and spanning a height of four rows of CLBs, is indicated by an empty and a shaded rectangle respectively. Henceforth, we assume this architecture for this chapter as well as the next one, although the methodology is applicable to other similar ones presented in Chapter 1.

6.2.2 FPGA floorplanning problem

First, the basic terminology is given below.

Definition 6.1 (Modules and Signal nets:) *Let $M = \{m_1, m_2, \ldots, m_n\}$ be a set of n distinct modules. Let $S = \{S_1, S_2, \ldots, S_q\}$ be a set of q signal nets. Each signal net $S_i \in S$ is associated with a set of distinct modules $M_{S_i} = \{m_j \mid m_j \in M\}$,* *and the set S is called a netlist. If $M_{S_i} = M_{S_j}$, then the two distinct signal nets S_i and S_j connect the same set of modules.*

Essentially, the set of CLBs C, defined in Problem 3.1 of Chapter 3 is now a set of modules M, where each *module* has many CLBs along with RAMs and MULs.

Definition 6.2 (Resource requirement vector [Cheng 2004]:) *For a module m, a 3-tuple vector $\rho_m = (m^{clb}, m^{ram}, m^{mul})$ represents the number of CLBs, RAMs and MULs required by module m.*

Definition 6.3 (Target architecture:) *Let W and H be the width and height of a target FPGA architecture (also referred as chip), where the units are the width and height of a CLB respectively. A coordinate system $(0, 0, W, H)$ with top-left corner at $(0, 0)$ and bottom-right corner at (W, H), is assumed for the given chip.*

In Figure 6.1, it is $(0, 0, 87, 103)$. Each resource on the architecture is identified by its coordinate position (x, y), where $0 \leq x \leq W$ and $0 \leq y \leq H$.

Problem 6.1 (FPGA Floorplanning) *Given*

- *a target architecture$(0, 0, W, H)$ with its resource locations,*
- *a digital circuit design $\mathcal{D}$ consisting of*
  - *a set of soft (flexible in shape) modules M,*
  - *the resource requirement vectors ρ_{m_i} for each $m_i \in M$,*
  - *the netlist S,*

find a floorplan by assigning a connected region $R_i = (x_i^{min}, y_i^{min}, x_i^{max}, y_i^{max})$ to each module m_i on the target architecture having an optimal value of a certain cost function, such that

(i) $0 \leq x_i^{min} \leq x_i^{max} \leq W$ *and* $0 \leq y_i^{min} \leq y_i^{max} \leq H$,

(ii) region for no two modules overlap with each other,

(iii) resource requirement ρ_{m_i} is satisfied within its region R_i

A floorplan is said to be *feasible* if it satisfies the three conditions (i), (ii) and (iii) listed above. The cost function to be optimized is typically the wire-length [Kahng 2000, Roy 2006] for which the popular metric HPWL (half-perimeter wirelength), i.e., the sum of the semi-perimeter of the bounding box of each net, is chosen as in most of the prior works on FPGA *heterogeneous* floorplanning [Feng 2006, Cheng 2006]. The net terminals on a soft module are assumed to be at the center of the module in the absence of information on the position of its terminals. The problem formulation as stated above is a generalization of that given in [Kahng 2000, Feng 2006, Feng 2004], and as such is NP-hard.

6.2.3 Proposed floorplanning method

In our work, we begin from a point where neither the *slicing topology* nor the shape of the modules are given; only the netlist and the resource requirements after technology mapping are known. Our floorplanning methodology *HeteroFloorplan* consists of three phases, namely,

- construction of a recursive *partition tree* as a *template* for possible *slicing topologies*,
- generation of *slicing topologies* with *sizing* [Sarrafzadeh 1996, Dasgupta 1998]
- feasible realization of the topology on a given FPGA architecture.

A flowchart of our proposed method *HeteroFloorplan* is given in Figure 6.2.

The target FPGA architecture is represented as a $2D$ array of rectangular *basic tiles* (defined below), each consisting of a predefined number of each type of resources. The *resource requirements* of a module is converted accordingly in terms of the number of *basic tiles* required. It is possible that, the size of a *basic tile* is too large for a module requiring very few *resources.* Allocation of an entire *basic tile* to such a module results in fragmented under-utilized resources, which may be required elsewhere. For a design with many small modules, we employ a clustering step to pack as many small modules as possible in a *basic tile.* In the first phase, we recursively bi-partition the netlist in a balanced manner to obtain a binary *partition tree* with modules and clusters at the leaves. This *partition tree* is used as a *template* for *slicing topology* generation in the next phase.

In the second phase, for each module, we generate a set of possible rectangular shapes in terms of *tiles* satisfying its *resource requirements.* The *template partition tree* is traversed bottom-up, exploring both *horizontal* and *vertical* cut directions at the internal nodes. This combined *sizing* leads to generation of a list of candidate floorplan *slicing topologies*, called *slicing trees* [Sarrafzadeh 1996], in polynomial time [Stockmeyer 1983].

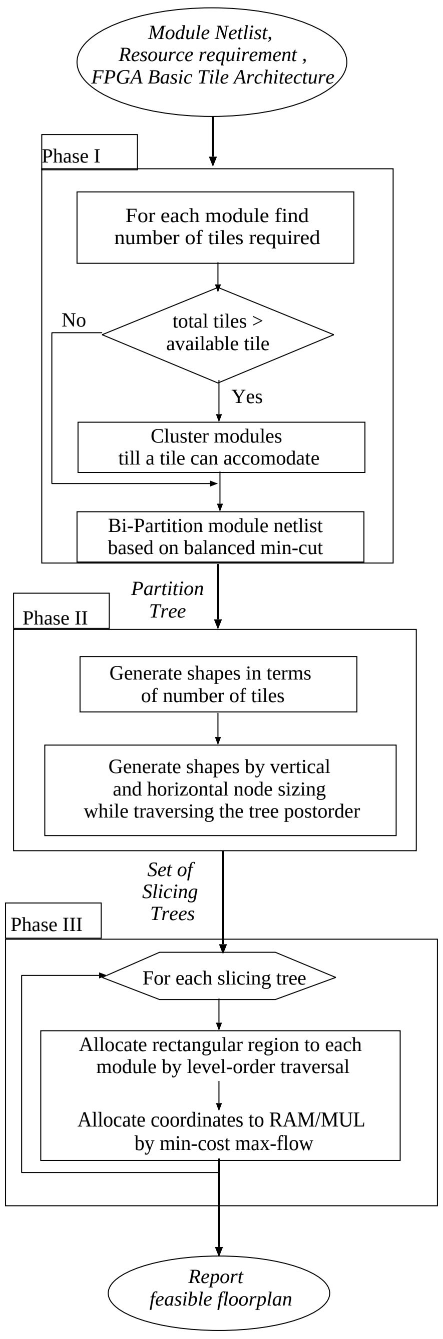

Module Netlist,
Resource requirement ,
FPGA Basic Tile Architecture
Phase I
For each module find number of tiles required
No
total tiles > available tile
Yes
Cluster modules till a tile can accomodate
Bi-Partition module netlist based on balanced min-cut
Partition Tree
Phase II
Generate shapes in terms of number of tiles
Generate shapes by vertical and horizontal node sizing while traversing the tree postorder
Set of Slicing Trees
Phase III
For each slicing tree
Allocate rectangular region to each module by level-order traversal
Allocate coordinates to RAM/MUL by min-cost max-flow
Report feasible floorplan

Figure 6.2: Flow of our floorplanning method *HeteroFloorplan*.

In the third phase, for each *slicing topology* obtained in the previous phase, a rectangular region within the target boundary $(0, 0, W, H)$ is assigned to each module which respects the cut direction and the *resource requirements* in a top-down manner. Finally, among all the *feasible* floorplans obtained, the realization with acceptable aspect ratios of modules and the best HPWL is reported.

The notion of a *basic tile* relevant to our method, is discussed next, followed by a related pre-processing step for clustering of small modules.

6.2.4 Basic tile of a FPGA architecture

Definition 6.4 (Basic tile:) *A basic tile A of a given FPGA architecture is an indivisible contiguous unit having a fixed rectangular shape and containing minimum number of each type of resource, such that the architecture can be represented as a 2D array of these units.*

Although the number of each type of resource in any *basic tile* remains constant, the relative positions of these resources may vary within the rectangular boundary of the *tile* depending on its indices in the 2D array of *basic tiles.* Let the given architecture be thus composed of $(W_t * H_t)$ *basic tiles*, arranged in H_t rows and W_t columns. In the first two phases, we represent the FPGA by $(W_t,\ H_t)$. For the typical architectures with CLBs, RAMs and MULs, let us denote a *basic tile* by a 3-tuple vector $A = (a^{clb},\ a^{ram},\ a^{mul})$. In Figure 6.1, the *basic tile* $A = (80,\ 1,\ 1)$ consists of $20 * 4$ CLBs placed in 20 columns and 4 rows, and a pair of 1 RAM and 1 MUL, placed in two adjacent columns. A *basic tile A* is indicated by a thick-lined rectangle in Figure 6.1. The entire architecture (Spartan-3) shown in Figure 6.1 can be covered by $(\frac{100}{4} =)$ 26 rows and 4 columns of the *basic tile A*, i.e., $(W_t,\ H_t) = (4,\ 26)$.

Assuming the RAMs and MULs to occur in pairs, let there be r columns of such pairs of RAM/MUL in a given FPGA. Let x_i $(i = 2 \ldots r)$ be the number of CLB columns between the $(i-1)^{th}$ and i^{th} column of RAM/MUL. There may be x_1 and x_{r+1} number of CLB columns respectively to the left of the first pair and to the right of the r^{th} pair of RAM/MUL columns. The width W of the FPGA can be expressed as

$$W = 2r + \sum_{i=1}^{r+1} x_i \tag{6.1}$$

Therefore, the width w_A of a *basic tile* is given by $w_A \simeq \frac{W}{r}$, with one RAM-MUL pair in a *tile.* The two columns of RAM and MUL are consecutive but their indices may vary within the width of a *tile.*

The maximum number of CLB columns between two consecutive pairs of RAM/MUL columns on the target chip is $\max_{i=2}^{r}\{x_i\}$. This situation can occur if for two *horizontally* adjacent *tiles*, the RAM/MUL columns for the left *tile* is located at its left-most, and that for the right *tile* is located at its right-most positions, as shown

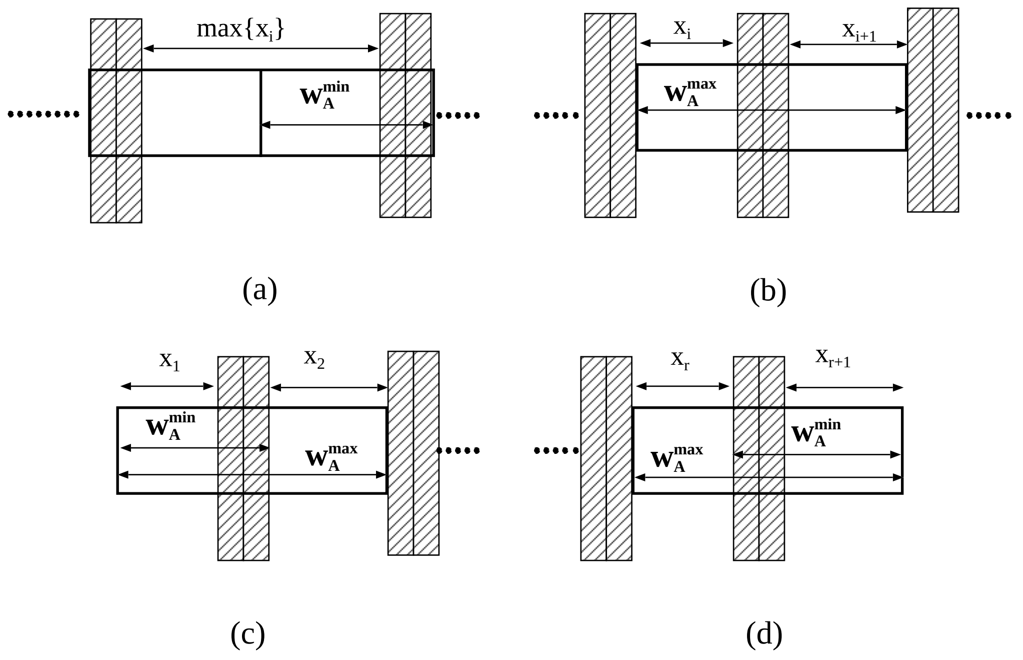


Figure 6.3: The width w_A of a *basic tile*: (a) lower bound, (b) upper bound, (c) for the CLB columns to the left of the left-most RAM/MUL column pair, (d) for the CLB columns to the right of the right-most RAM/MUL column pair.

in Figure 6.3(a). So, we have two inequalities,

$$\begin{aligned} \max_{i=2}^{r}\{x_i\} &\leq 2(w_A) - 4; \text{with 4 RAM-MUL columns at two ends} \\ or, \quad \frac{\max_{i=2}^{r}\{x_i\}}{2} + 2 &\leq w_A \end{aligned}$$

For the left-most CLB columns x_1 and the right-most CLB columns x_{r+1}, we have

$$\begin{aligned} x_1 + 2 \leq w_A \\ x_{r+1} + 2 \leq w_A \end{aligned} \tag{6.2}$$

as shown in Figures 6.3(c) and (d) respectively.

Since a *basic tile* has only one pair of RAM/MUL columns, the width of the *basic tile* w_A cannot exceed the number of columns between the $(i-1)^{th}$ and $(i+1)^{th}$ RAM/MUL pairs, for $i = 2, \ldots, r$. From Figure 6.3(b), we have

$$\min_{i=1}^{r}\{x_i + x_{i+1} + 2\} \quad \geq \quad w_A$$

Combining all the above inequalities, we have

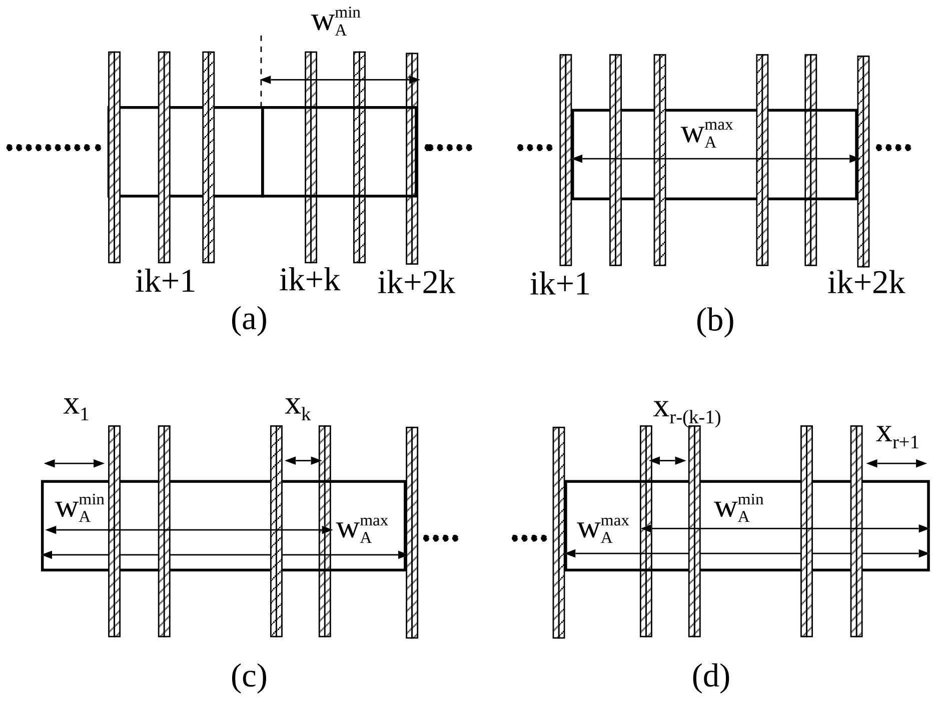


Figure 6.4: The width w_A of a *basic tile* for $\gamma > 1$: (a) lower bound, (b) upper bound, (c) for the left-most CLB columns (d) for the right-most CLB columns.

$$\max_{i=2}^{r}\{x_1, \frac{x_i}{2}, x_{r+1}\} + 2 \leq w_A \leq \min_{i=1}^{r}\{x_i + x_{i+1} + 2\} \tag{6.3}$$

This formulation can be generalized for the case where it may be necessary to have more than one pair of RAM-MUL columns per *tile* as $w_A \simeq \frac{\gamma W}{r}$, where γ (> 1) is the number of RAM-MUL pairs in a *tile*. Figure 6.4(a) shows the condition for lower bound on w_A, where the first RAM/MUL column for the left *tile* is located at its left-most, and that for the last RAM/MUL column of the right *tile* is located at its right-most positions in two *horizontally* adjacent *tiles*. Figures 6.4(c) and (d) depict the case for the left-most and the right-most *tiles* on the chip. Thus we have three equations corresponding to the three conditions shown in Figures 6.4(a), (c) and (d).

$$\max_{i=1}^{\frac{r}{\gamma}-1}\{\sum_{j=2}^{2\gamma} x_{(i-1)\gamma+j}\} \leq 2 \cdot w_A - 2 \cdot 2\gamma$$

$$or, \max_{i=1}^{\frac{r}{\gamma}-1}\{\frac{\sum_{j=2}^{2\gamma} x_{(i-1)\gamma+j}}{2}\} + 2\gamma \leq w_A \tag{6.4}$$

$$\sum_{j=1}^{\gamma} x_j + 2\gamma \leq w_A \tag{6.5}$$

$$\sum_{j=r-\gamma+2}^{r+1} x_j + 2\gamma \leq w_A \tag{6.6}$$

Since the *basic tile* has only γ pairs of RAM-MUL columns, the width of the *basic tile* w_A cannot exceed the number of columns between the $(i \cdot \gamma^{th}$ and $(i+1) \cdot \gamma^{th}$ RAM-MUL pairs, for $i = 1, \ldots, \frac{r}{\gamma}$ as shown in Figure 6.4(b). This gives us,

$$\min_{i=1}^{\frac{r}{\gamma}}\{\sum_{j=1}^{\gamma+1} x_{(i-1)\gamma+j} + 2\gamma\} \geq w_A \tag{6.7}$$

Combining the inequalities 6.6 to 6.7 above for $\gamma > 1$, we have

$$\begin{gathered}\max_{i=1}^{\frac{r}{\gamma}-1}\{\textstyle\sum_{j=1}^{\gamma} x_j, \frac{\sum_{j=2}^{2\gamma} x_{(i-1)\gamma+j}}{2}, \sum_{j=r-\gamma+2}^{r+1} x_j\} + 2\gamma \leq w_A, \\ w_A \leq \min_{i=1}^{\frac{r}{\gamma}}\{\textstyle\sum_{j=1}^{\gamma+1} x_{(i-1)\gamma+j} + 2\gamma\}\end{gathered} \tag{6.8}$$

For any FPGA architecture satisfying Equation 6.8, we can have a basic tile of width w_A. The Spartan-3 board shown in Figure 6.1 satisfies inequality 6.3. Thus, the width of the *basic tile* is $\frac{88}{4} = 22$. The *basic tile* of width 22 consists of 20 columns of CLBs, 1 RAM and 1 MUL.

If it is not possible to derive such a *basic tile* tessellating the entire chip, a tile that covers at least all the minority resources like RAM-MUL uniformly, and most of the majority resource like CLBs, can be computed. There may be a few fractional tiles, consisting of fewer CLBs. Our floorplanning method can still be used for architectures requiring fractional tiles, as Phase III of our method takes care of this situation. For the target FPGA architecture used in this chapter, *fractional tiles* are not required.

Definition 6.5 (Tile requirement:) *For a given basic tile A of target FPGA architecture, the tile requirement T_m of a module m with resource requirement vector ρ_m, is the minimum number of basic tiles satisfying ρ_m. This is given by*

$$T_m = \lceil max(\frac{m^{clb}}{a^{clb}}, \frac{m^{ram}}{a^{ram}}, \frac{m^{mul}}{a^{mul}})\rceil \tag{6.9}$$

Let $T_{tot} = \sum_{i=1}^{n} T_{m_i}$ be the total number of tiles required by all the modules. It may happen that in terms of tiles $T_{tot} > (W_t * H_t)$, yet the sum of *resource requirements* in terms of CLB, RAM, MUL is less than the total resources available on the chip. This occurs if there are many small modules with *resource requirement* much less than a *basic tile*. For such circuits, we employ a greedy pre-processing step of clustering. This facilitates the third phase of *HeteroFloorplan* to generate a *feasible* floorplan of a connected region for each module satisfying the respective *resource requirements*.

6.2.5 Clustering step for large number of small modules

In a given design $\mathcal{D}$, the set of small modules $M_{small} \subseteq M$, each having requirement of one or zero RAM and/or MUL, and m^{clb} less than or equal to a^{clb} in a *basic tile* are clustered by modeling M_{small} along with their signal nets as a graph $G_c = (V_c, E_c)$. Each vertex $v_i \in V_c$ corresponds to a small module $m_i \in M_{small}$. Each net $S_i \in S$ of $\mathcal{D}$ is modeled as a clique of vertices that constitute the net. Thus, an edge $e = (v_i, v_j)$ is in E_c if the vertices v_i and v_j in V_c belong to M_{S_k} for net $S_k \in S$.

First, the elements of M_{small} are sorted in ascending order of their CLB requirements. In each iteration, the smallest module m_c is chosen as seed from this sorted list. A new cluster is grown around m_c by packing as many adjacent modules from the graph G_c as possible into a *basic tile*, in increasing order of their CLB requirements. The modules included in this new cluster are deleted from the sorted list of modules, and the new cluster is inserted into the sorted list according to its size. This process is repeated until no more packing is possible based on net connectivity. Then we pack the rest of the modules/clusters by *best-fit bin packing* strategy, minimizing the number of bins. Thus we arrive at a new reduced netlist of modules/clusters which has to be floorplanned on the given chip (W_t, H_t). The time complexity of this greedy clustering method is $O(n_s^2)$, where $n_s \leq n$, is the number of small modules in M_{small}.

6.3 Phase I: Generation of partition tree

In order to bring connected modules closer such that the wirelength is minimized in the feasible floorplan, we obtain a *linear arrangement* of modules similar to the *linear arrangement* of CLBs described in Chapters 4 and 5. Even et al. [Even 2000] established the fact that, left to right ordering of the leaves of a *decomposition tree* $\mathcal{B}$ is a good estimate of an *optimal linear arrangement*. Hence, we use the state-of-the-art hypergraph partitioning tool *hMetis* [hMetis , Karypis 1999b] to obtain a *partition tree* $\mathcal{B}$. The *linear arrangement* of modules can very well reduce the final wirelength and hence, the delay. Moreover, the min-cut based balanced partitioner helps in distributing the modules evenly on the FPGA board on the basis of RAM-MUL. The partitions are also weight balanced across the cut edges according to the *tile requirements* of each module, such that less white spaces are generated during node *sizing* of almost equal sized modules in Phase II. This partitioning yields the relative ordering of the circuit modules which is the *template* for generating a set of slicing floorplan topologies.

The input to the *hMetis* tool is a netlist, which is a hypergraph $H = (V, E)$. Each vertex $v \in V$ corresponds to a module m_i, $i = 1, 2, \ldots, n$. A hyperedge $e_i = \{v_1, v_2, ...\} \in E$ corresponds to the modules in M_{S_i} for $S_i \in S$. The weight of e_i is the number of nets associated with the modules in M_{S_i}. The weight of a vertex $v \in V$ is T_{m_i} for module m_i corresponding to v_i. The hypergraph H thus generated, is bi-partitioned recursively into n parts, generating a binary *partition tree* or a *decomposition tree* $\mathcal{B}$ with its leaves corresponding to n modules and/or clusters.

6.4 Phase II: Floorplan topology generation

In this step of unified topology generation and *sizing*, a set of sliceable floorplan topologies (i.e., *slicing trees*) is generated by appropriate *horizontal* and *vertical* node *sizing* of a set of possible shapes (in terms of *basic tiles*) for each module. While the work of Otten and Stockmeyer [Otten 1982, Stockmeyer 1983] assumes a *slicing topology* to be given, and finds the shape of each module for an area optimal floorplan, in our work neither the *slicing topology* nor the shape of the modules are given – only the *netlist* S, the *resource requirements* $\rho_{m_i}, i = 1 \cdots n$, and the *partition tree* $\mathcal{B}$ are known.

6.4.1 Generation of module shapes

Definition 6.6 (Irreducible list of shapes:) *A list $D = \{(w_1,\ h_1),\ (w_2,\ h_2),\ \ldots\ (w_t,\ h_t)\}$ of irredundant shapes of a module m, is a list of t possible shapes of m, where $(w_i,\ h_i)$ denotes the width and height of the i^{th} shape of m in terms of basic tiles. D is said to be irredundant if each individual w_i and h_i are distinct [Sarrafzadeh 1996], and, for any $i \neq j, w_i < w_j \iff h_i > h_j$.*

By making individual w_i and h_i distinct as in Definition 6.6, a shape with smaller height, and hence a better shape is chosen from the two implementations with the same width. Given the maximum aspect ratio α_{max}, a set of possible irredundant rectangular shapes with aspect ratio $1 \leq \alpha \leq \alpha_{max}$ for m_i is created with all possible pairs of integers whose product is T_{m_i}. If T_{m_i} is a prime, we consider the shapes corresponding to $(T_{m_i}+1)$ to generate a few extra shapes. As the size of each cluster (discussed in Section 6.2.5) is a single tile, only one shape $(w, h) = (1, 1)$, is possible. For each module m_j, $j = 1, 2, \ldots, n$, we generate a set of t_j possible irredundant shapes $D_j = \{(w_1, h_1),\ (w_2, h_2),\ \ldots,\ (w_i, h_i),\ \ldots\ (w_{t_j}, h_{t_j})\}$. The time complexity for generating all distinct shapes is given by the following lemma.

Lemma 6.1 *If $\kappa = max\{t_j\}$ is the maximum number of shapes generated for any module m_j, then it requires $O(n\kappa)$ time to find the values of all the shapes for n modules.*

Proof: Obvious. □

Lemma 6.2 *The upper bound on the value of κ is given by $O((\log \nu)^{\log \nu})$, where ν denotes the maximum number of basic tiles that a module may require.*

Proof: Let ν denote the maximum number of *basic tiles* that a module may require. This implies that if each of $(n-1)$ modules has only one tile, then the remaining one module can have at most $\nu = \lceil \frac{W \times H}{c} \rceil - (n-1)$ tiles, where c is the size of a *basic tile A* defined for a given floorplan problem.

Let the size (number of CLBs) of the basic tile be c. So, the number of *basic tiles* being targeted on to the $W * H$ chip is $\lceil \frac{W*H}{c} \rceil$. Thus, ν the maximum number of tiles that a rectangular module can take is $\lceil \frac{W \times H}{c} \rceil - (n-1)$, where n is the number of modules in the floorplan.

The maximum number of rectangular shapes κ that can be generated for such a module is equal to the number of distinct factor pairs of ν. From basic number theory [Hardy 1979, Niven 1991], any natural number ν has a unique prime factorization given by

$$\nu = f_1^{r_1}.f_2^{r_2}.\ldots f_\mu^{r_\mu} \tag{6.10}$$

where each f_i is a prime with $f_1 < f_2 < \ldots < f_\mu$. Then, η the number of all possible integer factors of ν is given by

$$\eta = (r_1 + 1).(r_2 + 1).\ldots(r_\mu + 1). \tag{6.11}$$

By taking logarithm of both sides of Equation 6.10, we get

$$\log \nu = \sum_{i=1}^{\mu} r_i \log f_i \tag{6.12}$$

As each r_i and f_i are positive, for all i, we have $1 \leq r_i \leq O(\log \nu)$, i.e, $r_i \leq O(\log \nu)$. So, Equation 6.11 becomes $\eta = O((\log \nu)^\mu)$. Moreover, both sides of Equation 6.12 tally only if the number of terms on the right hand side, i.e., the number of primes $\mu \leq O(\log \nu)$. Hence, the upper bound on the number of factor pairs of ν, and therefore κ is given by $O((\log \nu)^{\log \nu})$. □

Typically the value of κ is very small, ranging between 6 to 10 and hence practical for the proposed method.

6.4.2 Generation of slicing trees

By an extension of the node *sizing* algorithm of [Otten 1982, Stockmeyer 1983], a set of *slicing trees* is next derived at every node p of $\mathcal{B}$, the *partition tree* obtained in Phase I. Let the list of irredundant shapes, each corresponding to a sub-floorplan, at the left (l) child of p be denoted by

$$D_l = \{(w_{l_1}, h_{l_1}), (w_{l_2}, h_{l_2}), \dots, (w_{l_s}, h_{l_s})\}$$

and at the right (r) child of p be

$$D_r = \{(w_{r_1}, h_{r_1}), (w_{r_2}, h_{r_2}), \dots, (w_{r_t}, h_{r_t})\}$$

If the node l or r is a leaf corresponding to a module m_j, then D_l or D_r is D_j itself. The shapes of a subfloorplan at node p are $(w_{l_l} + w_{r_r}, max\{h_{l_l}, h_{r_r}\})$ and $(max\{w_{l_l}, w_{r_r}\}, h_{l_l} + h_{r_r})$ when the left and right subfloorplans are abutted *vertically* and *horizontally* respectively [Sarrafzadeh 1996]. D_l and D_r are sorted in increasing order of width when shapes from D_l and D_r are abutted *vertically*, and, in increasing order of height when abutted *horizontally*. Two lists V^p and H^p at every node p (excluding leaf nodes) are generated by abutting the i^{th} member (w_i, h_i) of D_l with the j^{th} member (w_j, h_j) of D_r *vertically* and *horizontally* respectively, using the node *sizing* algorithms [Sarrafzadeh 1996, Stockmeyer 1983]. The cardinalities of both the irredundant lists V^p and H^p at node p are at most $s + t - 1$ [Sarrafzadeh 1996, Stockmeyer 1983], where $s = |D_l|$ and $t = |D_r|$.

Next, a combined list M^p of irredundant shapes at node p is computed by merging V^p and H^p such that the widths are in strictly increasing order. If the same shape (w, h) is generated in both V^p and H^p, typically the *horizontal* cut is chosen to preserve the contiguity of the RAM-MULs within the region allocated to a module. The algorithm for node *sizing* [Stockmeyer 1983, Sarrafzadeh 1996] to generate irredundant shapes at a node assumed a given *slicing topology*, whereas, our method *HeteroFloorplan* generates not a single, but a set of *slicing topologies* as we traverse the *partition tree* in *post-order* by considering cuts in both directions.

Lemma 6.3 *For a node p, if s and t are the respective cardinalities of the lists D_l and D_r of its left and right subfloorplans, then the number of shapes in M^p is no more than* $2(s + t - 1)$.

Proof: The cardinalities of both the irredundant lists V^p and H^p at node p are at most $s + t - 1$ [Sarrafzadeh 1996, Stockmeyer 1983], where $s = |D_l|$ and $t = |D_r|$.

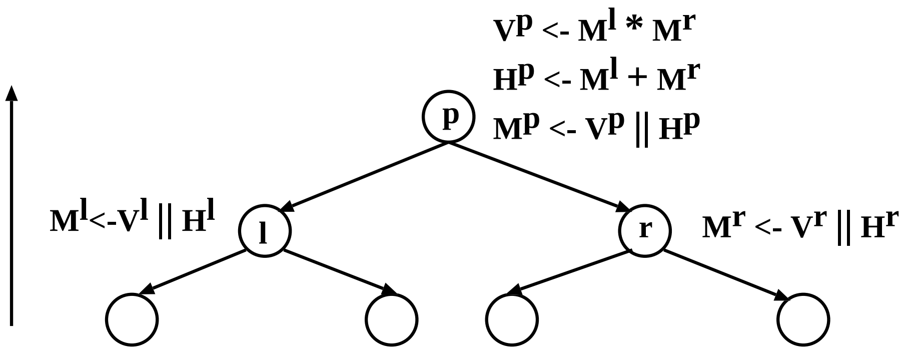


Figure 6.5: Lists of shapes created at an internal node p of *partition tree* $\mathcal{B}$ by *post-order* traversal. '$*$' and '$+$' represent abutment of shapes by *vertical* and *horizontal* cut respectively. '$||$' denotes merging of two lists of shapes obtained by *vertical* and *horizontal* node *sizing.*

Thus, by merging the *vertical* and *horizontal* lists V^p and H^p in increasing order of width, the size of M^p can grow at most by a factor of 2 compared to that in [Sarrafzadeh 1996]. $\square$

The combined lists M^l and M^r created at the left and right children l and r of the node p are used for sub-floorplan generation at its parent node p, as shown in Figure 6.5. Thus, the nodes of the tree $\mathcal{B}$ are processed in *post-order* to generate a set of subfloorplans at every internal node p. We store a subfloorplan at p as a 5-tuple $(w_i, h_i, cut_i, l_\sigma, r_\tau)$, where (w_i, h_i) is the i^{th} shape of node p which is generated by merging $(l_\sigma)^{th}$ shape of left child l and $(r_\tau)^{th}$ shape of right child r using $cut_i \in \{vertical, horizontal\}$.

Theorem 6.1 *By horizontal or vertical node sizing, at most $O(\kappa n^2)$ shapes (slicing trees) can be generated at the root of the tree, where n is the number of modules and κ is the maximum number of irredundant shapes for a module.*

Proof: Let the level of a leaf farthest from the root be 1. For any node at level i ($i = 1, \ldots, \log_2 n + 1$) of the *slicing tree*, the size of the list is $4^{i-1}(\kappa - \frac{2}{3}) + \frac{2}{3}$. Using Lemma 6.3, one can prove this result by induction. The number of nodes at a level i is $\frac{n}{2^{i-1}}$. So, the number of shapes at any level i is bounded by $\frac{n}{2^{i-1}}(4^{i-1}(\kappa - \frac{2}{3}) + \frac{2}{3})$. As $i = O(\log_2 n)$ at the root, the number of shapes/*slicing trees* generated at the root is $(n\kappa \sum_{i=1}^{\log_2 n} 2^{i-1})$, i.e., $O(\kappa n^2)$. $\square$

One such *slicing tree* with its *vertical* and *horizontal* cut lines marked at every internal node, is shown later on in Figure 6.9. During the *post-order* processing of nodes, we also calculate the *resource requirement* $\rho_p = (p^{clb}, p^{ram}, p^{mul})$ at every node p by summing the resources ρ_l and ρ_r required by its left and right child respectively. The requirement vector is used for realization of the *slicing tree* in Phase III.

Corollary 6.1 *The time taken to generate the $O(\kappa n^2)$ slicing trees is at most $O(\kappa n^2)$.*

Proof: The number of *slicing trees* generated is $O(\kappa n^2)$ by Theorem 6.1. The *slicing trees* at level i are produced in $O(\kappa 4^{i-1})$ time. With $O(\frac{n}{2^{i-1}})$ nodes at level i, the generation of *slicing trees* at that level requires $O(\kappa n 2^{i-1})$ time. Therefore, the worst case time complexity of generating all $O(\kappa n^2)$ *slicing tree* is $\sum_{i=1}^{\log_2 n}(\kappa n 2^{i-1})$ $= O(\kappa n^2)$. $\square$

In summary, we process the tree bottom-up only once, generating a set of *slicing trees* $\mathcal{F} = \{(T_{w_i},T_{h_i})\}$ at the root. $\mathcal{F}$ is in increasing order of width and decreasing order of height by our method of construction. We say that $(T_{w_i}, T_{h_i}) \leq (W_t, H_t)$ if $T_{w_i} \leq W_t$ and $T_{h_i} \leq H_t$, and $(T_{w_i}, T_{h_i}) > (W_t, H_t)$ otherwise. If $(T_{w_i},T_{h_i}) \leq (W_t, H_t)$ and the aspect ratio is permissible, the i^{th} *slicing tree* gives a feasible floorplan. There may be *slicing trees* with $(T_{w_i},T_{h_i}) > (W_t, H_t)$, (i.e., apparently infeasible) due to (i) rounding off of *heterogeneous resource requirements* of modules to tiles, and (ii) white spaces generated by node *sizing* of rectangular shapes. However, if the sum of the individual type of *resource requirements* of all modules is less than the total resources available on the target chip, Phase III becomes necessary for reallocation of different types of resources by appropriate re-positioning of *horizontal* and *vertical* cut lines of the *slicing topology.* The shape of some of the modules may ultimately become rectilinear in order to get a feasible floorplan. Further, if the definition of the *basic tile* for a target chip needs a few fractional tiles to cover it entirely, Phase III is necessary to produce a feasible floorplan.

6.5 Phase III: Realization of slicing tree on target FPGA

For every *slicing tree* in $\mathcal{F}$ generated in Phase II, we determine the coordinates of the regions assigned to the modules. There are two steps: (i) *Greedy Allocation of a Rectangular Region* (*GARR*) to each module satisfying the CLB requirements, and (ii) allocation of RAM/MUL blocks of a module in and around this region.

6.5.1 Greedy allocation of rectangular region (GARR)

Each *slicing tree* is traversed top-down, level by level (level order), in left to right order from root towards the leaves. During this traversal, a rectangular region $R_p =$

$(x_p^{min}, y_p^{min}, x_p^{max}, y_p^{max})$ is allocated to every node p by proportionate distribution of available CLB rows and columns using the CLB requirements at p, its sibling and their parent across the given cut line. The entire floorplan rectangle $(0, 0, W, H)$ is allocated to the root node of the slicing tree. Let the CLB requirements at node p, its left child l and its right child r be p^{clb}, l^{clb} and r^{clb} respectively. Let the number of CLB columns (rows) in the rectangle assigned to p be p_{col} (p_{row}). At the root, p_{col} and p_{row} are the number of CLB columns and rows in the chip, e.g. 80 and 104 respectively for Spartan-3 FPGA chip XC3S5000. For a *vertical* cut at p,

$$l_{col} = \frac{l^{clb}}{p^{clb}} . p_{col}; \quad l_{row} = p_{row};$$

$$r_{col} = p_{col} - l_{col}; \quad r_{row} = p_{row};$$

For a *horizontal* cut at p,

$$l_{row} = \frac{l^{clb}}{p^{clb}} . p_{row}; \quad l_{col} = p_{col};$$

$$r_{row} = p_{row} - l_{row}; \quad r_{col} = p_{col}.$$

The rectangular region $(x_p^{min}, y_p^{min}, x_p^{max}, y_p^{max})$ for each node p is computed from respective l_{col} (r_{col}) and l_{row} (r_{row}), depending on the direction of the cut. Realization of each module (exact requirement) is made compact within the assigned region by allocation of individual CLBs from top to bottom (left to right) for *vertical* (*horizontal*) cut. We estimate the quality of this method of greedy allocation using a *minimum cost max flow formulation* (MCMF) in Section 6.8.

6.5.2 Allocation of RAM and MUL

A rectangle assigned as above to a module m_i is guaranteed to have adequate CLBs but it may not have enough RAM/MUL blocks required by m_i. The deficit may have to be met by RAM/MULs located above or below the rectangle of m_i. But, another module may also require the same RAM/MUL block. So, the RAM/MUL requirement constraints are resolved globally by formulating it as a *minimum cost maximum flow (MCMF)* problem [Ahuja 1993], such that a module is not realized in discontinuous regions. For the ease of understanding, without loss of generality we demonstrate below with only RAMs.

We define a flow network $G = (U, Z)$ as follows. Let the set of vertices $U = \{s, t\} \cup U_L \cup U_R$ where s and t are the source and the sink respectively, and $U_L \cap U_R = \phi$. If a module m_i has RAM requirements, then there is a vertex $u_i \in U_L$. Each

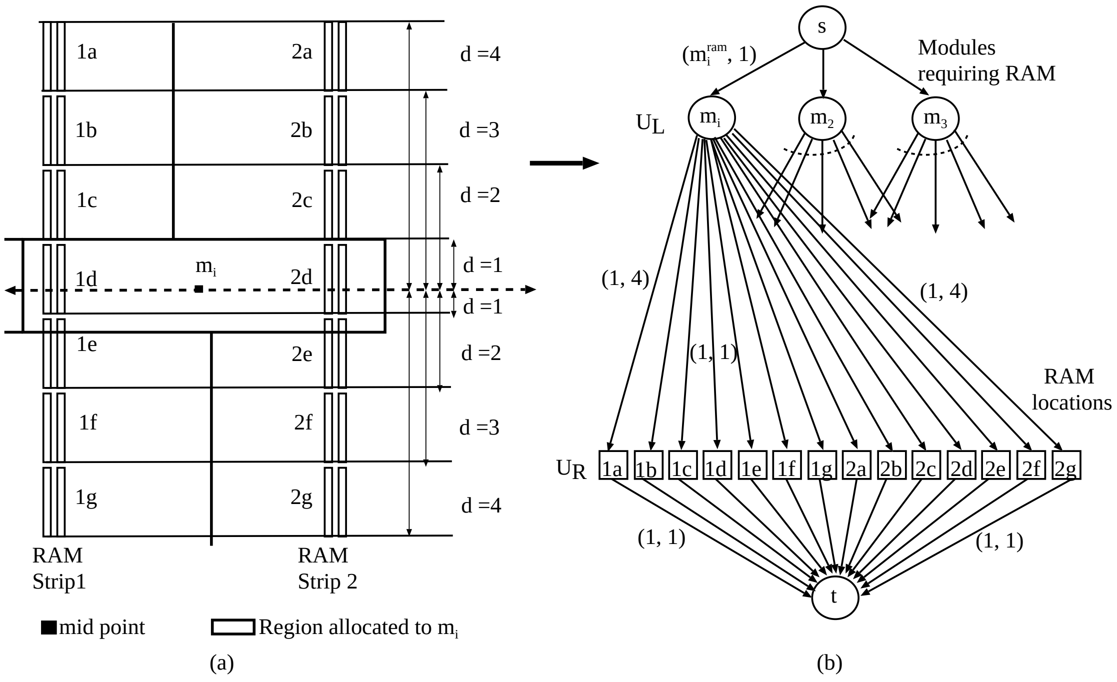


Figure 6.6: Candidate RAM/MUL locations for a module m_i with requirement of 3 RAM blocks, but having only 2 RAM blocks within its allocated rectangular region. The 7 RAM blocks within each of RAM strip 1 and RAM strip 2 are the candidate RAM locations; (b) portion of the min-cost flow network corresponding to RAM/MUL allocation for module m_i showing arcs to all 14 candidate RAM locations $1a, \ldots 1g, 2a, \ldots, 2g$; the pair of numbers on each arc indicate its capacity and cost respectively.

vertex $v_j \in U_R$ corresponds to a candidate RAM location on the target FPGA.

Let the RAM requirement for a module m_i be m_i^{ram}. Suppose, the rectangle R_i $= (x_i^{min}, y_i^{min}, x_i^{max}, y_i^{max})$ has been assigned to m_i as in Section 6.5.1 above. Then, for each column of RAMs intersecting R_i, a module m_i is said to have a *RAM strip* comprising all the RAM locations within R_i, along with m_i^{ram} locations above y_i^{max}, and m_i^{ram} locations below y_i^{min}. The set of arcs is $Z = Z_1 \cup Z_2 \cup Z_3$, where

Z_1= $\{(s, u_i)|u_i \in U_L\}$; the capacity $c(s, u_i)$ of arc (s, u_i) is m_i^{ram} and its cost is 1.

Z_2= $\{(u_i, v_j)|u_i \in U_L \text{and } v_j \in U_R\}$ such that there is an edge between a vertex u_i corresponding to a module m_i and a vertex v_j belonging to the *RAM strip* of the module m_i. The capacity $c(u_i, v_j)$ of arc (u_i, v_j) is 1 and its cost is a rational number representing the vertical distance d between the center of the rectangle R_i and that of the RAM location for v_j.

Z_3= $\{(v_j, t)|v_j \in U_R\}$; the capacity $c(v_j, t)$ of arc (v_j, t) is 1 and its cost is 1.

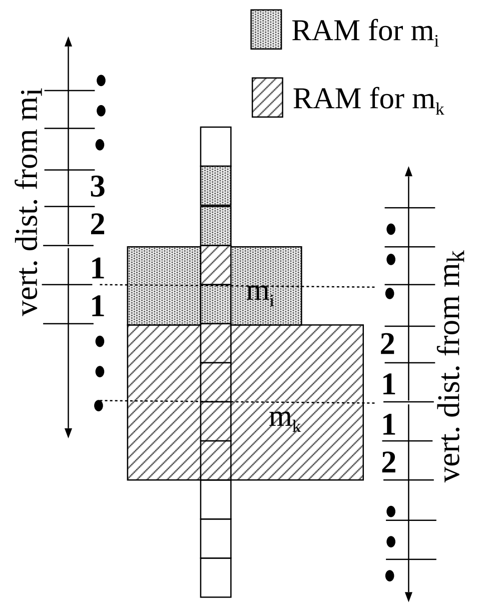


Figure 6.7: Example of an allocation of RAM/MUL which is not *order-preserving*.

Figure 6.6 shows an example. We solve *MCMF* on G to assign RAMs to available RAM locations. We say the RAM assignment is *order-preserving* if two modules m_i and m_k vying for a RAM column where module m_i is placed above m_k, have their RAM allocations also in the same order.

Lemma 6.4 *If the min-cost max-flow f_{max} in the flow network G is equal to the total RAM requirement $\sum_{i=1}^{n} m_i^{ram}$ of a given FPGA floorplanning problem, then there exists a feasible floorplan for the input. Moreover, a min-cost max-flow in G is order-preserving.*

Proof: For the first part, consider a cut $\mathcal{C} = (\{s\}, \{U_L \cup U_R \cup t\})$. The flow f across $\mathcal{C}$ signifies the *RAM* requirement of all modules that have been met by the *RAM* locations. Thus, $f \leq \sum_{u_i \in U_L} c(s, u_i) = \sum_{i=1}^{n} m_i^{ram}$. The capacity constraint of each incoming arc to $v_j \in U_R$ and the outgoing edge (v_j, t) is 1. As flow is conserved at v_j, a *RAM* location can be assigned to only one *RAM* required by a module. Now, if $f_{max} = \sum_{i=1}^{n} m_i^{ram}$ in G, the *RAM* requirements of all modules are satisfied.

For the second part, let module m_i lie above module m_k. Suppose, there exists at least one RAM allocation for m_k (say y_i) above that of RAM implementation for m_i corresponding to the *MCMF* solution f_{max} (vide Figure 6.7 for such an instance). As module m_i is above m_k, the cost of the arc into the vertex for y_i in G is more for m_k than for m_i. This implies that we can swap y_i with any of the RAM blocks below it for m_i, thereby resulting in the same flow value but with a reduced cost.

This implies that the flow f_{max} is not of minimum cost and hence, we arrive at a contradiction. $\square$

Excessively large RAM requirements of adjacent modules might yield an infeasible MCMF. This implies that the chosen *slicing topology* is inappropriate for a feasible realization and is hence rejected. If there is no *slicing tree* with feasible MCMF, a different relative ordering of modules in Phase I is required. The MUL units are also assigned to the physical locations in a similar manner by solving a separate *MCMF.*

Lemma 6.5 *The* MCMF *takes* $O(H^2 \log^2 H)$ *time for an FPGA architecture with co-ordinates* $(0, 0, W, H)$.

Proof: Let R_i be the rectangle assigned to module m_i having l_i RAMs. So, the number of edges from vertex u_i corresponding to m_i is $(l_i + 2 * m_i^{ram})$. Therefore, the total number of edges in G is $\sum_{i=1}^{n}(l_i + 2 * m_i^{ram})$. Both the total number of RAMs enclosed within the non-overlapping rectangles and the total RAM required by all the modules are $O(H)$. So, $|Z| = O(H)$. Also, $|U| = O(H)$. The *MCMF* can be solved in $O(|Z| \log |U|(|Z| + |U| \log |U|))$ time [Ahuja 1993]. With $|Z| = |U| = O(H)$, the time complexity of our *MCMF* is $O(H^2 \log^2 H)$. $\square$

The RAM/MUL allocation as done here by solving an *MCMF* is conceptually similar to the network flow formulation of Feng and Mehta [Feng 2006]. But our formulation, based on one-dimensional geometry, ensures *order-preserving* assignment (vide Lemma 6.4). Further, our formulation leads to a network of size linear in H because we use a *RAM strip*, whereas the size of the flow network in [Feng 2006] is $O(\mathcal{R})$, $\mathcal{R}$ being the total number of CLBs, RAMs and MULs on the FPGA chip. In our formulation, the CLBs, which are in majority, are excluded from the time complexity.

Allocation of regions to modules of a cluster : At the end of Phase III, the shapes of modules within each cluster are to be decided. The region for a cluster is at most the size of a *basic tile* consisting of 80 CLBs, 1 RAM and 1 MUL. Hence, a greedy heuristic is employed to place the modules inside this region. We first place any module having RAM and/or MUL requirement next to the RAM/MUL column in the region, and then the other modules of the cluster in the remaining space. Fast placement methods of Chapters 3 or 4 can also be employed.

The process of CLB assignment followed by RAM and MUL assignment is carried out for every *slicing tree* generated in Phase II. The HPWL is calculated for each floorplan generated. Since RAM/MUL columns are pre-placed on FPGA, 2D compaction of the entire floorplan keeping the wirelength to be minimal, is a challenging

Algorithm 6.1: HeteroFloorplan

```
Input  : Module netlist $S$, list of resource requirements $\rho_m$ of modules,
         FPGA architecture $(0, 0, W, H)$ and its basic tile $A$
Output: A feasible floorplan
1  Preprocessing:
2  Compute $W_t$ and $H_t$;
3  for each module $m_i$ do
4      find the minimum number of tiles required ($T_{m_i}$) from the resource
       requirement vector $\rho_{m_i}$;
5  if $T_{tot}$, total tiles required $> (W_t * H_t)$, and, yet sum of resource requirements
   is less than that available on the FPGA then
6      cluster small modules with resource requirement less than that in basic
       tile $A$, into a set of clusters
7  Phase I: Generation of Partition Tree
8  Recursively bipartition the netlist of modules/clusters based on balanced
   min-cut;
   /* This tree is used as template for slicing topologies                */
9  Phase II: Generation of Floorplan Topologies
10 Enumerate shapes $(w, h)$ in terms of the number of tiles;
11 Determine slicing topology and shapes by vertical and horizontal node sizing
   using post-order traversal of the partition tree of Phase I.
   /* The output is a set of slicing topologies                           */
12 Phase III: Realization of Slicing Tree on FPGA
13 for each slicing tree do
14     GARR: Traverse in level order and allocate rectangular region to each
       module based on CLB requirement;
15     Allocate pre-placed RAM/MULs to modules by minimum-cost
       maximum-flow (MCMF);
16 Report feasible floorplans;
```

task. However, compaction such as in [Cheng 2006] can be applied to *HeteroFloorplan* as well, although it might change the aspect ratio of modules and thereby the wirelength. The floorplans with minimal wirelength and no discontinuity of modules are reported as feasible floorplans.

6.5.3 Time complexity of *HeteroFloorplan*

The overview of our floorplanner is given in Algorithm 6.1.

Theorem 6.2 *The time complexity of HeteroFloorplan, excluding recursive balanced bi-partitioning in Phase I is $O(\kappa n^3 + \kappa n^2 H^2 \log^2 H)$, where H is the height of the chip, n is the number of modules and κ is the maximum number of shapes* *generated for any module.*

Proof: The clustering step takes $O(n^2)$. By Corollary 6.1, the time taken for generating $O(\kappa n^2)$ *slicing trees* is $O(\kappa n^2)$. For each of the *slicing trees*, we traverse the tree of size $O(n)$ from root to leaves in order to fix the rectangular regions of the CLBs in $O(n)$ time. Then, we solve a *MCMF* to assign RAM/MULs in $O(H^2 \log^2 H)$ time by Lemma 6.5. The total time complexity is thus $O(\kappa n^2(n + H^2 \log^2 H))$, or $O(\kappa n^3 + \kappa n^2 H^2 \log^2 H)$. □

The authors of *hMetis* in [Karypis 1999b] claim that in spite of its iterative nature, the time taken by *hMeTis* is almost linear in the number of hyperedges, i.e., netlists. It may be noted that the value of κ is very small, typically 6 to 10 for a floorplanning problem with hundreds of modules. The reason is that as the value of n increases, the maximum value of κ decreases. Thus, our $O(\kappa n^3 + \kappa n^2 H^2 \log^2 H)$ method compares favorably against that of [Yuan 2005] which the authors claim to take $O(W^2 n^5 \log n)$ time (although as per [Berg 2000] the $\log n$ term should be replaced by $\sqrt{n}$). It may be noted here that as the other two methods, viz. [Cheng 2004] and [Feng 2006], use simulated annealing, it is not possible to compare our time complexity with that of those two methods.

6.6 An example

Here, we demonstrate our floorplanning method with a synthetic example circuit taken from [Cheng 2004, Cheng 2006]. Cheng and Wong [Cheng 2004] devised an experiment as follows. They took XC3S5000 which is the largest chip in the Xilinx [Xilinx] Spartan 3 family. They divided the XC3S5000 almost evenly into 20 blocks, each corresponding to a module. Of the 20 modules, 16 need 400 CLBs, 5 RAMs and 5 MULs each, and the rest of the 4 modules need 480 CLBs, 6 RAMs and 6 MULs each. The authors claim this to be a very tight problem. A little bit of inefficiency on the part of the floorplanning method could render the floorplan to be infeasible. The method *HeteroFloorplan* is explained with this tight problem of [Cheng 2004]. We considered the same circuit from [Cheng 2004] with 20 modules and constructed an appropriate netlist for comparison purpose.

Figure 6.9 shows the binary *partition tree* obtained in Phase I of *HeteroFloorplan.* The integers $0, \ldots, 19$ written below the leaves, indicate the indices of the modules. A set of *slicing trees* is generated in Phase II. One such slicing tree with its *vertical* and *horizontal* cut lines marked at every internal node, is shown here. Finally, the realization of the *slicing tree* on to the coordinates of the target architecture in

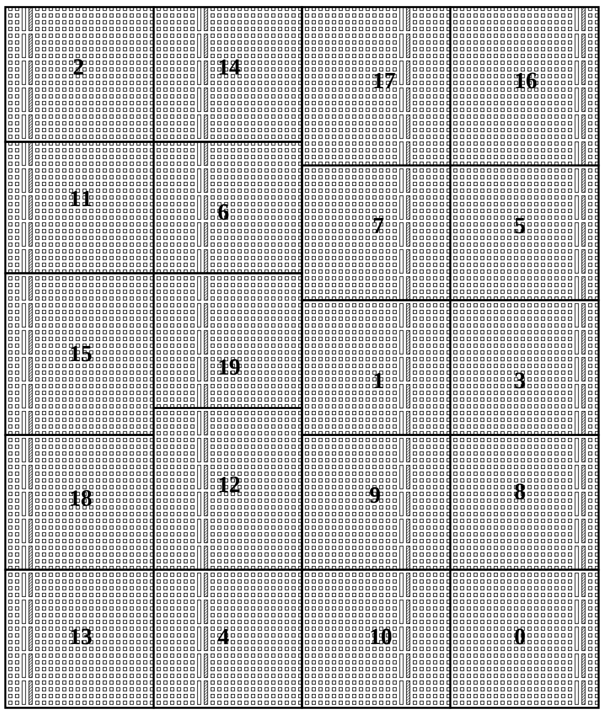


Figure 6.8: An example circuit with 20 modules [Cheng 2004]: floorplan produced by *HeteroFloorplan*.

terms of $(x^{min}, y^{min}, x^{max}, y^{max})$ are reported in a vertical box below every node. For example, the root is realized as (0,0,87,103), i.e., the entire target architecture. Within the floorplan area, a module, say m_{11}, is realized as $(0, 64, 21, 83)$. Figure 6.8 shows the final allocation for each module on Spartan-3 XC3S5000.

Table 6.1: Floorplan results for *20-module example* [Cheng 2004]

Index of *slicing tree*	(T_w,T_h)	Wirelength(HPWL)	Avg. aspect ratio
1	$(1, 104)$	392	17.01
2	$(2, 52)$	560	4.25
3	$(3, 39)$	X	X
4	$(4, 26)$	816	1.06
5	$(5, 22)$	X	X
6	$(6, 20)$	X	X
7	$(7, 17)$	X	X
8	$(8, 14)$	X	X
9	$(9, 13)$	X	X
10	$(10, 11)$	X	X

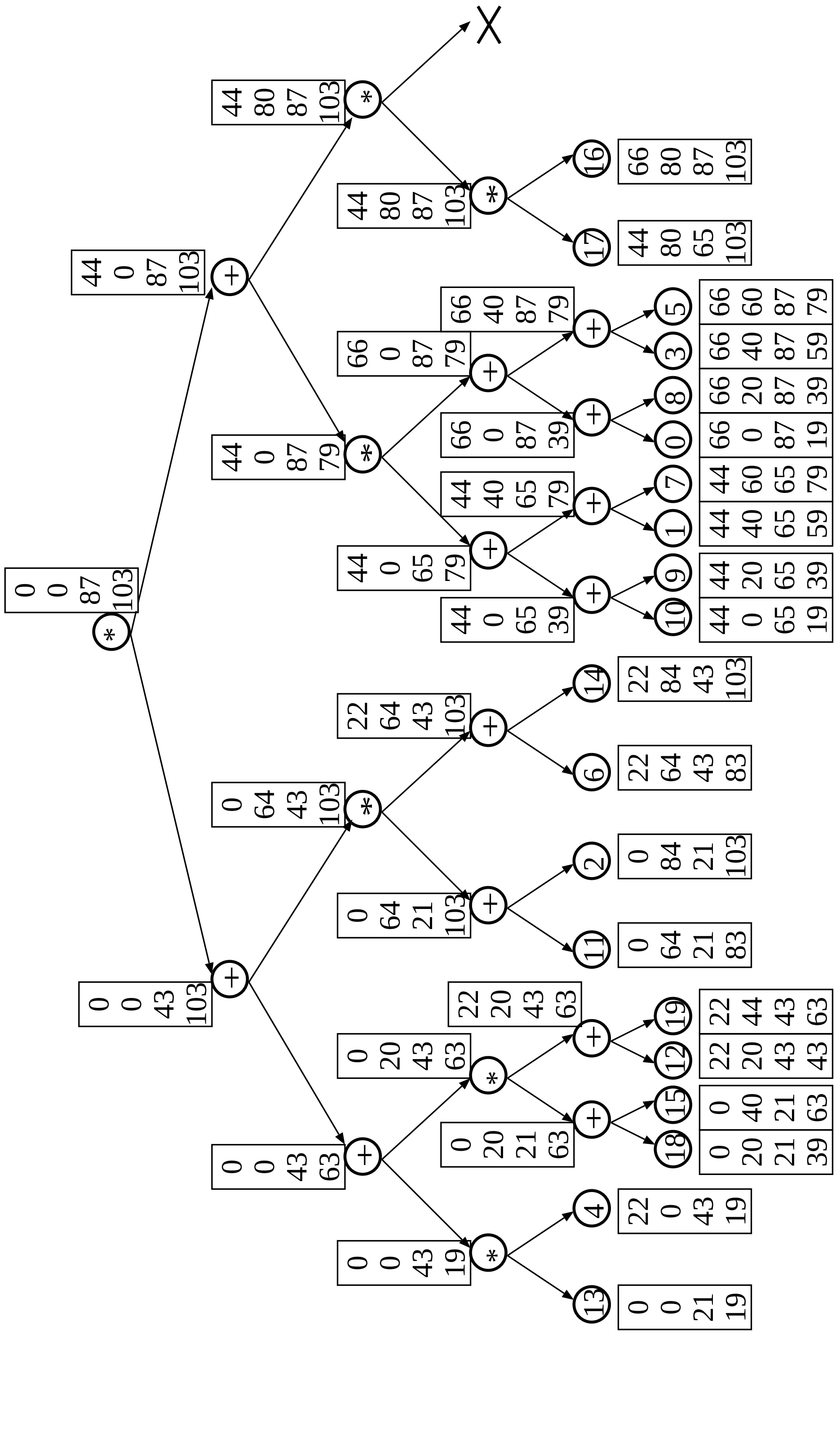


Figure 6.9: An example circuit of [Cheng 2004]: one of its *slicing trees* and the box next to each node gives the four co-ordinates $(x^{min}, y^{min}, x^{max}, y^{max})$ of the region allocated to the node.

The effectiveness of *HeteroFloorplan* is amply demonstrated with the 20-module example circuit of [Cheng 2004] that covers the entire target architecture. Table 6.1 shows the result obtained by *HeteroFloorplan* for the example circuit. The column (T_w, T_h) is the width and height (in terms of *basic tiles*) of each floorplan topology generated after phase II. The column marked wirelength (HPWL) shows the wirelength obtained for each *slicing tree* realization after phase III. The X mark in the column titles $HPWL$ indicates that the corresponding *slicing tree* is infeasible.

The 4^{th} slicing tree has $(T_w, T_h) = (4, 26) = (W_t, H_t)$. Hence, for this topology, we need not execute Phase III. However, we execute Phase III on all the slicing topologies generated if their $(T_w, T_h) > (W_t, H_t)$. As mentioned in Section 6.4.2, the *tiles* remain under utilized due to the mismatch in three different *resource requirements* of a module. The topology can be rendered feasible by modifying the cut lines. Thus, we execute phase III on all the topologies to get a set of feasible floorplans. Table 6.1 shows that the configurations 1 and 2 have $(T_w, T_h) > (W_t, H_t)$ and they have become feasible after execution of Phase III. However, topologies with $T_h < H_t$ and $T_w > W_t$ usually lead to infeasible solutions. The time taken by *HeteroFloorplan* to generate the floorplans for all the 10 *slicing trees* is 2.98 seconds on a relatively slower 1.2GHz SunBlade 2000, which is far less than 88 seconds taken by [Cheng 2004] on a faster 2.4GHz Intel (R) Xeon CPU. We observed that *HeteroFloorplan* can construct the same floorplan of [Cheng 2004] with an appropriate *partition tree.* In Table 6.1, the topology in column 5 (*slicing tree* index 4), is identical to that reported in [Cheng 2004]. Since [Cheng 2004] does not report the wirelength for this 20 module example, we cannot compare it with ours. Further, many of the *slicing trees* remained infeasible as the *resource requirement* is very tight. Column 4 in Table 6.1 shows the average aspect ratio of module dimensions. The first two floorplan realizations have smaller wirelength, but the average aspect ratios are far away from 1. But the realization (4, 26) has the average aspect ratio very close to 1 and we report this as the final feasible floorplan.

6.7 Experimental results

We have implemented the proposed method in 'C' using LEDA [LEDA] on 1.2GHz SunBlade 2000 workstation with SunOS Release 5.8. Our method is tested on Xilinx XC3S5000 (Spartan-3) FPGA architecture with 8320 CLBs, 104 RAMs and 104 MULs. These are arranged in 88 columns (including 4 RAM-MUL column pairs) and 104 rows of CLBs. The basic tile size is $A = (20 \times 4, 1, 1)$.

Experimental results on nine benchmark circuits derived from MCNC [Cheng 2004]

Table 6.2: Benchmark circuits, C: CLB, R:RAM, M:MUL

1	2	3	4
	Circuit Characteristic		
Circuit	**#Modules**	**#Nets**	**#(C, R, M)**
ideal	20	18	$(8320, 104, 104)$
apte	9	44	$(6614, 70, 70)$
xerox	10	183	$(6625, 66, 50)$
hp	11	44	$(6591, 66, 66)$
ami33	33	84	$(6289, 61, 60)$
ami49	49	377	$(6300, 63, 63)$
n100a	100	576	$(6352, 39, 38)$
n200a	200	1585	$(6342, 44, 34)$
n300a	300	1893	$(6399, 65, 54)$

Table 6.3: Comparison of wirelength (HPWL *a la* ASIC): *HeteroFloorplan* vs. [Feng 2006]; Case I: center-to-center; Case II: terminals on periphery

1	2	3	4	5	6	7
	HeteroFloorplan			Method in [Feng 2006]	% Gain	
		HPWL		HPWL		
Circuit	(T_w, T_h)	I	II	II	I	II
ideal	$(4, 26)$	816	1605	–	–	–
apte	$(4, 22)$	189720	441801	–	–	–
xerox	$(4, 28)$	919485	1636096	–	–	–
hp	$(4, 26)$	106170	194597	–	–	–
ami33	$(4, 30)$	42623	87592	89283	52	20
ami49	$(3, 45)$	950322	1242337	1173000	18	-5
n100a	$(4, 39)$	209456	371339	358338	41	-3
n200a	$(4, 38)$	439154	594254	700045	37	15
n300a	$(4, 42)$	690391	886015	875602	21	-1
Average						

and GSRC (Giga Scale Research Center) Bookshelf ASIC floorplanning benchmarks [GSRC] are reported next. ASIC benchmarks are converted to FPGA benchmarks as in [Feng 2006] by proportional CLB requirements. Table 6.2 has the details of the 9 benchmark circuits, namely, the number of modules, the signal nets, the total requirements of the three types of resources, in columns 2, 3 and 4 respectively.

Table 6.3 shows the comparative results of wirelength (HPWL) obtained by *HeteroFloorplan* and by that in [Feng 2006]. The second column reports the (T_w, T_h)

pair in Phase II, for which we obtained a feasible solution with aspect ratio closest to 1. As explained in Section 6.4.2, $(T_w, T_h) > (W_t, H_t)$ in many of the cases. A feasible floorplan of the circuit *ami33* after Phase III is shown in Figure 6.10.

The HPWL is computed after scaling the FPGA board to ASIC dimension as in [Feng 2006]. The wirelength HPWL can be computed with the assumptions that the terminals are either (Case I) at the center of the modules, or (Case II) on the periphery of the modules, as shown in Figure 6.11. As the locations of terminals on the soft modules are not known for the benchmarks, the center-to-center HPWL (case I) considering the terminals at the center of a module, is reported in column 3. Since the method in [Feng 2006] employs SA based PARQUET [Adya 2003, PARQUET], the reported wirelength is assumed to be computed considering the terminals on the periphery (Case II) of each module. For comparison purpose, we take the worst case scenario by considering the terminals to be at the top-left and bottom-right corner of the enclosing bounding box of the net. The worst case HPWL thus obtained is reported in column 4 and as expected, the values are much larger than center-to-center HPWL. Column 5 reports the wirelength (HPWL) directly from [Feng 2006]. We observed that the percentage gain in HPWL (columns 6-7) ranges between 18% to 52% in Case I (comparing columns 3 and 5) and −5% to 20% in Case II (comparing column 4 and 5). On the average, the improvement for

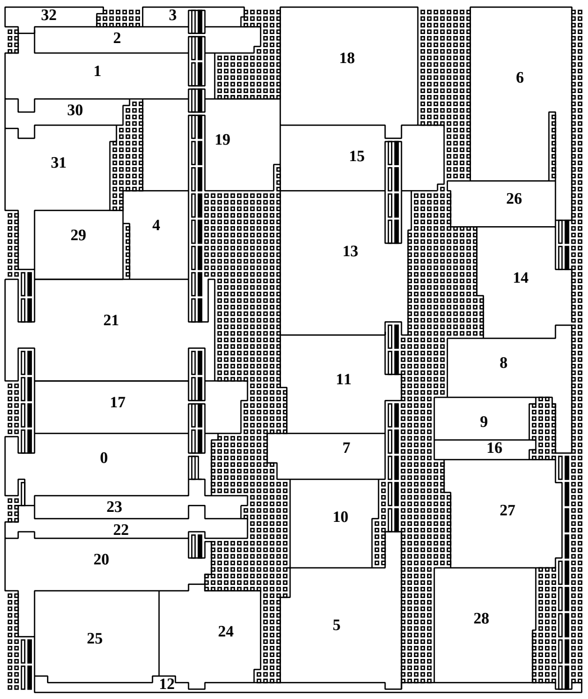


Figure 6.10: Floorplan of *ami33* after phase III for $(T_w, T_h) = (4, 30)$.

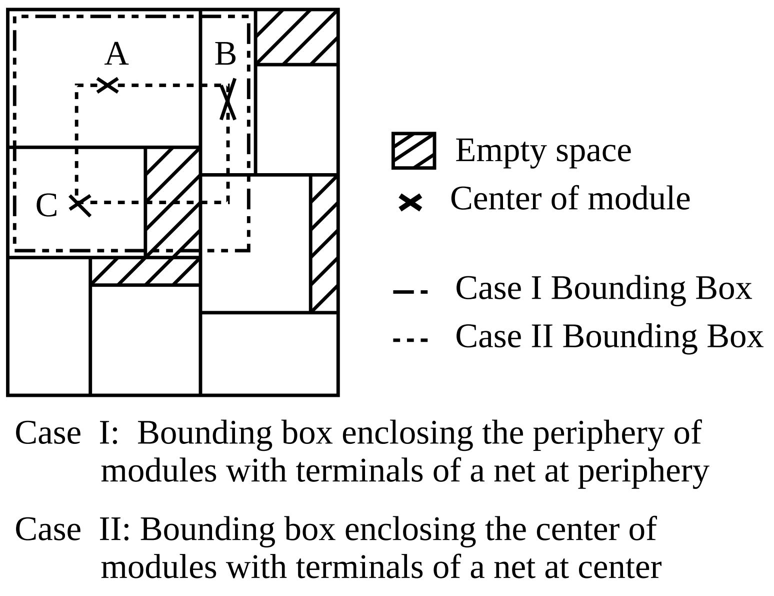

A
B
C
Empty space
Center of module
Case I Bounding Box
Case II Bounding Box
Case I: Bounding box enclosing the periphery of modules with terminals of a net at periphery
Case II: Bounding box enclosing the center of modules with terminals of a net at center

Figure 6.11: Computation of bounding box of a net with terminals on modules marked A, B and C.

Table 6.4: Comparison of wirelengths (HPWL): *HeteroFloorplan* vs. [Cheng 2006]

1	2	3	4	5
	HeteroFloorplan		Method in [Cheng 2006]	
Circuit	(T_w, T_h)	HPWL	HPWL	% Gain
ideal	$(4, 26)$	758	–	–
apte	$(4, 22)$	2599	2704	3
xerox	$(4, 28)$	9187	10476	12
hp	$(4, 26)$	2732	3123	12
ami33	$(4, 30)$	3644	4114	11
ami49	$(3, 45)$	13336	15311	12
n100a	$(4, 39)$	25896	30295	14
n200a	$(4, 38)$	58586	–	–
n300a	$(4, 42)$	72820	–	–
Average				

Case I and Case II are 34% and 5% respectively.

The center-to-center HPWL on the 88×104 XC3S5000 board is also compared with that of [Cheng 2006] in Table 6.4. Column 2 shows the same (T_w, T_h) as in Table 6.3. The center-to-center HPWL obtained by *HeteroFloorplan* is shown in column 3. The wirelength obtained by [Cheng 2006] is reported in column 4. We observed from column 5 that the percentage gain in HPWL ranges from 3% to 14%.

Figure 6.12 shows how the wirelength increases as the aspect ratios approach 1, for a subset of circuits. The user can select a feasible floorplan from a set of *slicing trees* with a trade-off between aspect ratio and wirelength as desired.

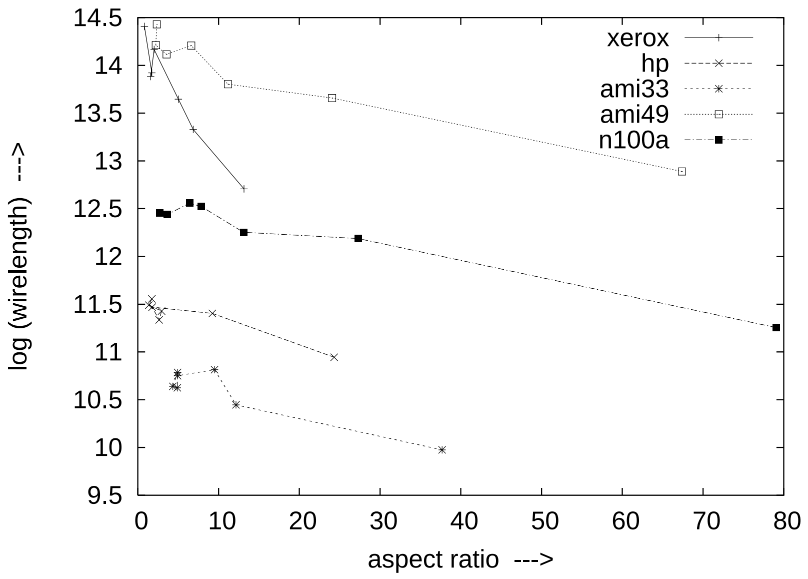


Figure 6.12: Variation of HPWL (Case I) with average aspect ratio of a module.

Table 6.5 shows the CPU time (in secs.) taken by *HeteroFloorplan* to produce the floorplans on SunBlade 2000 workstation which is much slower than the platform reported in [Feng 2006, Cheng 2006]. For a fair comparison of CPU time with simulated annealing based fixed outline ASIC floorplanner PARQUET [Adya 2003, PARQUET], we report the CPU time taken by default PARQUET (without wirelength minimization) on our platform in column 2. The min-cost max-flow based Constrained Floorplanning (CF) step of [Feng 2006] is implemented using LEDA [LEDA] and the time taken is reported in column 3. The number of *slicing trees* generated in Phase II and the time taken by our method are given in columns 4 and 5 respectively. From the experiments, we conclude that the combined step of PARQUET and CF is 2× to 373× slower depending on the size of the circuit. As [Cheng 2006] is also based on simulated annealing followed by a compaction step, the time taken must be more than the time taken by PARQUET. Hence *HeteroFloorplan* must be faster than [Cheng 2006] by the same order of magnitude as [Feng 2006]. The CPU time taken by *HeteroFloorplan* can be further reduced by pruning some slicing trees before realizations. In summary, the experimental results establish the suitability of our method for fast FPGA floorplanning.

Table 6.5: Comparison of CPU time; [CF: Constrained Floorplanning]

1	2	3	4	5
	Time(s)	**[Feng 2006]**	*HeteroFloorplan*	
Circuit	**Parquet**	**CF**	**#slicing trees**	**Time(s)**
ideal	–	–	20	2.98
apte	1.78	0.61	16	1.22
xerox	2.1	0.61	20	1.02
hp	2.57	0.61	18	0.96
ami33	19.9	0.55	23	1.39
ami49	43.12	0.61	23	3.84
n100a	92.31	0.95	24	1.16
n200a	695.92	0.92.	32	2.6
n300a	1605.07	1.02	39	4.3

6.8 How good is our GARR?

In this section, we evaluate our greedy heuristic of positioning the cut line in the *slicing topology* based on the *resource requirements* on either side of that cut line, against a network flow based formulation that is global in nature. Our greedy strategy is locally optimal because it determines the location of a particular cut line at node p from the ratio of the CLB requirements of its left child to that of its right child.

6.8.1 Max-flow formulation for CLB allocation

The slicing trees generated in Phase II of our method may have shapes (T_w, T_h)> (W_t, H_t), as explained at the end of section 6.4.2. For a feasible floorplan within the fixed outline of the target chip, the floorplan is linearly scaled down to the target $(0, 0, W, H)$ obtaining a realization $\mathcal{S}$. Linear scaling may not guarantee satisfaction of *resource requirements* of each module within the rectangle assigned to it in $\mathcal{S}$. But, there are white spaces generated in $\mathcal{S}$ due to tile estimation of *resource requirement* of module and node *sizing*. To satisfy the CLB *resource requirements* of each module contiguously by using white spaces available in the rectangles assigned to the modules, a flow network is defined similar to [Feng 2004]. As shown in Figure 6.13(a), $\mathcal{S}$ is a rectangle dissection of the target chip $(0, 0, W, H)$, n of which are earmarked for the modules and the remaining are white spaces. For a rectangle R_j earmarked for module m_i, let the available resource in R_j be ra_j and the resource requirement of m_i be rr_i. For an empty rectangle R_k, only available resources ra_j

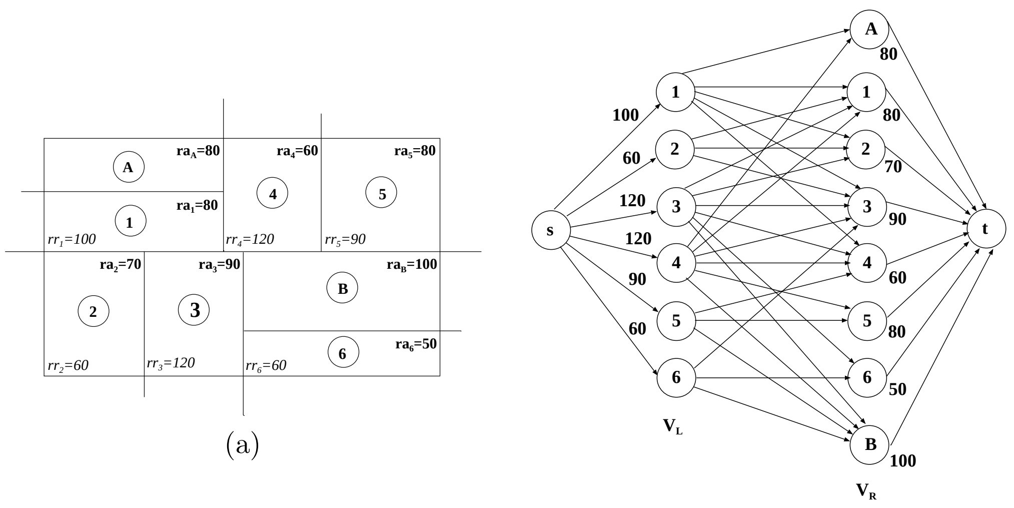


Figure 6.13: (a) A linearly scaled down realization $\mathcal{S}$ of a node-sized slicing tree to fit an $(0, 0, W, H)$ architecture. Rectangles marked $1, \ldots, 6$ denote the module realizations; ra_i (rr_i) denotes resource available (requirement). (b) Flow network corresponding to the realization shown in (a).

is defined.

We define the corresponding flow network $G = (V, E)$ as follows. Let $V = \{s, t\} \cup V_L \cup V_R$ where s and t are the source and the sink vertices respectively. Each vertex $u_i \in V_L$ corresponds to module m_i to be floorplanned. Each vertex $v_j \in V_R$ corresponds to the rectangle R_j. Certain rectangles are earmarked for modules; the rest are white spaces that are to be utilized for reallocation of CLBs to satisfy *resource requirements*. Again, there are three types of arcs in E, i.e. $E = E_1 \cup E_2 \cup E_3$ where

$E_1 = \{(s, u_i) | u_i \in V_L\}$; capacity $c(s, u_i)$ of (s, u_i) is rr_i.

$E_2 = \{(u_i, v_j) | u_i \in V_L$ and $v_j \in V_R\}$ such that either $i = j$, or R_i and R_j are adjacent horizontally, vertically or diagonally; capacity $c(u_i, v_j)$ of (u_i, v_j) is ra_j.

$E_3 = \{(v_j, t) | v_j \in V_R\}$; capacity $c(v_j, t)$ of (v_j, t) is ra_j.

Figure 6.13(b) shows the flow graph for the realization in Figure 6.13(a). The capacity for each edge from s to V_L is shown next to the edge, and that for any edge incident on a vertex in V_R is marked beside the vertex. The capacity $c(u_i, v_j)$ of an edge $(u_i, v_j) \in E_2$ is assigned in such a way that the flow $f(u_i, v_j)$ is equal to either the entire or a part of the CLB requirements of module m_i met by rectangle R_j.

Definition 6.7 (Feasible realization:) *A realization $\mathcal{S}$ of b rectangles and n modules is said to be a feasible realization of a given floorplan problem if $\sum_{i=1}^{n} rr_i \leq \sum_{j=1}^{b} ra_j$ and the module requirements are met by the resource available with the rectangles.*

Along the lines of [Feng 2004], we have the following observation.

Observation 6.1 *Let f_{max} be the maximum flow in the network G corresponding to the realization $\mathcal{S}$. If $f_{max} = \sum_{i=1}^{n} rr_i$, then $\mathcal{S}$ is feasible.*

Proof: The proof follows as a general case of Lemma 6.4. Also, see [Feng 2004] for details. $\square$

By flow computation, if we obtain a feasible $\mathcal{S}$, then a module m_i can be assigned to one or more neighboring rectangles (as per arc set E_2 of network G above), with a condition that it does not become discontinuous. It may be observed that each rectangle R_j has to be partitioned among the modules (part or whole) corresponding to $V_L^s \subseteq V_L$ (a set of vertices from which non-zero flow values originate), which itself is again a floorplanning problem of smaller size. Hence, we adopted the greedy method GARR and not this network flow approach in our *HeteroFloorplan*. Nevertheless, this network flow formulation gives us a global picture of the realization of a slicing tree, against which we can validate our greedy heuristic *GARR*.

6.8.2 Comparison of GARR with network flow method

Let $M = \{m_1, m_2, \ldots, m_n\}$ be a set of n distinct modules with CLB requirements being $\{c_1, c_2, \ldots, c_n\}$ respectively. Let c_{ij}^o and c_{ij}^{nf} be the number of CLBs of module m_i that are contained in rectangle R_j by our method and the network flow based method respectively. As both c_{ij}^o and c_{ij}^{nf} may be 0, we introduce two new boolean variables x_{ij}^o and x_{ij}^{nf}; $x_{ij}^o = 0$ ($x_{ij}^{nf} = 0$) means module m_i is not implemented within rectangle R_j by *HeteroFloorplan* (by network flow based method); similarly, $x_{ij}^o = 1$ ($x_{ij}^{nf} = 1$) means module m_i or a part of it has been implemented in rectangle R_j by *HeteroFloorplan* (by network flow based method). Figure 6.14 gives an illustration of comparison of original slicing shapes generated and the ones obtained by adjusting cut lines (dotted lines) as done by *HeteroFloorplan*.

In order to measure the similarity between the CLB distribution of our floorplan and that by the network flow based method, we define a metric χ for percentage of match as $\chi = (1 - \psi).100$, where,

$$\psi = \frac{1}{b} \sum_{j=1}^{b} \frac{1}{\sum_{i=1}^{n} (x_{ij}^o \vee x_{ij}^{nf})} \sum_{i=1}^{n} \frac{|c_{ij}^o - c_{ij}^{nf}|}{c_i} \tag{6.13}$$

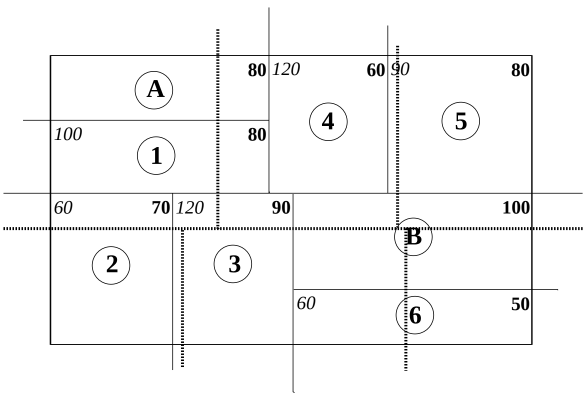


Figure 6.14: A comparison with original slicing shapes generated (solid lines) and the ones obtained by shifting cut lines (dotted lines) as done by *HeteroFloorplan*.

Table 6.6: Similarity of CLB allocation by GARR in *HeteroFloorplan* with network flow based method

1	2	3
Circuit	#rectangles	χ %
ideal	20	96
apte	10.5	83
xerox	11.9	81
hp	12.8	76
ami33	41.4	75
ami49	57.8	63
n100a	110.0	65
n200a	219.5	63
n300a	219.1	62

The parameter ψ can take a real value in $[0, 1]$. While $\psi = 0$ indicates an absolute match between the CLB distribution of our floorplan and the network flow based one, $\psi = 1$ implies an absolute mismatch. Table 6.6 presents the value of χ for the different circuits under consideration. Column 2 lists the average number of rectangles and Column 3 gives χ. As can be observed from the values of χ ranging from 62% to 96%, *HeteroFloorplan* follows to a great extent, resource allocation in the network flow based method. Furthermore, due to inclusion of the diagonal neighbors in the edge set E_2, the CLBs of the modules may be distributed among all its neighbors. But, *HeteroFloorplan* realizes the floorplans by *horizontal* and *vertical* cut lines, thereby retaining mainly rectangular shapes. This explains why the value of χ is not very close to 100% in some of the cases.

6.9 Conclusion

In this chapter, we have reported a fast floorplanning methodology for FPGAs with *heterogeneous* resources consisting of CLBs, RAMs and Multipliers as in Spartan-3 FPGA architecture. We propose a deterministic three phase method for unified floorplan topology generation and *sizing* for such *heterogeneous* FPGAs. The time complexity of our approach excluding the recursive balanced bi-partitioning by *hMetis* in Phase I, is $O(\kappa n^3 + \kappa n^2 H^2 \log^2 H)$, where H is the height of the chip, n is the number of modules and κ is the maximum number of shapes generated for any module. This is an improvement over the other deterministic heuristic presented in [Yuan 2005]. Experimental results demonstrate a speed-up in the range of $2\times$ to $373\times$, depending on the size of circuit, over the existing methods, and an improvement on the average, of 34% in wirelength. Further, we evaluated our greedy resource allocation *GARR* against a network flow based formulation to establish that *GARR* measures up to the global allocation by max-flow method. A two-dimensional compaction step in the presence of pre-placed resources without sacrificing the wirelength is a challenging task and needs deeper investigation. Our method can be used for other architectures by selecting appropriate tile structure.

CHAPTER 7

Floorplanning for Partial Reconfiguration in FPGAs

Contents

7.1 Introduction

Modern FPGA architectures like Xilinx Virtex series allow *partial dynamic reconfiguration* [Xilinx]. This implies that inactive parts of a design implemented on FPGA chip can be replaced by other designs while the remaining part of FPGA continues to execute. Thus, *partial reconfiguration* helps executing a large application in the same piece of hardware by swapping in and out the active and inactive parts of design when the whole application does not fit completely on the chip. This incurs an additional *partial reconfiguration* overhead each time the *bitstream* for a new part is loaded on the FPGA chip. Hence, an appropriate *scheduling* of task/application/design is necessary to reduce the *partial reconfiguration* overhead such that common tasks/designs need not be programmed repeatedly. Given a *schedule of instances* consisting of a set of common as well as other tasks, the resources on the chip may get fragmented due to arbitrary placement of tasks on the chip. The tasks of the consecutive *instances* may not fit contiguously in the fragmented resources scattered across the chip. This may lead to reconfiguration of the whole chip to make contiguous space for each task incurring a reconfiguration overhead. This may defeat the whole purpose of partial reconfigurability. Modern FPGAs are *heterogeneous* in nature with pre-placed blocks like RAM, Multipliers along with array of CLBs. For such FPGAs, the mapping of tasks allocating *heterogeneous* resources contiguously for each *instance*, meeting the performance objective, becomes more complex. In this chapter, we propose a fast performance aware *global floorplan* generation method for the tasks/modules of each *instance* of a given *schedule* such that the common tasks/modules across *instances* occupy the same position and shape on the target FPGA chip resulting in minimal reconfiguration overhead. As modules other than the common modules are placed relative to the common modules depending on their connectivity, the total half-perimeter wirelength (HPWL) over all *instances* is also optimized.

As most of the floorplanning methods such as [Singhal 2006] for *partial reconfiguration* are based on stochastic methods, they suffer from long execution time. We overcome the long execution time of simulated annealing based approach and yet arrive at a *global floorplan* with our fast deterministic *global topology generation* and *sizing* method *PartialHeteroFP*. Our method places the common modules with same shape across all *instances* at a specific position on the chip so that they need not be reconfigured repeatedly. Moreover, the remaining modules are placed in such a way that the total wirelength (HPWL) over the entire *schedule* is minimized globally. Unlike the method in [Singhal 2006], where sequence-pair is used as floorplan representation for simulated annealing moves, the method proposed here uses *slicing tree* [Sarrafzadeh 1996] representation and node sizing for topology generation.

The rest of the chapter is organized as follows. In Section 7.2, we formulate the problem for *partial reconfiguration* and describe our three phase method in brief. The steps of the proposed method are detailed in Sections 7.3, 7.4 and 7.5 respectively. The method is illustrated with an example in Section 7.6. Section 7.7 reports the experimental results. Concluding remarks appear in Section 7.8.

7.2 Floorplanning for partial reconfiguration

The *resource requirement vector* of a module ρ_m is defined as in Definition 6.2 of Chapter 6.

Definition 7.1 (Static and Dynamic modules:) *In a schedule of instances, modules which are common and remain active in all instances are called static modules. The rest of the modules of an instance, which are swapped in and out, are called dynamic modules.*

The floorplanning problem for *partial reconfiguration* is essentially the generation of a *global* floorplan where each floorplan corresponding to each *instance* of a *schedule* is a feasible floorplan. The common modules are placed at the same location with same shape in each of the *instances*, while the total HPWL across all *instances* is minimal. We build upon the single *instance* floorplanning problem for *heterogeneous* FPGAs of Chapter 6 as follows.

In a given *schedule*, let there be

- q *instances* $I_1, I_2, \cdots I_q$,
- $SM = \{\sigma_1, \sigma_2, \cdots \sigma_m\}$, be the m *static* modules that remain active in all *instances*,
- $DM_i = \{\delta_{i1}, \delta_{i2}, \cdots \delta_{in_i}\}$ be the n_i modules of an *instance* $I_i, 1 \leq i \leq q$,
- the netlist of CLBs $\mathcal{S}_i$ for each *instance* of $I_i, 1 \leq i \leq q$

The objective is to find floorplans for all *instances*, such that

- the resource requirement of a module (either *static* or *dynamic*) ρ_m is satisfied within a region $(x^{min}, y^{min}, x^{max}, y^{max})$ in each *instance* without overlap,
- the location and shape (i.e., width and height) of each *static* module is same across all *instances*,
- the half-perimeter wirelength (HPWL) of netlist for all *instances* is minimized.

Floorplanning for *partial reconfiguration* is also NP-hard and thus needs design of very fast and effective heuristic in practice to utilize the benefit of reconfigurability.

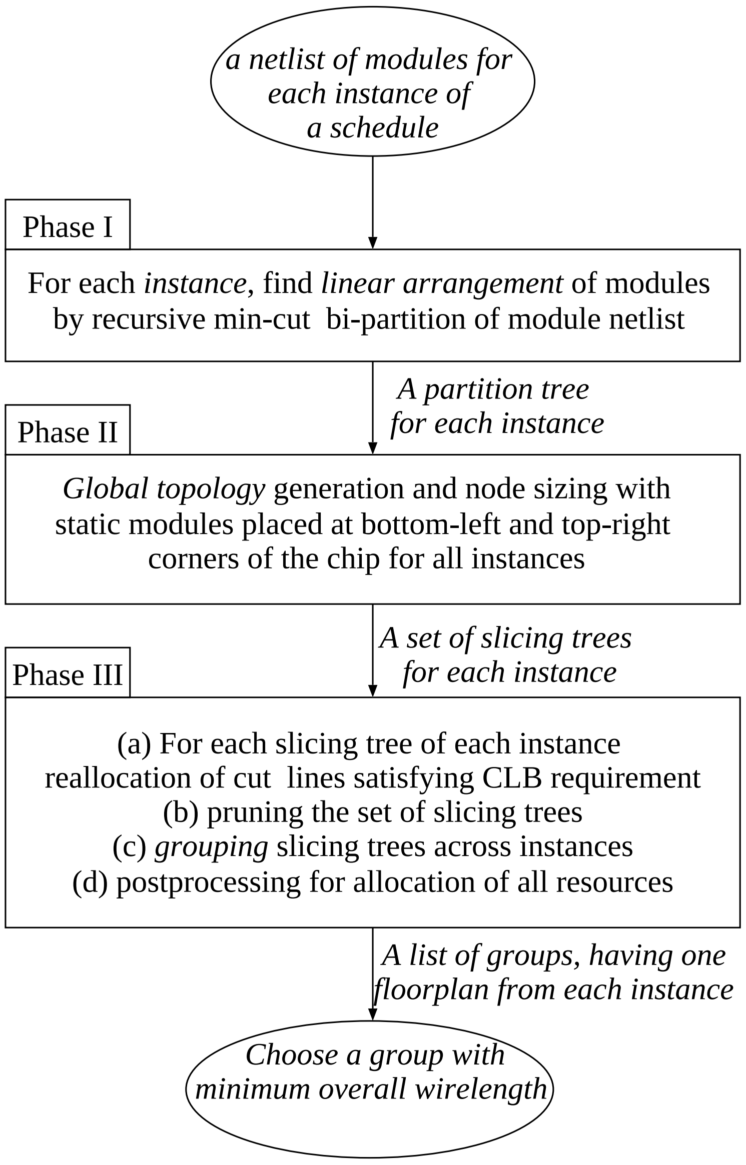

a netlist of modules for each instance of a schedule
Phase I
For each instance, find linear arrangement of modules by recursive min-cut bi-partition of module netlist
A partition tree for each instance
Phase II
Global topology generation and node sizing with static modules placed at bottom-left and top-right corners of the chip for all instances
A set of slicing trees for each instance
Phase III
(a) For each slicing tree of each instance reallocation of cut lines satisfying CLB requirement
(b) pruning the set of slicing trees
(c) grouping slicing trees across instances
(d) postprocessing for allocation of all resources
A list of groups, having one floorplan from each instance
Choose a group with minimum overall wirelength

Figure 7.1: Flow of the proposed method *PartialHeteroFP*.

7.2.1 Overview of proposed method

Our method *PartialHeteroFP* consists of three phases as shown in Fig. 7.1.

In the first phase, we obtain a *linear arrangement* of modules for each *instance* in order to bring heavily connected modules closer, minimizing the wirelength. The *linear arrangement* is obtained by recursive bi-partitioning based on *min-cut* for each of the q *instances* separately, where the positions of the *static* modules are kept invariant across all *instances.* The *partition trees* obtained are used as the templates for topology generation.

In the second phase, a list of *global slicing topologies* is generated for each *instance* such that the positions of *static* modules are fixed at diagonally opposite corners of all floorplans leaving as much contiguous space for *dynamic* modules as possible.

Finally, on the basis of similarity between *slicing trees* of different *instances*, a set of groups, each *group* having a set of *slicing trees*, is generated. For each *slicing tree* in each *group*, a rectangular region is assigned to every module, which respects the cut direction and the actual resource requirement of the modules. The *group* with least total wirelength is the final solution.

7.2.2 Basic tile on FPGA chip

For soft modules with homogeneous resource requirement such as only CLBs, the requirement can be factorized to generate a set of possible shapes (i.e, width and height), which can be later used for node sizing in traditional topology generation when floorplans are represented as *slicing trees* [Sarrafzadeh 1996]. For *heterogeneous* resource requirements, where each resource type has specific location on the board, shapes cannot be generated from the resource requirement vector ρ_m directly. A uniform entity, termed as *basic tile* and defined earlier in Definition 6.4 of Chapter 6, is used to compute the resource requirement of each module and is used for generation of shapes during node sizing.

Once the basic tile is imposed on the given target architecture, the chip is considered to be composed of $W_t \times H_t$ basic tiles arranged in H_t rows and W_t columns. In Figure 7.2, the basic tile $A = (80, 1, 1)$ consists of 20×4 CLBs, 1 RAM and 1 MUL. The entire architecture (Spartan-3 XC3S5000) in Fig. 7.2 can be covered by 26 rows and 4 columns of basic tile A.

7.3 Phase I: Generation of partition trees

In order to minimize the wirelengths over all *instances*, we obtain a *linear arrangement* of modules, taking the left to right order of the leaves of a *partition tree* obtained by recursive partitioning of module netlist. For each *instance* of the given *schedule*, we use a balanced min-cut bi-partitioning tool *hMetis* [hMetis] to partition the modules of a netlist (represented as *hypergraphs*) by an extension of the partitioning method described in Section 6.3 of Chapter 6. The *partition trees* generated in this phase serve as the baseline of *slicing tree* generation in the next phase.

As *static* modules must have the same shape and location across all *instances*, it is beneficial to place all the *static* modules at two diagonally opposite corners of

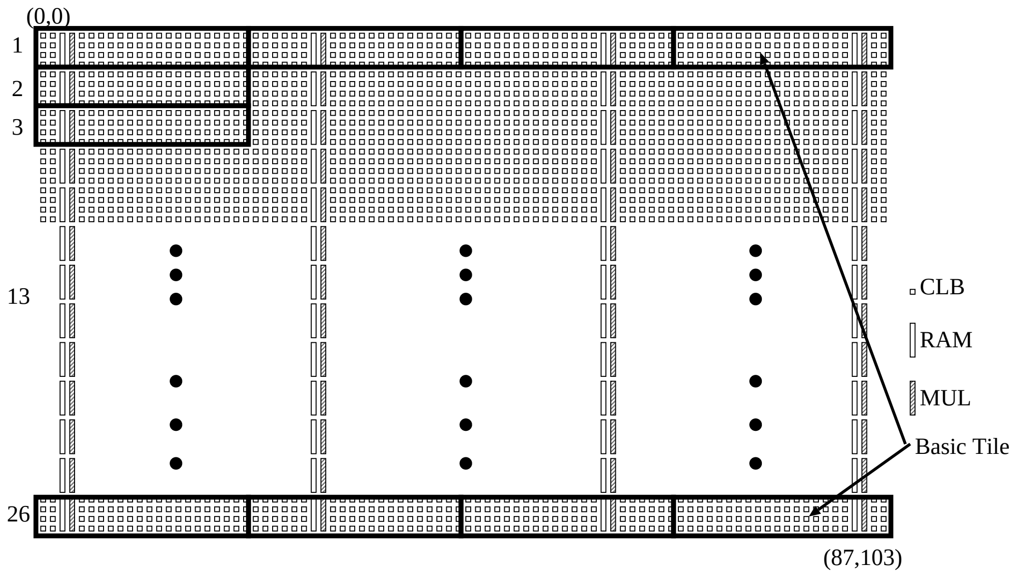


Figure 7.2: Spartan-3 XC3S5000 FPGA Architecture, tessellated with a *basic tile*, indicated by a rectangle of 4 rows and 20 columns of CLBs and 1 pair of RAM-MUL blocks (Figure 6.1 reproduced for convenience).

the floorplan. This provides the maximal contiguous space to place the rest of the *dynamic* modules.

Observation 7.1 *In a slicing tree representation of a floorplan, the modules at the left-most and the right-most leaves always correspond to the two diagonally opposite corners, the bottom-left and the top-right corner respectively, of the floorplan.*

Proof: By convention of the recursive construction of *slicing tree*, the module at left child of a node always goes below the horizontal cut and the right child of a node above the cut. Similarly, for a vertical cut, a module at left child of the node goes to the left of the cut line and the right child goes to the right of the cut. As a result, during bottom-up construction of the *slicing tree*, the module corresponding to the left-most leaf node is always at the left of all other subtrees of the nodes which lie in the path from root to this left-most leaf node due to vertical cuts. This left-most node is below all the nodes due to horizontal cut and to the left of all other nodes due to vertical cut. Similarly, for the right-most leaf, it is to the right of all the modules in the subtrees rooted at nodes in the path from the root to the right-most leaf and above them due to vertical and horizontal cut respectively. Figure 7.3 shows the conditions. □

From Observation 7.1, if the *static* modules are placed at the left-most and right-most ends of the *partition tree*, the modules will definitely be on the opposite

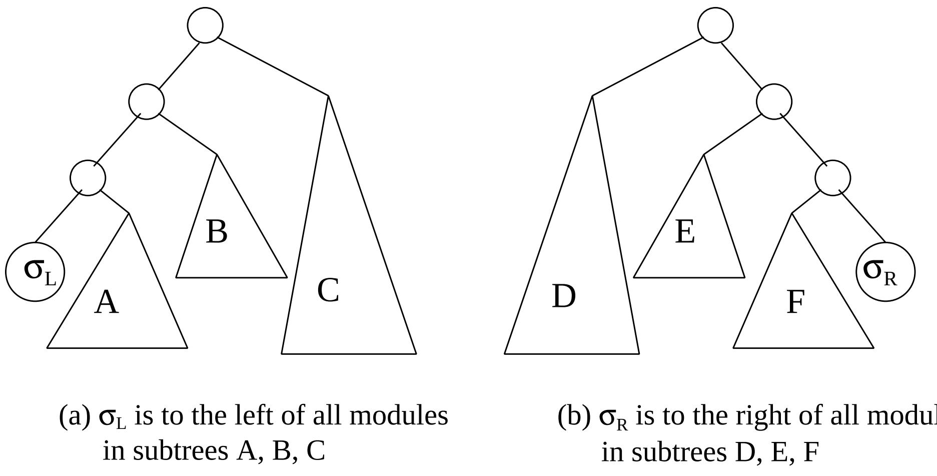

σ_L
A
B
C
(a) σ_L is to the left of all modules in subtrees A, B, C
D
E
F
σ_R
(b) σ_R is to the right of all modules in subtrees D, E, F

Figure 7.3: Modules at the left-most and the right-most leaves go to the bottom-left and top-right corner of the floorplan.

corners, i.e., at the bottom-left and top-right corners on the floorplan. If the netlist of modules of each *instance* is considered separately for *linear arrangement*, the *static* modules may go anywhere in the *linear arrangement* of each *instance*. We apply a constraint to the *linear arrangement* problem for each *instance* such that, in every *instance*, the positions of *static* modules are same in the *partition tree* and thus in the *linear arrangement*.

First we extract the *static* modules and the corresponding netlist from the given *schedule*. Then, we bi-partition the *static* modules into two partitions σ_L and σ_R and call each of them a *super module*. For each *instance*, the two super modules along with *dynamic* modules and their netlist is bi-partitioned recursively based on balanced min-cut until each partition contains at most one module/super module per partition. In the first level of recursive bi-partitioning, we force σ_L and σ_R to

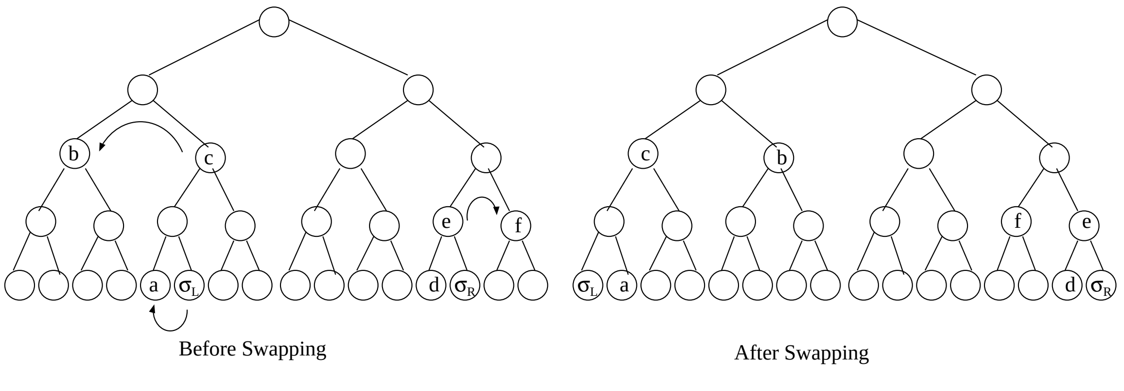


Figure 7.4: Swapping of *static* super modules to extreme ends of the *partition tree*; the arrow indicates the partitions to be exchanged.

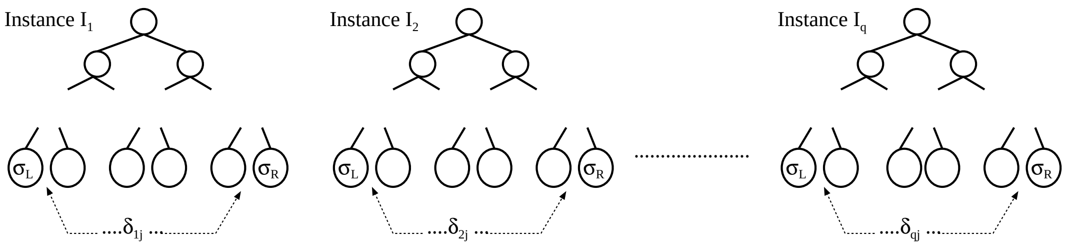


Figure 7.5: One *partition tree* for each *instance*.

be in different partitions for every *instance*, so that they can be pushed to extreme left and right positions respectively during further recursive partitioning. Since swapping of partitions in a *partition tree* does not affect the min-cut in the tree, during each recursive bi-partition, the left and right partitions are swapped such that partitions σ_L and σ_R are always pushed to the extreme left and extreme right of the *partition tree*. The swapping of partitions with *static* super modules is shown in Figure 7.4. Thus, we get one *partition tree* $\mathcal{B}_i, i = 1 \cdots q$, for each *instance* of the given *schedule* where the *static* super modules are at the extreme left and right leaves of each *partition tree* as shown in Figure 7.5.

Lemma 7.1 *The recursive bi-partitioning phase for an instance i takes $O(n_i)$ time, where n_i is the total number of modules to be partitioned in the i^{th} instance.*

Proof: The authors of *hMetis* [Karypis 1999b] claim that, although iterative, the time complexity of *hMetis* is almost $O(n)$, n being the number of hyperedges in the graph. Since, the number of hyperedges corresponding to each *instance* I_i is $O(n_i)$, the construction of the *partition tree* takes $O(n_i)$ time for each *instance*, where n_i is the number of modules in *instance* I_i. We perform a number of swaps to shift the *static* modules to the extreme left and right of the *partition tree* which takes $O(\log n_i)$ time if the tree is balanced. The number of swaps for unbalanced tree is $O(n_i)$, as the number of internal nodes is proportional to the number of leaf nodes. Thus the overall time complexity of the partitioning step is $O(n_i)$. $\square$

The first phase of the proposed method *PartialHeteroFP* is given in Algorithm 7.1.

7.4 Phase II: Global floorplan topology generation

In this step, a set of sliceable floorplan topologies is generated for each *instance* simultaneously by appropriate horizontal and vertical node sizing starting from a set of possible shapes (in terms of tiles) of each module.

Algorithm 7.1: PartialHeteroFP: Phase I

```
Input  : schedule of $q$ instances $I = \{I_1 \cdots I_q\}$, with static modules $SM$,
         dynamic modules $DM$ and netlist $\mathcal{S}_i$ for each instance $I_i$,
         list of module resource requirements $\rho_m$,
         FPGA architecture $(0, 0, W, H)$ and its basic tile $A$
Output: A feasible floorplan for each instance with all $\sigma_i \in SM$ having same
         position and shape
1  Preprocessing:
2  Compute $W_t$ and $H_t$ for target architecture with basic tile $A$;
3  for each module $m \in SM \cup DM$ do
4      Find minimum number of tiles required ($T_m$) from the resource
       requirement vector $\rho_m$;
5  Phase I: Generation of Partition Tree
6  Extract static modules and the netlist from each instance and partition them
   into two super modules, say $\sigma_L$ and $\sigma_R$;
7  for each instance $I_i$ do
8      Bi-partition the netlist of dynamic and super modules of the instance
       once such that $\sigma_L$ goes to left partition and $\sigma_R$ goes to the right;
9      Balanced min-cut recursive bi-partitioning of the netlist of modules (both
       dynamic and super modules) of an instance till there is only one module
       per partition;
10     $p \leftarrow$ parent of partition having $\sigma_L$;
11     while $\sigma_L$ is not at the extreme left of partition tree do
12         if $\sigma_L$ belongs to the right child of the subtree rooted at node $p$ then
13             Swap the left and right child of the subtree rooted at node $p$ ;
14         $p \leftarrow$ parent of node $p$;
15     $p \leftarrow$ parent of partition having $\sigma_R$;
16     while $\sigma_R$ is not at the extreme right of partition tree do
17         if $\sigma_R$ belongs to the left child of the subtree rooted at node $p$ then
18             Swap the left and right child of the subtree rooted at node $p$ ;
19         $p \leftarrow$ parent of node $p$;
   /* A template partition tree $\mathcal{B}_i$ for each instance $I_i$ for slicing
      topology generation                                                    */
```

As described in Section 6.4.1, a list of *irredundant shapes* are generated for each *dynamic* and the super modules by factorizing the tile requirement of the module T_m. Thus, each leaf node of the *partition tree* corresponding to a module contains a list of possible shapes, i.e., (width, height) pair in terms of tiles. For all *instances*, the corresponding *partition trees* are traversed simultaneously bottom-up, level by level, generating a set of irredundant sub-floorplans by combining the available shapes of its left and right children with vertical or horizontal cut as described in Section 6.4.2

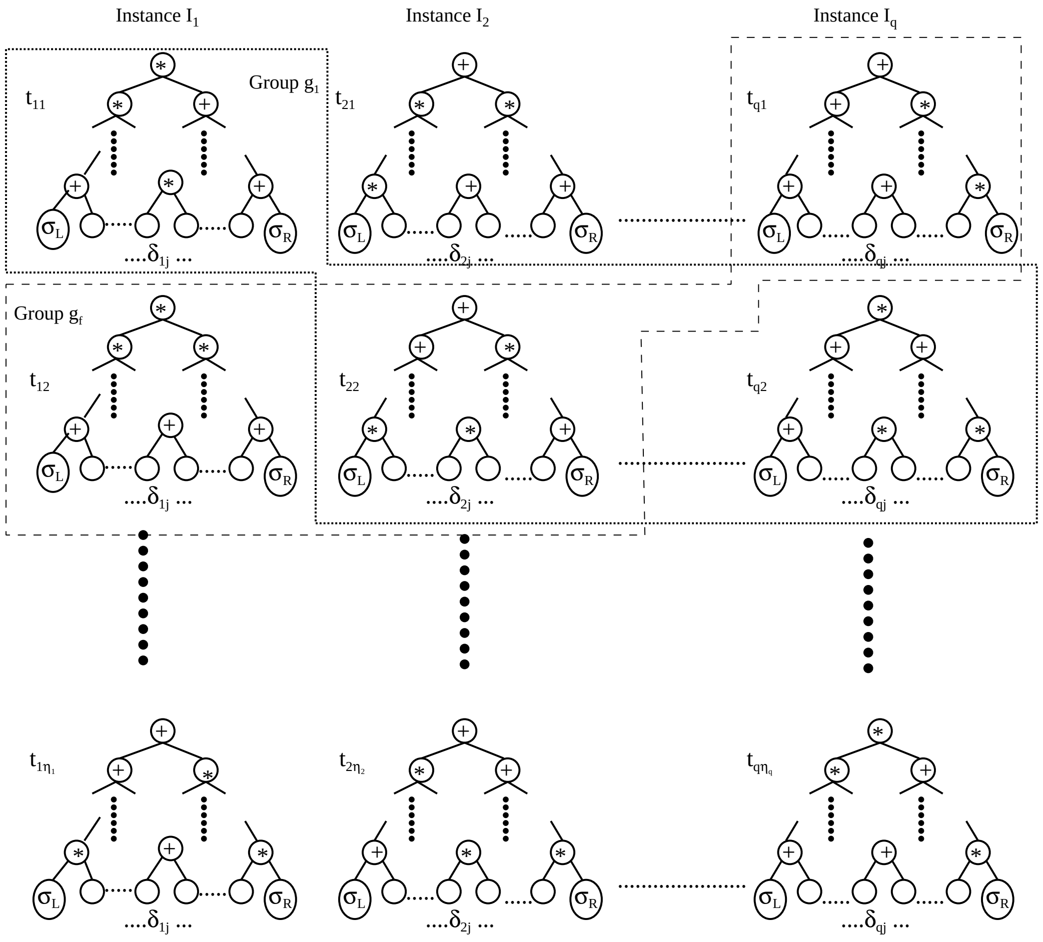


Figure 7.6: Set of *slicing trees* for each *instance*; vertical and horizontal cuts are denoted by * and $+$; the dotted polygon indicated the *group* g_1 of *slicing trees* and the dashed polygon, another *group* g_f.

of Chapter 6.

At the end of this phase, a set of *slicing trees* $\mathcal{F}_i = \{t_{ij} | i = 1 \cdots q, j = 1 \cdots \eta_i\}$ for each *instance* i, is generated with *static* super modules at two opposite corners of the floorplan as shown in Figure 7.6. A t_{ij} corresponds to the j^{th} *slicing tree* of i^{th} *instance*, and η_i is the number of *slicing trees* generated for i^{th} instance. The final shape of the floorplan as a result of node sizing, is stored at the root of each *slicing tree.* These final shapes (width, height) may not fit the target FPGA chip when the shapes are considered in terms of tiles. Here, the target chip is 4×26 tiles. All shapes that are either wider or longer than the target chip may not be useful to satisfy the exact requirements of all modules as described at the end of Section 6.4.2. Thus we select only those *slicing trees* with final shape of width between 3 and 6 as there is a high possibility of obtaining a feasible floorplan on this target

Algorithm 7.2: PartialHeteroFP: Phase II

```
1 Phase II: Generation of Floorplan Topologies
  Input  : Set of partition trees
  Output: Set of slicing trees $\mathcal{F}_i$ generated for each instance $I_i$
2 for each instance $I_i$ do
3     for each module do
4         Enumerate shapes $(w, h)$ by factorizing $(T_m)$;
5     Determine slicing topology and shapes by vertical and horizontal node
      sizing by post-order traversal of the partition tree $\mathcal{B}_i$ of Phase I.
```

board for those shapes in the third phase.

The second phase of the proposed method *PartialHeteroFP* is given in Algorithm 7.2.

Lemma 7.2 *The time complexity of slicing tree generation for all instances is $O(\sum_{i=1}^{q} \kappa {n_i}^2)$, where κ is the maximum number of shapes generated for any module.*

Proof: By Theorem 6.1, the number of *slicing trees* generated in an *instance* is $O(\kappa n^2)$. So, the total number of *slicing trees* generated for the q *instances* is $O(\sum_{i=1}^{q} \kappa {n_i}^2))$, κ and n_i being the maximum number of shapes generated for a module and the number of modules in *instance* I_i respectively. Hence, the result for q *instances.* □

7.5 Phase III: Realization of slicing trees on the chip

For the selected *slicing trees* $\mathcal{F}_i = \{t_{ij}\}$ of each *instance* i, coordinate positions are assigned to each module and *static* super modules respecting the cut lines and satisfying the exact CLB and RAM/MUL requirements. One or more solutions are generated in this phase. Each solution consists of one feasible floorplan for each *instance* in the *schedule.*

7.5.1 Allocation of rectangular region to a module

Each selected slicing tree of every *instance* is traversed top down and a rectangular region is assigned to every node using the cut direction at its parent and the number of CLBs required at that node. The root node of the slicing tree corresponds to the target board with 80 CLB columns and 104 CLB rows in it. Let the region $x_p^{min}, y_p^{min}, x_p^{max}, y_p^{max}$ allocated to a parent node p contain p_{row} rows and

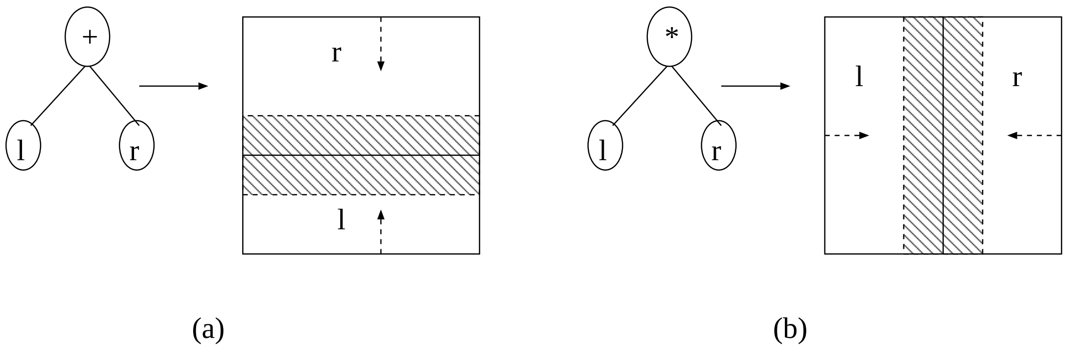


Figure 7.7: The convention used for allocation of CLBs for (a) vertical cut (b) horizontal cut; the shaded part shows the *free regions*; unshaded part represents the allocated region; '+' denotes *horizontal cut* and '*' denotes *vertical cut.*

p_{col} columns of CLBs. Similarly, we can define these terms for the left child l and the right child r of p. Let the CLB requirements at node p, its left child l and its right child r be p^{clb}, l^{clb} and r^{clb} respectively. The rectangular region assigned to the left child l of parent p is determined by the following equations (which are slightly different from the ones of *GARR* in Section 6.5.1).

For a vertical cut at p,

- $l_{col} = \frac{l^{clb}}{p_{row}}$; $r_{col} = \frac{r^{clb}}{p_{row}}$; $l_{row} = r_{row} = p_{row}$,
- $x_l^{min} = x_p^{min}$; $y_l^{min} = y_p^{min}$,
- $x_l^{max} = x_p^{min} + l_{col} - 1$; $y_l^{max} = y_p^{max}$

For a horizontal cut at p,

- $l_{row} = \frac{l^{clb}}{p_{col}}$; $r_{row} = \frac{r^{clb}}{p_{col}}$; $l_{col} = r_{col} = p_{col}$,
- $x_l^{min} = x_p^{min}$; $y_l^{min} = y_p^{min}$,
- $x_l^{max} = x_p^{max}$; $y_l^{max} = y_p^{min} + l_{row} - 1$

The coordinate positions of the rectangle for the right child r is computed in a manner similar to that for the left child l. Essentially, the vertical cut line is positioned by counting the CLB columns from left to right for the left child, and right to left for the right child. Similarly for horizontal cut, it is positioned by counting the rows from bottom to top for the left child, and top to bottom for the right child as shown in Figure 7.7. Although Figure 7.7 shows only *free regions*, there can also be overlapping regions if the resource requirement is not met. Positioning of cut lines in this fashion generates two types of regions:

- two *allocated regions* corresponding to two modules at opposite sides within the rectangle assigned to the parent node

- an *overlapping* or *free region* at the middle of the parent region.

The *overlapping region* is generated when a column or row has to be shared by both modules and a *free* rectangular region is generated when resource requirement of both modules are much less than the available resources in the parent region. We allocate the CLBs required by a module to the *allocated region* of the rectangles assigned to the corresponding module. The remaining CLB requirement of a module, called *deficit*, has to be satisfied either by the overlapping rectangle or by the neighboring rectangles assigned to other modules. This deficit of each module (if any exists), is satisfied during post processing described later in Section 7.5.4.

Thus, a set of floorplans are generated corresponding to each selected slicing tree of each *instance* by allocation of rectangular regions to each module / super module satisfying the CLB requirements either completely or partially. The conflicts for CLB requirements of more than one module need to be resolved in the overlapping or free rectangles.

As given in Lemma 7.2, the time complexity of finding the rectangular realization is also $O(\sum_{i=1}^{k} \kappa {n_i}^2)$.

7.5.2 Pruning the set of slicing trees

While allocating the rectangular regions to modules in different *instances*, their RAM/MUL requirements are not considered. To check whether the RAM/MUL requirement of each module and super modules are satisfied within the rectangular region allocated, we define the following condition.

Definition 7.2 (Major Violation:) *If a module has RAM/MUL requirement and has been assigned a rectangular region such that no RAM/MUL column passes through it, then the module is said to have the major violation.*

We discard all the floorplans from each *instance* if there is at least one module with major violation. These floorplans are discarded because a module with major violation has to borrow all its required RAM/MUL resources from its neighboring regions allocated to different modules. This may make a module non contiguous and the shape of the module can be severely affected.

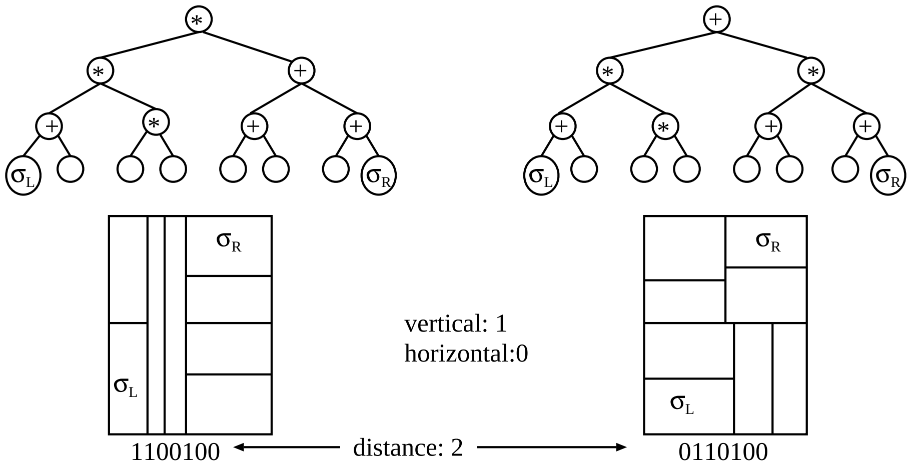


Figure 7.8: Distance between two *slicing trees.*

7.5.3 Grouping of slicing trees for global floorplan

The floorplans in the pruned set for each *instance* of the given *schedule* have *static* modules placed in the same topological location but may not necessarily have the exact same shape after the rectangular region allocation. Thus, the question of selecting a single floorplan for each of the *instances* arises, where not only the position but the shapes of each *static* modules match. So, we find a set of groups $\mathcal{P} = \{g_f\}$, where $g_f = \{t_{ij} | i = 1 \cdots q, j \in 1 \cdots \eta_i, \text{each } i \text{ is distinct}\}$. In other words, each *group* g_f consists of a single tree from $\mathcal{F}_i$ corresponding to each *instance* and the floorplans in each *group* are similar with respect to their cut lines or aspect ratios of *static* modules (vide Figure 7.6).

Definition 7.3 (Distance between two slicing trees:) *Let a and b be the strings representing level order traversal of nodes from root till the last but one level (i.e., the level above that of the leaves) of two given slicing trees respectively, with horizontal (vertical) cut represented as* 0 *(*1*). Let* $\lambda = min\{length(a), length(b)\}$ *and the length of the longer string be truncated till* λ *from right. Then, the distance between these two trees is the number of ones in* $a \oplus b$*,* $\oplus$ *denoting the exclusive-or operation.*

This measures the *closeness* among two *slicing trees* in terms of slicing topology as shown in Figure 7.8. It may be argued that, two *slicing trees* may become equivalent by flipping the subtrees of the *slicing trees.* But, in case of *PartialHeteroFP*, flipping the subtrees may change the positions of the *static* modules. Thus, oriented *slicing trees* are considered here rather than the equivalent slicing tree obtained by flipping its subtrees. In the context of *partial reconfiguration*, a *schedule* implies the ordering of the *instances* on the time line. To have same shapes of *static* modules

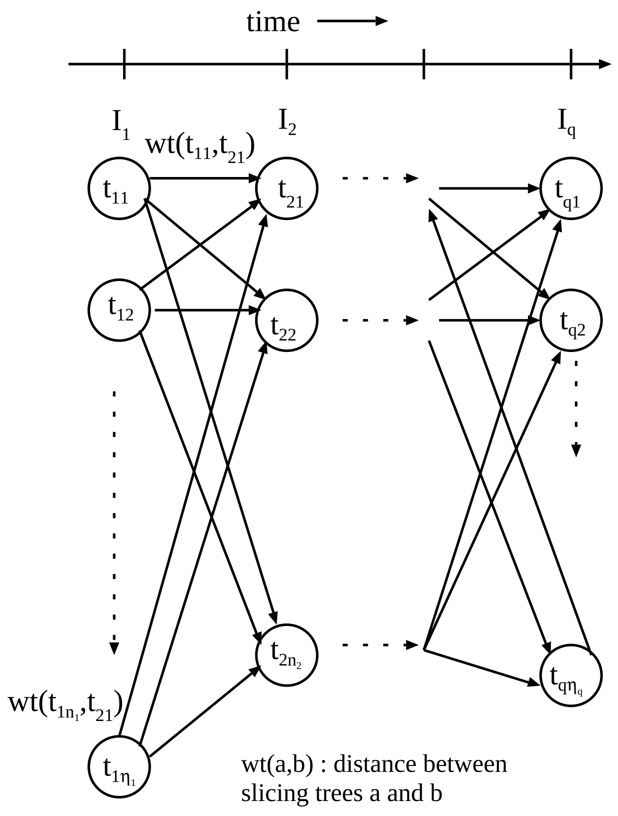


Figure 7.9: *Grouping* of *slicing trees* by finding shortest path in the associated digraph.

from one *instance* to its consecutive one, the change in slicing tree must be minimum. Let $\mathcal{F}_1 < \mathcal{F}_2 < \cdots \mathcal{F}_q$ be the q sets of *slicing trees* for the q *instances* in a given *schedule*, where the set $\mathcal{F}_1$ and $\mathcal{F}_q$ have the trees for the first and the last *instances* in the *schedule*. The '$<$' sign denotes the sequence of the *instances* in a *schedule*. An associated *distance digraph* $G_d = (V, E)$ is defined as follows:

- vertex $v \in V$ corresponds to a slicing tree t_{ij} for j^{th} tree of *instance* i as shown in Figure 7.9.
- there exists a weighted edge $e \in E$, between $u, v \in V$, if u and v corresponds to the *slicing trees* in consecutive *instances*, i.e., u and v correspond to $\mathcal{F}_i$ and $\mathcal{F}_j$ respectively while the relationship $\mathcal{F}_i < \mathcal{F}_{i+1}$ holds.
- the weight is the distance between u and v as in Definition 7.3.

If there are q *instances*, we find a *minimum weighted path* of length $q-1$, starting from a vertex corresponding to t_{1j}. There may be more than one such minimum weighted path. Each of them corresponds to a *group* and hence yields a *global* *floorplan.* A minimum weighted $(q-1)$ length path is obtained by keeping a heap with the edge weights at every node and selecting the minimum weighted edge from the node till there is no more edge to traverse.

7.5.3.1 Complexity of grouping the set of slicing trees for q instances

Lemma 7.3 *The construction of graph $G_d = (V, E)$ takes $O(\sum_{i=1}^{q-1} l_{i,i+1} \cdot \eta_i \cdot \eta_{i+1})$, where $l_{i,i+1} = min\{length(t_{ij}), length(t_{(i+1)j})\}$, where $length(t_{ij})$ represents the length of the string corresponding to slicing tree t_{ij}.*

Proof: There exists a vertex v corresponding to each slicing tree t_{ij} of each *instance* i. Let there be $t_{i1}, \cdots, t_{i\eta_i}$ *slicing trees* in *instance* I_i. There is an edge from each vertex corresponding to t_{ij} to all vertices $t_{(i+1)k}$, where $j = 1 \cdots \eta_i$ and $k = 1 \cdots \eta_{i+1}$. The i^{th} *instance* has η_i *slicing trees* and $(i+1)^{th}$ *instance* has η_{i+1} *slicing trees.* The number of edges in the graph is $\sum_{i=1}^{q-1} \eta_i \cdot \eta_{i+1})$ and the number of vertices is $\sum_{i=1}^{q} \eta_i$. The edge weight is the distance between the *slicing trees* corresponding to the end vertices of the edge. This distance is calculated by truncating the longer string corresponding to the tree to the length of the shorter string. Thus for each edge of G_d, $l_{i,i+1} = min\{length(\eta_i), length(\eta_{i+1})\}$ is computed. Hence the time complexity of the graph formation is $O(\sum_{i=1}^{q-1} l_{i,i+1} \cdot \eta_i \cdot \eta_{i+1})$. □

Note that, this graph G_d is constructed only once before the post-processing step of satisfying the remaining resource requirements, discussed in Section 7.5.4.

Lemma 7.4 *The number of groups generated from G_d is $O(\kappa.n_1^2)$ when all edge weights of the graph are distinct and $O(\left(\frac{\sum_{i=1}^{q} \kappa n_i^2}{q}\right)^q)$ when all the edges have same weights.*

Proof: If all the edge weights of the graph G_d are distinct, the number of distinct paths from each vertex corresponding to each tree of *instance* 1 to the vertices corresponding to q^{th} *instance* is simply the number of vertices corresponding to the trees t_{1j} for *instance* I_1 in graph G_d. Thus, in the best case, the number of groups generated is the number of *slicing trees* generated for *instance* I_1, i.e., $O(\kappa.n_1^2)$.

In the worst case, all edge weights are same. If η_i is the number of vertices corresponding to each *instance*, then the number of q length paths is $\Pi_{i=1}^{q} \eta_i$ each representing a *group.* Let AM and GM denote the arithmetic and geometric mean

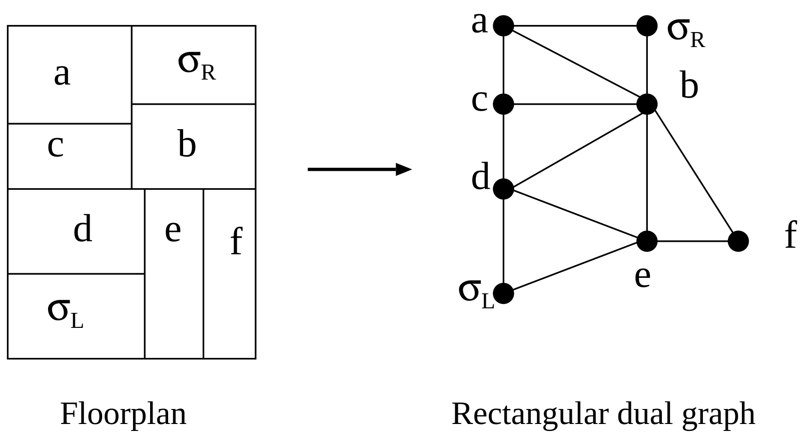


Figure 7.10: Rectangular dual graph (RD) corresponding to a floorplan.

respectively of a set of q values. Then,

$$\begin{aligned}
GM &\leq AM \\
or, (\Pi_{i=1}^{q}\eta_i)^{\frac{1}{q}} &\leq \frac{\sum_{i=1}^{q}\eta_i}{q} \\
or, \Pi_{i=1}^{q}\eta_i &\leq \left(\frac{\sum_{i=1}^{q}\eta_i}{q}\right)^q \\
or, \Pi_{i=1}^{q}\eta_i &\leq \left(\frac{\sum_{i=1}^{q}\kappa n_i^2}{q}\right)^q
\end{aligned}$$

Thus, the result follows. □

7.5.4 Postprocessing for satisfying resource requirements

The *slicing trees* selected for each *instance* in a *group* have *static* modules with nearly equal aspect ratios but not exactly the same shape. We consider all pair of shapes, taking one from the list of σ_L and the other from σ_R. For all *instances*, we impose the respective shape in a shape pair to *static* modules at bottom-left and top-right corner of the floorplan. This requires reallocation of CLBs of some of the *dynamic* modules which are neighbors of *static* modules in each floorplan due to new overlaps generated by imposition of the exact shape of *static* modules in each floorplan. We next reallocate CLBs of these *dynamic* modules along with the deficits generated earlier (described in Section 7.5.1) by utilizing the free regions available on the chip. We formulate a minimum cost maximum flow (MCMF) formulation for each floorplan in a *group* to resolve all the deficits in CLB requirements of modules in the floorplan.

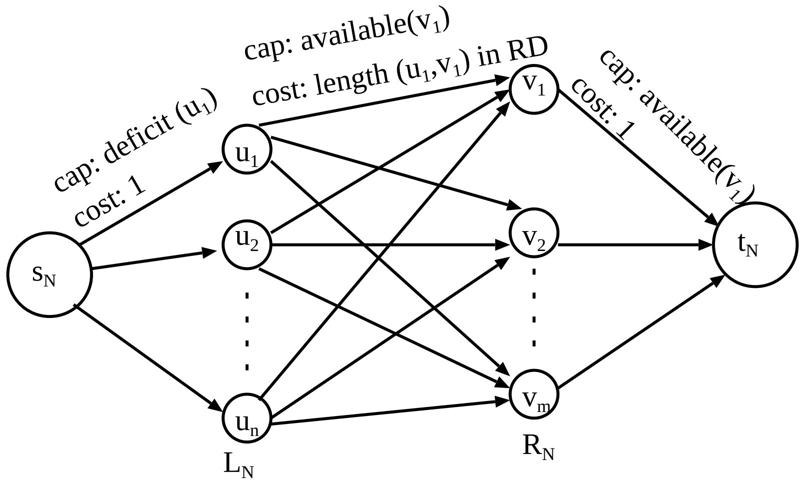


Figure 7.11: Postprocessing by *min-cost max-flow* for satisfying CLB requirements.

A network flow graph $N = (V_N, E_N)$ is generated for each floorplan of the *group* as follows. Here, N is a bipartite graph with a *source* node s_N and a *sink* node t_N. Let $V_N = L_N \cup R_N$, and $L_N \cap R_N = \phi$. Each $v \in L_N$ corresponds to a module that has deficit of CLBs. Each $v \in R_N$ corresponds to the rectangular region if it has any free CLBs, described in Section 7.5.1. For each floorplan, a *rectangular dual graph RD* [Sarrafzadeh 1996] is generated from the adjacency relationship of rectangles as shown in Figure 7.10. Let $E_N = E_s \cup E_{uv} \cup E_t$.

- For each $u \in L_N$, there exists an edge $e \in E_s, e = (s_N, u)$ with capacity as the short-fall in the requirement of CLBs of a module, i.e., the deficit, corresponding to u and cost as 1.
- For each $v \in R_N$ there exists an edge $e \in E_t, e = (v, t_N)$ with capacity as the number of free / unallocated CLBs in the rectangle corresponding to v and cost as 1.
- For each $u \in L_N$, and for each $v \in R_N$, there exists an edge $e \in E_{uv}$ with capacity equal to the number of free CLBs in the rectangle corresponding to v. The cost is the length of the shortest path in RD from the vertex in RD corresponding to u to the vertex in RD corresponding to v.

Figure 7.11 shows one such network flow graph. By solving MCMF, if the amount of flow is equal to the total deficit of CLBs, then these deficit CLBs corresponding to each $u \in L_N$ is satisfied by its neighboring rectangles. If there is a positive flow f in an edge $e = (u, v) \in E_{uv}$ having cost c, then the module corresponding to u borrows

f CLBs from the rectangle corresponding to v following a c length path in the graph RD, from vertex u to vertex v. This results in rectilinear shape of a module in a floorplan. If MCMF does not have a solution for any one of the floorplan in a *group*, this *group* is rejected as a candidate solution for the *partial reconfiguration* problem.

Lemma 7.5 *The time complexity of solving MCMF for each group is*

$$O(\sum_{i=1}^{q} n_i^4 \log n_i + n_i^3 (\log n_i)^2)$$

and for all the groups it is

$$O(\sum_{j=1}^{g} \sum_{i=1}^{q} n_i^4 \log n_i + n_i^3 (\log n_i)^2),$$

where g is the total number of groups.

Proof: Each flow graph $N = (V_N, E_N)$ has $|V| = O(n_i)$ and $|E| = O(n_i^2)$ corresponding to the *instance* I_i. The time complexity of the MCMF is $O(|Z| \log |U| (|Z| + |U| \log |U|))$ [Ahuja 1993], where $|Z|$ and $|U|$ are the number of edges and vertices in the flow graph. Thus, for a *group* the time complexity is

$$O(\sum_{i=1}^{q} n_i^4 \log n_i + n_i^3 (\log n_i)^2).$$

The overall time complexity to solve the MCMF for all g groups is

$$O(\sum_{j=1}^{g} \sum_{i=1}^{q} n_i^4 \log n_i + n_i^3 (\log n_i)^2).$$

□

Finally, the RAM/MULs of each module are allocated by minimum weighted bipartite matching (MWBM) formulation as described in Section 6.5.2 for that *group* of floorplans, where each floorplan is feasible in terms of CLBs. This produces the final floorplans for each *instance* in the *partial reconfiguration* problem. Since there may be more than one *group* with feasible solution, we choose a *group* with feasible floorplans of all *instances* having the minimum sum of HPWL over all *instances*.

Theorem 7.1 *The time complexity of PartialHeteroFP is $O(n^{2q+4} \log n)$, when the edge weights of the distance graph G_d are all same and it is $O(n_1^2 n^4 \log n)$ when G_d has distinct edge weights, n_1 being the number of modules in the first instance, n,* *the total number of modules across all instances, and q the number of instances in a schedule.*

Proof: Solving the MCMF is the dominant computation compared to the slicing tree generation, grouping, graph formation for our method *PartialHeteroFP*. From Lemma 7.5, the time complexity of solving the MCMF for a *group* is $O(n^4 \log n)$. From Lemma 7.4, there are $O(n^{2q})$ groups when distance graph G_d has same edge weights. Hence, the total time complexity of the method is $O(n^{2q+4} \log n)$ for this case. With distinct edge weights of G_d, the number of groups is $O(\kappa n_1^2)$, n_1 being the number of modules in the first instance. Hence, the time complexity of *PartialHeteroFP* is $O(n_1^2 n^4 \log n)$. □

With q being a constant for all practical purposes, the time complexity is a polynomial in n. Algorithm 7.3 gives the third phase of our method *PartialHeteroFP*.

7.6 An example

The three phases of our floorplanner *PartialHeteroFP* are given in Algorithms 7.1, 7.2 and 7.3 respectively. The method proposed in this chapter is illustrated through an example here. We consider a synthetic benchmark that fits on a Spartan 3 Xilinx FPGA chip XC3S5000. This benchmark has two *instances* 0 and 1 with 26 and 20 modules respectively. It has 4 *static* modules numbered $0, \dots, 3$. *Instance* 0 has the *static* modules $0, \dots, 3$ and *dynamic* modules numbered from 4 to 25 and *instance* 1 has *dynamic* modules 26 to 41 with the same *static* modules. *Instance* 0 requires 8141 CLBs, 84 RAMs and 84 Multipliers while *instance* 1 requires 8226 CLBs, 83 RAMs and 83 Multipliers.

We partition the *static* modules into two partitions, called super modules. The modules numbered 0 and 1 form one partition, and the ones numbered 2 and 3 form another. σ_R contains modules 0 and 1 while σ_L contains modules 2 and 3. Next, we partition the *dynamic* modules of each *instance* by the balanced min-cut bi-partitioner recursively, fixing the *static* modules to their respective partitions as described in Section 7.3. The left to right order of the leaves of the *partition tree* gives the *linear arrangement* of modules as shown in Figure 7.12.

Then we generate a list of irredundant shapes (width, height) for each of the module from the requirements in terms of basic tiles. Here, both the *static* super modules require 5 tiles. The shapes generated are $1 \times 5, 5 \times 1$. Two more shapes $2 \times 3, 3 \times 2$ are also included. We build up the *slicing trees* for both the *instances* as described in Section 7.4. Out of the many *slicing trees* generated in this phase

Algorithm 7.3: PartialHeteroFP: Phase III

```
1  Phase III: Realization of Slicing Tree on FPGA
2  for each instance $I_i$ do
3      for each slicing tree do
4          Allocate rectangular region to modules satisfying CLB requirement ;
5      Discard floorplans not satisfying the RAM/MUL requirements;
6  Grouping slicing trees across instances:
7  Construct the directed graph $G_d = (V, E)$ to find similar trees from different
   instances;
8  Find a minimum weighted path of length $q - 1$ starting from the nodes
   corresponding to $I_1$ to $I_q$;
   /* Output: list of groups each having one slicing tree from each
      instance                                                        */
9  Processing for satisfying resource requirements:
10 for each group do
11     for all pair of shapes $\sigma_L$, $\sigma_R$ do
12         Impose a shape pair of $\sigma_L$, $\sigma_R$;
13     for the slicing tree corresponding to each instance do
14         Construct an adjacency graph corresponding to the floorplan with free and
           allocated regions;
15         for each module do
16             Find the shortest path from a vertex corresponding to a module with
               unallocated CLBs to a vertex corresponding to a free region;
17         Employ min-cost max-flow to satisfy the unresolved CLB requirements;
18         if there is no feasible flow then
19             continue with next pair of shapes
20         else
21             Allocate RAM/MULs to modules by min-cost max-flow (MCMF);
   /* for a group, more than one set of feasible floorplans corresponding
      to $q$ instances may be generated                               */
22 if the group has one or more set of q feasible floorplans then
23     report each set of floorplans as feasible solution with total wirelength across
       instances;
24 else
25     Discard that group;
   /* Choose the set of floorplans with minimum total wirelength      */
```

for each *instance*, one slicing tree for each *instance* is shown in Figure 7.12. The internal nodes represent the cuts used to join the child subtrees at parent node. The shapes, i.e. (width, height) pair, of the module/super module/internal node, are shown beside the corresponding node in the slicing tree. The shape at the root for both the trees is 4×28. Since this width equals that of the target chip, these *slicing trees* are possible candidates for generating feasible floorplans.

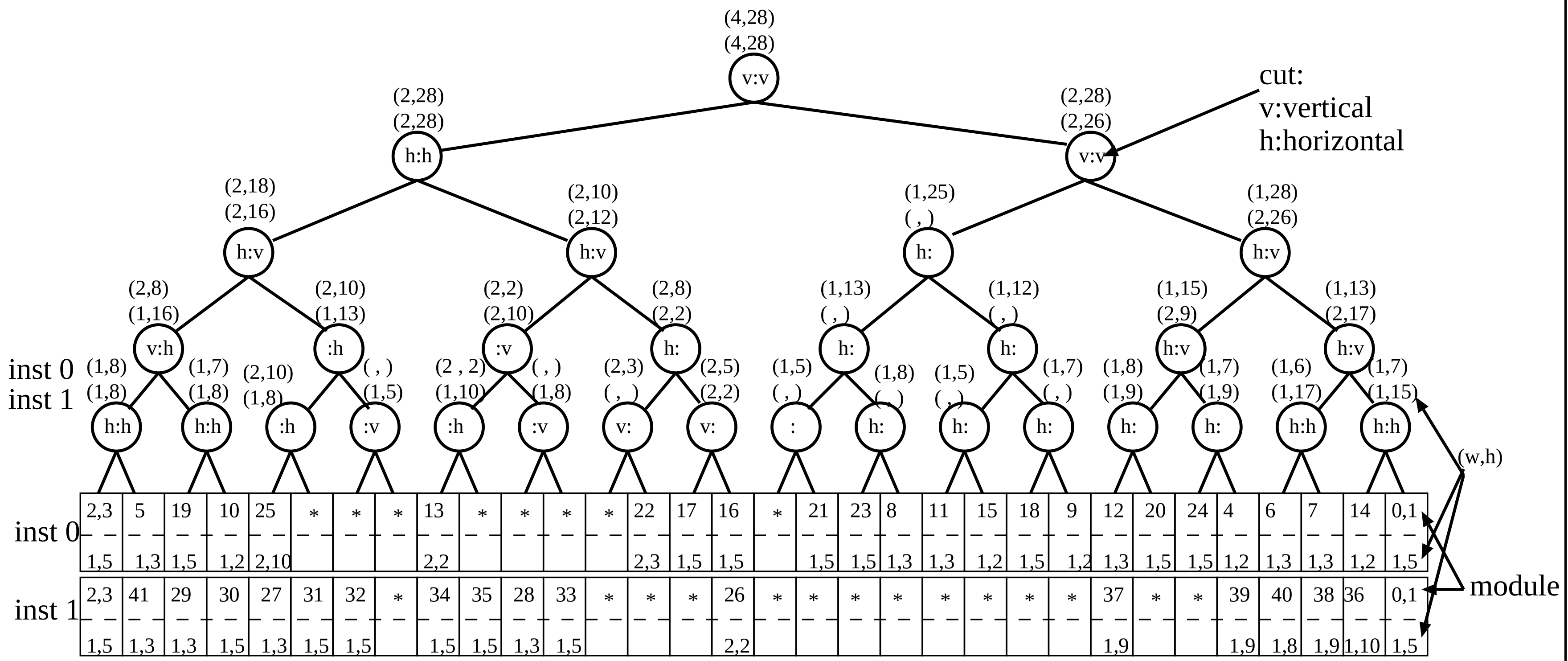
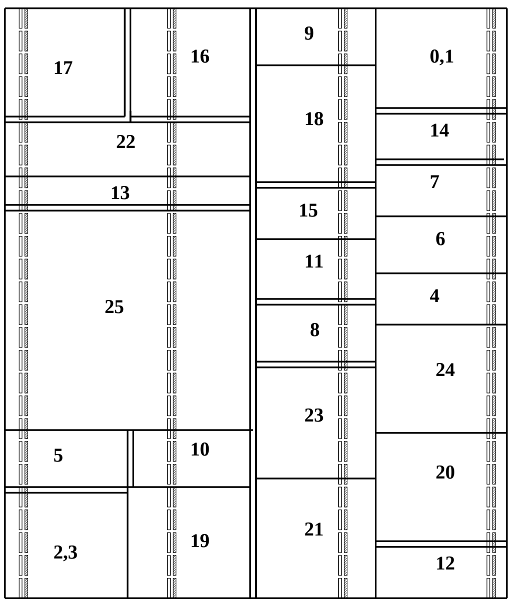
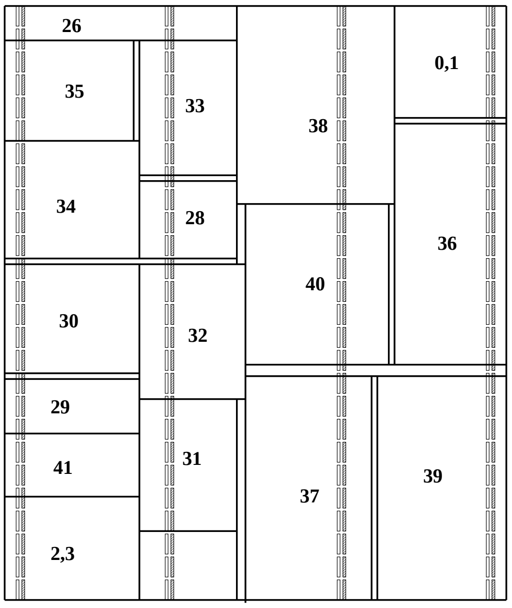


Figure 7.12: *Slicing trees* for two *instances*; *v* and *h* represent vertical and horizontal cut lines; the cut line directions for two *instances* are separated by a ':' at a node; shape of a module is given as (width, height) pair.



(a)



(b)

Figure 7.13: Floorplan after greedy allocation of CLBs: (a) *instance* 0, (b) *instance* 1.

The realization of the two floorplans corresponding to the *slicing trees* of Figure 7.12, after rectangular region allocation to each module, are shown in Figures 7.13(a) and 7.13(b). The rectangles with module numbers in the figures, show the non-overlapping regions, which may or may not have free CLBs, and the rectangular strips without any module numbers indicate the regions which are either unallocated (free) or shared by more than one module. The shapes of σ_L and σ_R, in terms of actual resources and not tiles, are respectively 19×19 and 21×18 in *instance* 0, and 21×18 and 19×20 in *instance* 1. Thus, we have four possible shape pairs for *static* modules for the above two *instances.* Imposing each of the shape pairs on the *static* modules may cause some overlap of CLBs with the neighboring modules. So, minor re-allocation is needed.

We choose the common shape for σ_L and σ_R to be 21×18 and 21×18 respectively and impose the shape on each floorplan for the *static* modules. A rectangular dual graph for each *instance* is drawn as shown in Figure 7.14(a) and 7.14(b), where numbered vertices correspond to the modules with same number and vertices labeled with letters correspond to the free or overlapped region. From these graphs, we draw a network flow graph described in Section 7.5.4 for both the floorplans shown. After solving MCMF as described in Section 7.5.4, we obtain the floorplans with all the CLB requirements satisfied. The final floorplan may contain modules with rectilinear shapes as shown in Figures 7.15(a) and 7.15(b) for the example *instances.*

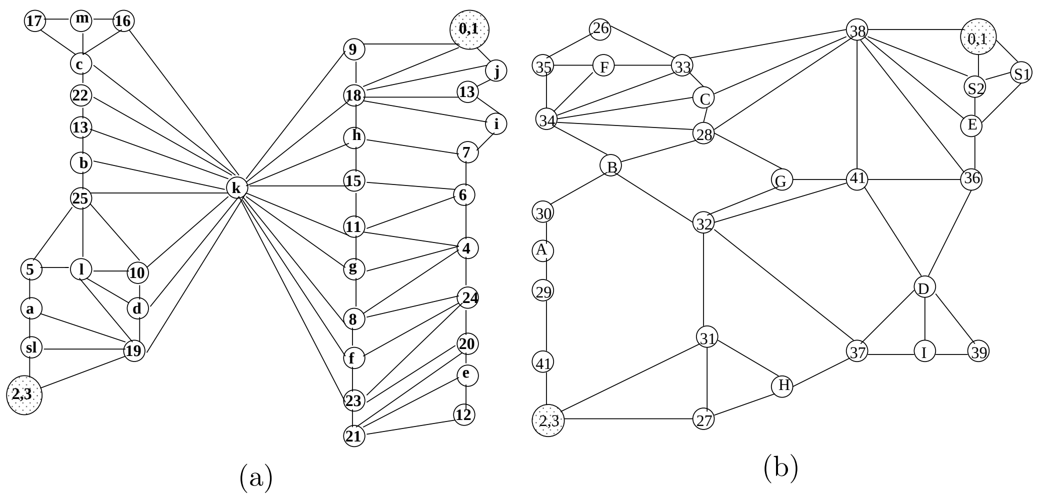


Figure 7.14: Rectangular dual graph for (a) *instance* 0 (b) *instance* 1; letters represent a rectangular region either unallocated or shared by two modules, and numerals denote the modules with deficit in CLB requirements within its allocated rectangular region.

7.7 Experimental results

We implemented the proposed method in C on Unix platform using *hMetis* [hMetis] and LEDA [LEDA] library on 1.2 GHz SunBlade 2000 workstation with SunOS Release 5.8. The method has been tested for 9 different synthetic benchmarks.

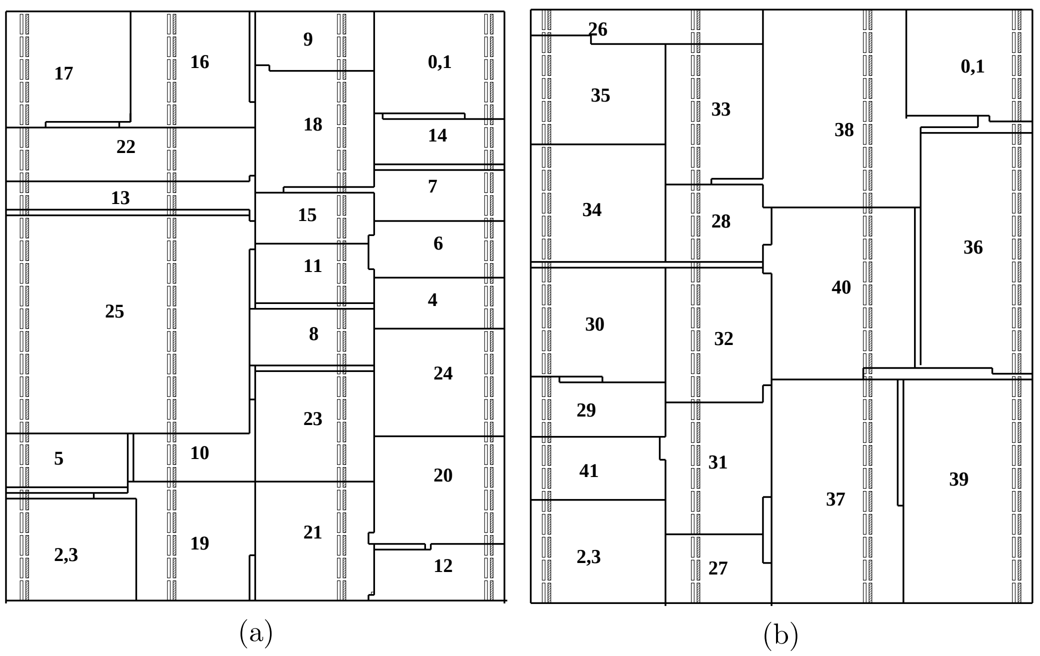


Figure 7.15: Final floorplan for (a) *instance* 0, (b) *instance* 1.

Table 7.1: Characteristics of benchmark; [SM: Static modules]

1	2	3	4
Benchmark	# instances	# SM	max # modules, nets
bench1	5	4	31,660
bench2	5	2	31,527
bench3	6	3	33,510
bench4	6	4	29,486
bench5	6	2	31,450
bench6	7	3	30,510
bench7	8	3	34,500
bench8	9	5	30,420
bench9	10	3	29,544

Table 7.2: Comparison of HPWL, CPU time and reconfiguration overhead; *Partial-HeteroFP* gives 100% overlap of SM

1	2	3	4	5	6	7
	HPWL			CPU time(s)		Overlap of SM (%)
Benchmark	Global	Indiv.	Avg. Inc.(%)	Global	Indiv.	Indiv.
bench1	268184	212094	26	6.5	5.3	2
bench2	186247	141391	31	6.6	5.0	0
bench3	285659	232910	22	7.9	6.5	9
bench4	223642	180048	24	7.6	5.8	8
bench5	264308	200501	31	7.9	5.9	1
bench6	308010	247481	24	8.9	6.9	7
bench7	330114	237856	38	9.8	8.3	7
bench8	379354	287604	31	11.4	9.2	1
bench9	326026	257314	26	12.2	8.6	17

Table 7.1 shows the number of *instances*, the number of *static* modules, maximum number of modules and signal nets in each benchmark in our experiment.

In Table 7.2, the total wirelength obtained by summing up the HPWL over all *instances* is compared with the total wirelength obtained if each individual *instance* is floorplanned independently. Column 4 shows the average increase in wirelength. Over the nine benchmark circuits, the average increase in wirelength is 28%, while the time taken to generate the global floorplans for all *instances* is 1.27× of the total time taken for floorplanning each of the *instances* individually. These CPU times are shown in columns 5 and 6 for global and individual floorplan generation respectively. Although the worst case time complexity of the proposed method is $O(n^{2q+4} \log n)$, the number of groups is very small in practice as evident from the time taken. Our global floorplan generation method places the *static* modules of same shape at the same location thereby yielding the overlap of *static* modules of consecutive *instances* to 100%. Whereas, in case of floorplanning individual *instances* consecutively, the overlap for *static* modules is very less, about 5.8% on the average, over nine benchmarks. This shows that, with a small increase in wirelength and running time, it is possible to generate a set of floorplans for a given *schedule* satisfying all its resource requirement and yet causing least *partial reconfiguration* overhead. This shows the suitability of our fast deterministic floorplanning method for *partial reconfiguration.*

7.8 Conclusion

In this chapter, we proposed a fast deterministic floorplanning method in the context of *partial reconfiguration* for FPGAs with *heterogeneous* resources. In order to reduce the reconfiguration overhead, the *static* modules are placed on the chip at a fixed location with same shapes for each *instance* of a given *schedule*, while the remaining contiguous space is used for placing the *dynamic* modules. Experiments on a set of benchmarks show that being a deterministic method, it is fast and it can generate feasible floorplans for each of the *instances* of each benchmark with a small increase in wirelength compared to the near-optimal floorplan of individual *instances.* Relaxing the constraint on positions of *static* modules is under study.

Chapter 8

Concluding Remarks

Contents

8.1 Summary of the contributions

In this thesis, we proposed a set of deterministic and fast placement methods for island-style FPGAs without sacrificing the quality compared to the existing methods of placement. First, three bottom-up greedy methods have been designed and implemented for placement of CLBs and IOBs. The key facets of these methods are ease of implementation and considerable speedup over the popular tool VPR, which serves as the yardstick for most researchers in this area. The quality of the placement solutions produced, assessed by bounding box cost, is sacrificed nominally. Hence, these methods are very useful for fast reconfiguration.

Next, we have also addressed top-down partitioning based placement by not only proposing a novel and fast algorithm employing space filling curves but also for the first time to the best of our knowledge, we have derived analytical upper bounds on the quality of the placement obtained by our approach in order to bridge the gap between theory and practice.

Since our placement methods perform very well for medium scale circuits, this can well be applied in modern FPGA architectures for placing the CLBs of a module in the islands of CLBs defined naturally by the pre-placed heterogeneous resources on the chip.

Owing to the need of the floorplanning step in the physical design phase for modern FPGAs with pre-placed heterogeneous resources, we have also proposed an unified floorplan topology generation and sizing method for a given netlist of modules which consist of several CLBs and other types of resources such as RAM, MUL blocks. The deterministic method is based on basic tile of the target FPGA architecture, and is very fast. It improves the total wirelength significantly compared to that produced by the handful of existing floorplanning methods for FPGAs.

Lastly, we have also designed a fast global floorplan generation method in the context of partial reconfigurability to place modules of a sequence of instances (tasks) in a schedule such that the total wirelength over all floorplans is minimized. Moreover, our algorithm places the modules which are common across all instances, at same fixed positions and having the same shape so that the reconfiguration overhead is also minimized.

8.2 Future directions

In this thesis, we restricted to the objective of optimizing the total half-perimeter wirelength throughout in the design of efficient floorplaning and placement methods. Our main focus was to develop faster methods without sacrificing the quality. With a variety of different FPGA architecture coming up, the simple half-perimeter wirelength metric may not suffice for the new architectures. Other parameters such as power, critical path delay, also need to be incorporated in an integrated framework of floorplanning and placement for the latest fabrication technology with diminishing feature size along with higher clock speeds.

In the context of floorplanning for modern FPGAs, the definition of a *basic tile* is going to be more complex with increasing irregularities in the FPGA fabric. This may require the use of more than one type of tile to handle the situation. Non-slicing mosaic and even general floorplan topologies may have to be considered which is likely to call for devising a different floorplan representation. For the partial reconfiguration case, constraints on the positions of common modules to the two diagonally opposite corners, or in fact to positions decided a priori, may have to be relaxed in order to improve upon the total wirelength of the floorplans.

Finally, instead of considering placement and floorplanning as a separate step in the design flow, an integrated framework for floorplanning and placement approach is possibly needed to achieve better quality of solution. Applicability of the methods proposed in this thesis to other array style architectures is also an issue worth pursuing. To mention a few very recent ones, there are Field-programmable Object Arrays (FPOAs) where the objects are small procesing elements, or even multi-core systems having cores which may be heterogeneous in functionality as well as in size and shape.

Bibliography

[Abramowitz 1972] M. Abramowitz and I. A. Stegun, editeurs. Handbook of mathematical functions with formulas, graphs, and mathematical tables. Dover, New York, 1972.

[Achronix] Achronix. *Achronix Semiconductor Corporation.* http://www.achronix.com/.

[Actel] Actel. *Actel Corporation.* http://www.actel.com/.

[Adolphson 1973] D. Adolphson and T. C. Hu. *Optimal Linear Ordering.* SIAM Journal Applied Math, vol. 25, no. 3, pages 403–423, 1973.

[Adya 2003] S. N. Adya and I. L. Markov. *Fixed Outline Floorplanning: Enabling Hierarchical Design.* IEEE Transaction on CAD of Integrated Circuits and Systems, vol. 11, no. 6, pages 1120–1135, 2003.

[Ahmadinia 2007] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich and J. C. van der Veen. *Optimal Free-Space Management and Routing-Conscious Dynamic Placement for Reconfigurable Devices.* IEEE Transaction on Computers, vol. 56, no. 5, pages 673–680, 2007.

[Ahuja 1993] R. Ahuja, T. Magnanti and J. Orlin. Network flows: Theory, algorithms and applications. NJ: Prentice Hall, Englewood Cliffs, 1993.

[Alexander 1998] M. J. Alexander, J. P. Cohoon, J. L. Ganley and G. Robins. *Placement and Routing for Performance-Oriented FPGA Layout.* VLSI Design, vol. 87, no. 1, pages 97–110, 1998.

[Alpert 1995] C. J. Alpert and A. B. Kahng. *Recent directions in netlist partitioning: A Survey.* Integration, the VLSI Journal, vol. 19, pages 1–81, 1995.

[Alsuwaiyel 1999] M. H. Alsuwaiyel. Algorithms: Design techniques and analysis. World Scientific, 1999.

[Altera] Altera. *Stratix II Device Handbook.* http://www.altera.com/literature/hb/stx2/stratix2.handbook.pdf.

[Altera 2003] Altera. *FLEX 10K embedded programmable logic device family,DS-F10K-4.2,.* http://www.altera.com/literature/ds/dsf10k.pdf, January 2003.

[Anderson 2000] J. H. Anderson, J. Saunders, S. Nag, C. Madabhushi and R. Jayaraman. *A Placement Algorithm for FPGA Designs with Multiple I/O Standards.* In FPL '00: 10th International Workshop on Field-Programmable Logic and Applications, pages 211–220, London, UK, 2000. Springer-Verlag.

[ARM] ARM. *ARM INC.* http://www.arm.com/.

[Asano 1997] T. Asano, D. Ranjan, T. Roos, E. Welzl and P. Widmayer. *Space-filling curves and their use in the design of geometric data structures.* Theoretical Computer Science, vol. 181, no. 1, pages 3–15, 1997.

[Bazargan 2000] K. Bazargan, R. Kastner and M. Sarrafzadeh. *Fast Template Placement for Reconfigurable Computing Systems.* IEEE Design and Test, vol. 17, no. 1, pages 68–83, 2000.

[Berg 2000] M. D. Berg, M. V. Kreveld, M. Overmars and O. Schwarzkopf. Computational geometry: Algorithms and application. Springer, 2000.

[Betz 1997] V. Betz and J. Rose. *VPR: A new Packing, Placement and Routing Tool for FPGA Research.* In Wayne Luk, Peter Y. K. Cheung and Manfred Glesner, editeurs, FPL '97: International Conference on Field-Programmable Logic and Applications, pages 213–222. Springer-Verlag, Berlin, 1997.

[Betz 1999] V. Betz, J. Rose and A. Marquardt. Architecture and CAD for deep-submicron FPGAs. Kluwer Academic Publishers, Feb 1999.

[Bhasker 1987] J. Bhasker and S. Sahni. *Optimal Linear Arrangement of Circuit Components.* Journal of VLSI & Computer Systems, vol. 2, pages 87–109, 1987.

[Breinholt 1998] G. Breinholt and C. Schierz. *Generating Hilbert's Space-Filling Curve by Recursion.* ACM Transaction on Mathematical Software, vol. 24, no. 2, pages 184–189, June 1998.

[Brown 1995] S. Brown, R. Francis, J. Rose and Z. Vranesic, editeurs. Field programmable gate arrays. The Springer International Series in Engineering and Computer Science, 1995.

[Chang 2000] Y. W. Chang and Y. T. Chang. *An architecture-driven metric for simultaneous placement and global routing for FPGAs.* In DAC '00: 37th Design Automation Conference, pages 567–572, New York, NY, USA, 2000. ACM.

[Chen 2006] D. Chen, J. Cong and P. Pan. *FPGA Design Automation: A Survey.* Foundation and Trends in Electronic Design Automation, vol. 1, no. 3, pages 139–169, 2006.

[Cheng 2004] L. Cheng and M. D. F. Wong. *Floorplan Design for Multi-Million Gate FPGAs.* In ICCAD '04: International Conference on Computer Aided Design, pages 292–299, 2004.

[Cheng 2006] L. Cheng and M. D. F. Wong. *Floorplan Design for Multimillion Gate FPGAs.* IEEE Transaction on CAD of Integrated Circuits and Systems, vol. 25, no. 12, pages 2795–2805, 2006.

[Cormen 2003] T. H. Cormen, C. E. Leiserson and R. L. Rivest. Introduction to algorithms. McGraw-Hill Publisher, Dec 2003.

[Dasgupta 1998] P. Dasgupta, S. Sur-Kolay and B. B. Bhattacharya. *A Unified Approach to Topology Generation and Optimal Sizing of Floorplans.* IEEE Transaction on CAD of Integrated Circuits and Systems, vol. 17, no. 2, pages 126–135, 1998.

[Eisenmann 1998] H. Eisenmann and F. M. Johannes. *Generic global placement and floorplanning.* In DAC '98: 35th annual Conference on Design Automation, pages 269–274, New York, NY, USA, 1998. ACM.

[Emmert 1998] J. M Emmert, A. Randhar and D. Bhatia. *Fast Floorplanning for FPGAs.* In FPL '98: International Conference on Field Programmable Logic and Applications, pages 129–138, 1998.

[Emmert 1999a] J. M. Emmert and D. Bhatia. *A Methodology for Fast FPGA Floorplanning.* In ACM/SIGDA International Symposium in FPGAs, pages 47–56, 1999.

[Emmert 1999b] J. M. Emmert and D. K. Bhatia. *Tabu Search: Ultra-Fast Placement for FPGAs.* In FPL '99: International Conference on Field Programmable Logic and Applications, pages 81–90, Berlin / Heidelberg, 1999. Springer.

[Even 2000] G. Even, J. S. Naor, S. Rao and B. Schieber. *Divide-and-conquer approximation algorithms via spreading metrics.* Journal of the ACM (JACM), vol. 47, no. 4, pages 585–616, July 2000.

[Farbarik 1997] R. Farbarik, X. Liu, M. Rossman, P. Parakh, T. Basso and R. Brown. *CAD Tools for Area-Distributed I/O Pad Packaging.* In MCMC '97: IEEE Multi-Chip Module Conference, page 125, Washington, DC, USA, 1997. IEEE Computer Society.

[Feige 2007] U. Feige and J. R. Lee. *An improved approximation ratio for the minimum linear arrangement problem.* Information Processing Letters, vol. 101, no. 1, pages 26–29, 2007.

[Feng 2004] Y. Feng, D. P. Mehta and H. Yang. *Constrained Floorplanning Using Network Flows.* IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 4, pages 572–580, 2004.

[Feng 2006] Y. Feng and D. P. Mehta. *Heterogeneous Floorplanning for FPGAs.* In VLSID 06': IEEE International Conference on VLSI Design, 2006.

[Fiduccia 1982] C. M. Fiduccia and R. M. Mattheyses. *A linear-time heuristic for improving network partitions.* In DAC '82: IEEE/ACM Design Automation Conference, pages 175–181, 1982.

[Galil 1977] Z. Galil and N. Megiddo. *Cyclic Ordering is NP-Complete.* Theoretical Computer Science, vol. 5, pages 179–182, 1977.

[Garey 1979] M. R. Garey and D. S. Johnson. Computers and intractability: A guide to theory of NP-completeness. W. H. Freeman & Co., San Francisco, CA, 1979.

[Goldberg 1988] A. V. Goldberg and R. E. Tarjan. *A New Approach to the Maximum- Flow Problem.* Journal of ACM, vol. 35, no. 4, pages 921–940, 1988.

[Golub 1983] G.H. Golub and C. Loan. Matrix computations. JHU, 1983.

[Gopalakrishnan 2006] P. Gopalakrishnan, X. Li and L. T. Pileggi. *Architecture-aware FPGA placement using metric embedding.* In DAC '06: Design Automation Conference, pages 460–465, 2006.

[Gotsman 1996] C. Gotsman and M. Lindenbaum. *On the Metric Properties of Discrete Space-Filling Curves.* IEEE Transaction on Image Processing, vol. 5, no. 5, pages 794–797, 1996.

[GSRC] GSRC. http://www.cse.ucsc.edu/research/surf/GSRC/progress.html.

[Hardy 1979] G. H. Hardy and E. M. Wright. An introduction to the theory of numbers. Oxford, England, 5 édition, 1979.

[Hilbert 1891] D. Hilbert. *Über stetige Abbildung einer Linie auf ein Flächenstück.* Mathematische Annalen, vol. 38, pages 459–460, 1891.

[hMetis] hMetis. http://www-users.cs.umn.edu/ karypis/metis/hmetis.

[IBM] IBM. *IBM Corporation.* http://www.ibm.com/.

[IEEE 1987] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1987.

[Kahng 2000] A. B. Kahng. *Classical Floorplanning Harmful?* In ISPD '00: International Symposium on Physical Design, pages 207–213, 2000.

[Karypis 1999a] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar. *Multilevel Hypergraph Partitioning: Applications in VLSI Domain.* IEEE Transaction on Very Large Scale Integration (VLSI) Systems, vol. 7, no. 1, pages 69–79, 1999.

[Karypis 1999b] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar. *Multilevel Hypergraph Partitioning: Applications in VLSI domain.* IEEE Transaction on VLSI, vol. 7, no. 1, pages 69–79, 1999.

[Kernighan 1970a] B. W. Kernighan and S. Lin. *An efficient heuristic procedure for partitioning graphs.* Bell System Technical Journal, vol. 49, pages 291–307, 1970.

[Kernighan 1970b] W. Kernighan and S. Lin. *An Efficient Heuristic Procedure for Partitioning Graphs.* Bell System Technical Journal, vol. 49, pages 291–307, 1970.

[Khalid 1995] M. Khalid and J. Rose. *The Effect of Fixed I/O Pin Positioning on The Routability and Speed of FPGAs.* In FPD '95: Canadian Workshop of Field-Programmable Devices, pages 94–102, 1995.

[Kong 2002] T. Kong. *A novel net weighting algorithm for timing-driven placement.* In ICCAD '02: IEEE/ACM International Conference on Computer-aided design, pages 172–176, New York, NY, USA, 2002. ACM.

[Kuon 2007] I. Kuon, R. Tessier and J. Rose. *FPGA Architecture: Survey and Challenges.* Foundation and Trends in Electronic Design Automation, vol. 2, no. 2, pages 135–253, 2007.

[Lattice] Lattice. *Lattice Semiconductor Corporation.* http://www. lattice-semi.com/.

[LEDA] LEDA. http://www.algorithmic-solutions.com/.

[Liberatore 2002] V. Liberatore. *Circular Arrangements.* LNCS: Automata, Languages and Programming, vol. 2380/2002, pages 782–783, 2002.

[Maidee 2003] P. Maidee, C. Ababei and K. Bazargan. *Fast Timing-driven Partitioning-based Placement for Island style FPGAs.* In DAC '03: ACM /IEEE Design Automation Conference, pages 598–603, 2003.

[Maidee 2005] P. Maidee, C. Ababei and K. Bazargan. *Timing-driven partitioning-based placement for island style FPGAs.* IEEE Transaction on CAD of Integrated Circuits and Systems, vol. 24, no. 3, pages 395–406, 2005.

[Mak 2004] W. K. Mak. *I/O Placement for FPGAs With Multiple I/O Standards.* IEEE Transaction on CAD of Integrated Circuits and Systems, vol. 23, no. 2, pages 315–320, 2004.

[Mak 2005] W. K. Mak. *Modern FPGA constrained placement.* In ASP-DAC '05: International Conference on Asia South Pacific Design Automation, pages 779–784, New York, NY, USA, 2005. ACM.

[Manimegalai 2007] R. Manimegalai. Efficient logic synthesis and placement techniques for modern FPGA architectures. Ph. D Thesis, Dept. of Computer Science, Indian Institute of Technology, Madras, June 2007.

[Marquardt 2000] A. Marquardt, V. Betz and J. Rose. *Timing-driven placement for FPGAs.* In FPGA '00: ACM/SIGDA eighth International Symposium on Field Programmable Gate Arrays, pages 203–213, New York, NY, USA, 2000. ACM.

[Matousek 2002] J. Matousek. Lectures on discrete geometry. Springer, May 2002.

[Mokbel 2002] Mohamed F. Mokbel, Walid G. Aref and Ibrahim Kamel. *Performance of multi-dimensional space-filling curves.* In Proceedings of the 10th ACM international Symposium on Advances in Geographic Information Systems, pages 149–154, New York, NY, USA, 2002. ACM.

[Mulpuri 2001] C. Mulpuri and S. Hauck. *Runtime and quality tradeoffs in FPGA placement and routing.* In FPGA '01: ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pages 29–36, 2001.

[Nag 1998] S. K. Nag and R. A. Rutenbar. *Performance-Driven simultaneous placement and Routing for FPGAs.* IEEE Transaction on CAD of Integrated Circuits and Systems, vol. 17, no. 6, pages 499–518, 1998.

[Nandy 1997] S. C. Nandy, G. N. Nandakumar and B. B. Bhattacharya. *Efficient Algorithms for Single and Two-layer Linear Placement of Parallel Graphs.* Computers and Mathematics with Application, vol. 34, no. 12, pages 121–135, 1997.

[Ngai 1995] T. Ngai, J. Rose and S. Wilton. *An SRAM-programmable field-configurable memory.* In IEEE Custom Integrated Circuits Conference, pages 499–502, Santa Clara, CA, May 1995.

[Niven 1991] I. Niven, H. S. Zuckerman and H. L. Montgomery. An introduction to the theory of numbers. Wiley, Singapore, 5 édition, 1991.

[Otten 1982] R. H. J. M. Otten. *Automatic Floorplan Design.* In DAC '82: Design Automation Conference, pages 261–267, 1982.

[Papadimitriou 2006] C. H. Papadimitriou and K. Steiglitz. Combinatorial optimization: Algorithms and complexity. Prentice Hall of India, New Delhi, 2006.

[PARQUET] PARQUET. http://vlsicad.eecs.umich.edu/BK/parquet/.

[Peano 1890] G. Peano. *Sur une courbe qui remplit toute une aire plaine.* Mathematische Annalen, vol. 36, pages 157–160, 1890.

[QuickLogic] QuickLogic. *QuickLogic Corporation.* http://www.quicklogic.com/.

[Rao 1998] S. Rao and A. W. Richa. *New Approximation Techniques for Some Ordering Problems.* In ACM-SIAM Symposium on Discrete Algorithms, pages 211–218, 1998.

[Roy 2006] J. A. Roy, S. N. Adya, David A. Papa and I. L. Markov. *Min-Cut Floorplacement.* IEEE Transaction on CAD of Integrated Circuits and Systems, vol. 25, no. 7, pages 1313–1326, 2006.

[Sagan 1994] H. Sagan. Space-filling curves. Springer Verlag, ISBN 0-387-94265-3, 1994.

[Sankar 1999] Y. Sankar and J. Rose. *Trading quality for compile time: ultra-fast placement for FPGAs.* In FPGA '99: ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pages 157–166, New York, NY, USA, 1999. ACM.

[Sarrafzadeh 1996] M. Sarrafzadeh and C.K. Wong. An introduction to VLSI physical design. Mcgraw Hill, 1996.

[Sarrafzadeh 2001] M. Sarrafzadeh, E. Bozorgzadeh, R. Kastner and A. Srivastava. *Design and analysis of physical design algorithms.* In ISPD '01: ACM/SIGDA International Symposium on Physical Design, pages 82–89, 2001.

[Schrijver 1998] A. Schrijver. Theory of linear and integer programming. John Wiley & Sons, June 1998.

[Sechen 1988] C. Sechen. VLSI placement and global routing using simulated annealing. Kluwer Academic Publishers, Boston, USA, 1988.

[Shahookar 1991] K. Shahookar and P. Mazumdar. *VLSI Cell Placement Techniques.* ACM Computing Surveys, vol. 23, no. 2, pages 143–220, June 1991.

[Sherwani 1993] N. A. Sherwani. Algorithms for VLSI physical design automation. Kluwer Academic Publishers, Boston/Dordrecht/London, 1993.

[Shiloach 1979] Y. Shiloach. *Minimum linear arrangement algorithm for undirected trees.* SIAM Journal on Computing, vol. 8, no. 1, pages 15–32, 1979.

[SiliconBlue] SiliconBlue. *SiliconBlue Technologies Corporation.* http://www. siliconbluetech.com/.

[Singhal 2006] L. Singhal and E. Bozorgzadeh. *Multi-layer Floorplanning on a Sequence of Reconfigurable Designs.* In FPL '06: IEEE International Conference on Field Programmable Logic and Applications, pages 1–8, 2006.

[Singhal 2007a] L. Singhal and E. Bozorgzadeh. *Heterogeneous Floorplanner for FPGA.* In FCCM '07: 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pages 311–312, Washington, DC, USA, 2007. IEEE Computer Society.

[Singhal 2007b] L. Singhal and E. Bozorgzadeh. *Novel multi-layer floorplanning for Heterogeneous FPGAs.* In FPL '07: IEEE International Conference on Field Programmable Logic and Applications, pages 613–616, 2007.

[Stockmeyer 1983] L. J. Stockmeyer. *Optimal Orientations of Cells in Slicing Floorplan Designs.* Information and Control, vol. 57, no. 2/3, pages 91–101, 1983.

[Taghavi 2004] T. Taghavi, S. Ghiasi, A. Ranjan, S. Raje and M. Sarrafzadeh. *Innovate or perish: FPGA physical design.* In ISPD '04: International Symposium on Physical design, pages 148–155, New York, NY, USA, 2004. ACM.

[Tessier 2002] R. Tessier. *Fast Placement Approaches for FPGAs.* ACM Transaction on Design Automation of Electronic Systems, vol. 7, no. 2, pages 284–305, 2002.

[Vazirani 2001] V. V. Vazirani. Approximation algorithms. Springer, Heidelberg, Germany, 2001.

[Vempala 1998] S. Vempala. *Random projection: a new approach to VLSI layout.* In IEEE Symposium on Foundations of Computer Science, 1998.

[Verilog International 1993] Verilog International. *Verilog Hardware Description Reference*, March 1993.

[Vicente 2004] J. D. Vicente, J. Lanchares and R. Hermida. *Annealing Placement by Thermodynamic Combinatorial Optimization.* ACM Transaction on Design Automation of Electronic Systems, vol. 9, no. 3, pages 310–332, July 2004.

[Vorwerk 2009] K. Vorwerk, A. Kennings and J. W. Greene. *Improving Simulated Annealing-Based FPGA Placement With Directed Moves.* IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, vol. 28, no. 2, pages 179–192, February 2009.

[Vygen 2007] J. Vygen. *New theoretical results on quadratic placement.* Integration, the VLSI Journal, vol. 40, no. 3, pages 305–314, 2007.

[Wang 2003] M. Wang, A. Ranjan and S. Raje. *Multi-Million Gate FPGA Physical Design Challenges.* In ICCAD '03: International Conference on Computer Aided Design, pages 891–898, 2003.

[Wong 1988] D. F. Wong, H. W. Leong and C. L. Liu. Simulated annealing for VLSI design. Kluwer Academic Publishers, Boston, USA, 1988.

[Wood 1987] D. Wood. Theory of computation. Harper & Row, 1987.

[Xilinx] Xilinx. http://www.xilinx.com.

[Xilinx 2005] Xilinx. *Virtex-II platform FPGAs: Complete data sheet. DS031(v3.4).* http://direct.xilinx.com/bvdocs/publications/ds031.pdf, 2005.

[Xu 2005] Y. Xu and M.A.S. Khalid. *QPF: Efficient Quadratic Placement for FPGAs.* In FPL '05: IEEE International Conference on Field Programmable Logic and Applications, pages 555–558, 2005.

[Yuan 2005] J. Yuan, S.Q. Dong, X.L. Hong and Y.L. Wu. *LFF Algorithm for Heterogeneous FPGA Floorplanning.* In ASP-DAC '05: International Conference on Asia South Pacific Design Automation, pages 1123–1126, 2005.

Publications from the Thesis

Refereed Journals

(J1) **Pritha Banerjee**, Susmita Sur-Kolay and Arijit Bishnu, “Fast Unified Floorplan Topology Generation and Sizing on Heterogeneous FPGAs”, in *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5., May 2009, pp. 651-661.

(J2) **Pritha Banerjee**, Susmita Sur-Kolay, Arijit Bishnu, Sandip Das, Subhas C. Nandy and Subhasis Bhattacharjee, “FPGA Placement using Space Filling Curves: Theory Meets Practice”, in Special issue on Configuring Algorithms, Processes and Architecture (CAPA), *ACM Trans. on Embedded Computing Systems*, vol. 9, no. 2, October 2009, pp. 12:1-12:23.

(J3) **Pritha Banerjee**, Debasri Saha and Susmita Sur-Kolay, "Cone-based placement for field programmable gate arrays", to appear in *IET Computers and Digital Techniques.*

(J4) **Pritha Banerjee**, Megha Sangtani and Susmita Sur-Kolay, "Floorplanning for Partially Reconfigurable FPGAs", to appear in *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems.*

Refereed Conference Proceedings

(C1) **Pritha Banerjee** and Susmita Sur Kolay, “An Accelerator for FPGA Placement”, in *Progress in VLSI Design and Test*, 6th VLSI Design and Test Workshops, pp. 340-347, August 2002, Bangalore, India.

(C2) **Pritha Banerjee**, Subhais Bhattacharjee, Susmita Sur-Kolay, Sandip Das and Subhas C. Nandy, “Fast FPGA Placement using Space-filling Curve”, in *Proc. of the 15th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE CS Press, pp. 415-420, August 2005, Tampere, Finland.

(C3) **Pritha Banerjee**, Susmita Sur-Kolay and Arijit Bishnu, “Floorplanning in Modern FPGAs” in *Proc. of the 20th International Conference on VLSI Design*, IEEE CS Press, pp. 893-898, January 2007, Bangalore, India.

(C4) **Pritha Banerjee** and Susmita Sur-Kolay, “Faster Placer for Island-style FPGAs”, in *Proc. of International Conference on Computing: Theory and Applications*, IEEE CS Press, pp. 117-121, March 2007, Kolkata, India.

(C5) Debasri Saha, **Pritha Banerjee** and Susmita Sur Kolay, “Fast I/O Pad Placement in FPGAs”, in *Progress in VLSI Design and Test*, 11th VLSI Design And Test Symposium, pp. 153-161, August 2007, Kolkata, India.

(C6) **Pritha Banerjee**, Megha Sangtani and Susmita Sur-Kolay, “Floorplanning for Partial Reconfiguration in FPGAs”, in *Proc. of the 22nd International Conference on VLSI Design*, IEEE CS Press, January 2009, New Delhi, India.