Timing Driven Analytical Placement for FPGA

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Nimish Agashiwala

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL ENGINEERING

Prof. Kiarash Bazargan

September, 2015

© Nimish Agashiwala 2015
ALL RIGHTS RESERVED

Acknowledgements

First and foremost, I express my sincere gratitude to my advisor and mentor, Prof. Kia Bazargan, who has supported me throughout my thesis with his patience, guidance and constant motivation to innovate. It is due to his teachings and supervision that I have completed this research work with ease.

Besides my advisor, I would like to thank the faculty members on my thesis committee, Prof. Marc Riedel and Prof. Antonia Zhai, for their valuable time in reviewing my thesis and providing useful feedback and suggestions.

I take this opportunity to thank Jason Luu and the entire VTR software and support team, University of Toronto, for their prompt responses to queries I had regarding the VTR tool framework.

I would like to make a special mention about two people, my classmate, Darshak Gandhi and my lab partner, Satya Prakash Upadhyay, who initiated this idea and developed the resource base for ASIC framework during their VLSI CAD 2 course and for helping me ramp up with this topic in the summer of 2014. I would also like to thank my friends Vaishnavi, Harini, Utkarsh and Devavrata who were a constant source of motivation and fun during my thesis.

I would like to thank the ECE IT support staff at University of Minnesota, for their assistance with IT related activities in our research lab.

And last but not the least, I would like to express my sincere thanks to my father Mukesh Agashiwala, my mother Hemangini Agashiwala, my brother Amish and my family for giving me the opportunity to pursue Masters miles away from home. They always helped me keep my moral high and encouraged me to work without thinking about the end results.

Dedication

I dedicate this thesis to my parents who have been a constant source of motivation and have always guided and supported me throughout my life.

Abstract

Conventional Simulated Annealing (SA) based placement methods for FPGAs give best results in terms of wirelength and critical path delay. The runtime for these SA based methods is directly proportional to the total number of cells to be placed. In case of modern multi-million gate FPGAs, SA based methods for placement dominate the total runtime in the FPGA CAD flow. In this thesis, we propose a new fast and efficient timing driven analytical placement engine targeted at global placement for FPGAs followed by low temperature SA for detailed placement. Our global placement engine uses quadratic programming approach to minimize the wirelength and dynamic net weights based on timing criticality between the blocks to minimize the critical path delay. The placement engine proceeds by iteratively partitioning the placement area and making the Configurable Logic Blocks (CLBs) move near each partition's Center of Gravity (CG). After each iteration, to calculate the timing criticality between each CLB, they are snapped to physical grid locations. The placement engine uses this timing feedback to update the net weights and calculate new coordinates for the CLBs in the next iteration. We employ a spiral legalization method in the end to obtain a legalized placement which then undergoes low temperature Simulated Annealing in VPR to give comparable or better critical path delay and 8.7% bad overall wirelength after placement. Experimental results of 20 largest MCNC benchmark circuits show that our global placement engine outperforms the state-of-the-art academic placer VPR 7.0 in terms of runtime by 30% on an average, making it scalable and provides an overall similar QoR in terms of critical path delay.

Contents

Acknowledgements i

Dedication ii

Abstract iii

List of Tables vi

List of Figures vii

1 Introduction 1

2 Background and Motivation 5

2.1 FPGA Architecture 5

2.2 VTR Flow and VPR 9

2.2.1 VTR CAD Flow 10

2.3 Placement Problem Definition 12

2.3.1 Introduction to Quadratic Placement Approach 12

2.3.2 Timing Driven Placers 14

2.3.3 GORDIAN-like Approach 16

2.4 Previous Work 17

2.5 Motivation and Research Goal 19

2.6 Summary 20

3 Quadratic Placement CAD Flow 21

3.1 Timing Driven CAD Flow Overview 21

3.2 MATLAB CAD Flow . . . 23
3.3 Placement Engine Functions and Parameters . . . 26
3.3.1 The Connectivity Matrix . . . 26
3.3.2 quadprog Function . . . 28
3.3.3 try_legalize Function . . . 28
3.3.4 Number of Iterations . . . 33
3.3.5 Timing Analysis Methodology in VPR . . . 36
3.3.6 Net Weighting Scheme . . . 38
3.3.7 Low Temperature Simulated Annealing . . . 40

4 Experiment and Result Analysis 45
4.1 Discussion on Experimental Setup . . . 45
4.1.1 Architecture File . . . 45
4.1.2 Benchmark Circuits . . . 47
4.1.3 Timing Criticality File . . . 49
4.1.4 VPR Reference Placement IO Locations . . . 50
4.2 Results and Analysis . . . 50
4.2.1 Post Placement Results . . . 51
4.2.2 Post Route Results . . . 58

5 Conclusion and Future Ideas 68
5.1 Conclusion . . . 68
5.2 Future Ideas . . . 69

References 71

Appendix A. Acronyms 76
A.1 Acronyms . . . 76

Appendix B. Data Structures 77

List of Tables

3.1 Temperature Update Schedule [2] 43
4.1 Major Architecture Files in VTR 7.0 Release [6] 46
4.2 Statistics of Benchmark Circuits for k6_N10_40nm Architecture 48
4.3 Comparison: Sum of timing criticality across all nets after placement for VPR, MATLAB and after low temperature SA 61
4.4 Comparison: Critical Path Delay (CPD) after placement for VPR, MATLAB and after low temperature SA 62
4.5 Comparison: Estimated Wirelength after placement for VPR, MATLAB and after low temperature SA 63
4.6 Comparison: Placement runtime for VPR, MATLAB and low temperature SA 64
4.7 MATLAB Global placement runtime break-up for each stage 65
4.8 Comparison: Post Route Critical Path Delay (R_CPD) and Wirelength (R_WL) for Golden VPR and AP Placement 66
4.9 Comparison: Post Route Channel Factor for Golden VPR and AP Placement 67
A.1 Acronyms 76

List of Figures

2.1 Island Style Homogeneous FPGA Architecture 6
2.2 Island Style Heterogeneous FPGA Architecture [11] 7
2.3 Switch and Connection Box Configuration [3] 8
2.4 Basic Logic Element (BLE)[3] 9
2.5 Configurable Logic Element (CLB) [3] 9
2.6 VTR FPGA CAD Flow [5] 11
2.7 Dummy Circuit to illustrate Quadratic Placement Formulation [37] . . . 14
2.8 Gordian Method of Spreading by Center of Mass Constraints[47] 16
3.1 Timing Driven CAD Flow Overview 22
3.2 MATLAB CAD Flow 24
3.3 4x4 FPGA g rid with grey blocks representing physical grid locations and yellow blocks representing CLB placement by final quadprog iteration . 29
3.4 Spiral Legalization Pictorial Representation 32
3.5 8 CLBs - Iteration 0 34
3.6 8 CLBs - Iteration 1 34
3.7 15 CLBs - Iteration 0 35
3.8 15 CLBs - Iteration 1 35
3.9 16 CLBs - Iteration 0 36
3.10 16 CLBs - Iteration 1 36
3.11 16 CLBs - Iteration 2 36
4.1 Classical Soft Logic Block [6] 47
4.2 Graph showing comparison of sum of timing criticality of all nets per benchmark 53
4.3 Graph showing comparison for Critical Path Delay (CPD) 54

4.4 Graph showing comparison for estimated wirelength after placement (HPWL) 55
4.5 Graph showing relationship between VPR and MATLAB Runtime . . . 56
4.6 Pie Chart showing the break-up of MATLAB and Cool SA Placement Runtimes 57
4.7 Graph showing comparison post route critical path delay (R_CPD)) . . 59
4.8 Graph showing comparison post route wirelength (R_WL)) 60

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are today's most essential part of embedded computing ecosystem. Since 1985, when the first commercial FPGA was invented, the FPGA industry has seen almost an exponential growth in its use, primarily because of two reasons - reduced turn around times and lower manufacturing costs for prototyping circuits. FPGAs are termed as the "soft microporcessors" of modern day. Today, the FPGA market is expanding its territory in the High Performance Computing domain too, where reduced energy costs and higher performance in terms of better runtime and speed is the key.

With the advancement of technology, today's FPGAs can house millions of logic cells in Configurable Logic Blocks (CLBs), memory blocks, multipliers, adders, Digital Signal Processing Blocks (DSPs) and other circuit components. They have prefabricated horizontal and vertical interconnects connected using switch boxes that join wire segments of variable lengths. Similar to ASIC design flows, FPGA CAD flow, too, follows the same kind design cycle. First based on the specifications listed, a Hardware Descriptive Language (HDL) like Verilog or VHDL is used to develop the design. Synthesis converts this HDL into a flattened netlist of logic gates. Next step in FPGA design cycle performs technology independent logic optimization of each circuit followed by technology mapping of this circuit into Look-Up Tables (LUTs) and flip flops. Further, these LUTs and flip flops undergo a packing stage, wherein, they are packed into more coarse-grained logic blocks called Configurable Logic Blocks (CLBs). Packing stage aims to pack the LUTs and flip flops that are connected together, so that cost of routing is

reduced during the routing stage. Placement step places the packed netlist of CLBs onto the FPGA chip such that it consumes less area, less routing resources and meets timing. Once placement is done, the CLBs are connected using the prefabricated wire segments and switch boxes in the routing stage.

Achieving better overall run-times coupled with reduced power consumption calls for the need of better CAD algorithms for modern day FPGAs. One of the most challenging and time consuming steps in FPGA CAD flow is the placement step. The main difference between ASIC placement and FPGA placement is that, in the latter, every CLB has a specific physical location on the chip where it can be placed. It is the need of the hour to develop an algorithm that not only consumes less runtime, but also provides better quality of results and is scalable to larger circuits too. Placement can be broadly classified into Global Placement and Detailed Placement. Global placement aims at optimizing either wirelength or timing or a combination of both wirelength and timing, whereas, detailed placement concentrates on legalization and refining the final wirelength and timing of the placed circuit. Global placement strategy includes three major classes of algorithms being employed in ASIC/FPGA design styles. The first and the most efficient is simulated annealing based placement in which an attempt is made to reduce the wirelength or timing cost function by swapping of the logical blocks. If the swap results in reduction in the cost, then it is accepted; otherwise it is either accepted or rejected based on the number of factors. This algorithm models the physical process of heating a material and then slowly lowering the temperature to decrease the defects, thus minimizing the system energy [44]. Although, this method provides high quality results in terms of wirelength and critical path delay, it is not easy to scale and use it for large FPGA circuits because of tremendous increase in placement runtime. Other class of global placement algorithms is the partitioning based placement where the given circuit is repeatedly divided into densely connected sub-partitions. This partitioning is based on techniques such as Kernighan–Lin (KL) algorithm [7] or Fiduccia-Mattheyses (FM) algorithm [8]. The motive behind partitioning algorithms is to reduce the interconnection between the partitions i.e. put all the highly connected cells together in one partition so that overall wirelength can be minimized. This class of algorithm is faster than the simulated annealing process, however, it results in poor quality of results. The third category of global placement is the analytical placement algorithm which utilize

a quadratic wirelength objective function. The quadratic wirelength is only an indirect measure of the linear wirelength, still its main advantage is that it can be minimized quite efficiently. Hence, analytical placement algorithms are comparatively fast and efficient in handling large circuit placements. The main disadvantage with analytical placement methods is that the placement obtained contains large number of overlaps among the logic blocks. To legalize the placement, a detailed placement strategy to partition and spread the logic blocks is usually employed.

Our research focuses on developing a fast timing driven global analytical placement engine for homogeneous FPGA architectures. Our placement engine gives a legalized output, which then undergoes the detailed placement step using low temperature simulated annealing to optimize the overall critical path delay and routed wirelength. We implemented our idea using the Verilog-to-Routing (VTR) 7.0 [4] [6] framework as our base platform. It is a widely used state-of-the-art open source academic CAD tool for FPGA. We propose to replace the placement step of Versatile Place and Route (VPR) tool [1] within the VTR framework with our placement engine designed in MATLAB. We provide the architecture file of FPGA and circuit netlist to VPR and generate a connectivity matrix. This matrix is weighted based on the timing criticality information generated by performing timing analysis after each intermediate iteration of our analytical placer. Legalization is performed in the final iteration and this legalized output is optimized further using low temperature simulated annealing in VPR to get higher quality of results in terms of critical path delay and wirelegth. We analyze our placer with 20 MCNC benchmark circuits [9] designed for FPGAs and other homogeneous VTR benchmarks available in VTR 7.0 on k6_N10_40nm architecture.

The rest of the thesis chapters are organized as follows:

- Chapter 2 briefly presents details about the FPGA island style architecture, VTR flow and VPR Tool in general, various strategies in timing driven placement and our definition of the proposed quadratic placement method. We will also revisit the past works in the area of FPGA placement and highlight our motivation and research goal.

- In Chapter 3, the timing driven quadratic placement CAD flow is outlined.

- Chapter 4 discusses about our experimental setup. We will also look at the placement metrics obtained, analyse them and compare them with the results of VPR 7.0 placement flow.

- Chapter 5 concludes the thesis by highlighting the key achievements and results of this work and enlists possible direction of future research in this area.

Chapter 2

Background and Motivation

This chapter presents the background of the FPGAs, the FPGA Architecture, placement techniques for FPGA and the previous work done in this field. We start by discussing the FPGA Architecture in section 2.1, followed by a complete overview of VTR and VPR 7.0 Flows in section 2.2. In section 2.3, we describe our placement problem definition and in section 2.4, we highlight the recent works done in this field. Finally, we end the chapter by describing our motivation for going ahead with the research in section 2.5.

2.1 FPGA Architecture

Field Programmable Gate Arrays (FPGAs) are programmable semiconductor devices that consist of Configurable Logic Blocks (CLBs) connected using prefabricated interconnects. These devices can be electrically programmed in the field to any kind of digital circuit. FPGAs allow the designers to modify/update their design very late in the design cycle i.e. even after the end product has been manufactured and deployed in the field. Although many different flavours of architecture are available for FPGAs, all of them contain these fundamental components, viz.:

- Configurable Logic Blocks (CLBs)
- I/O Blocks
- Switch Boxes

- Connection Blocks
- Pre-fabricated Network of Interconnects

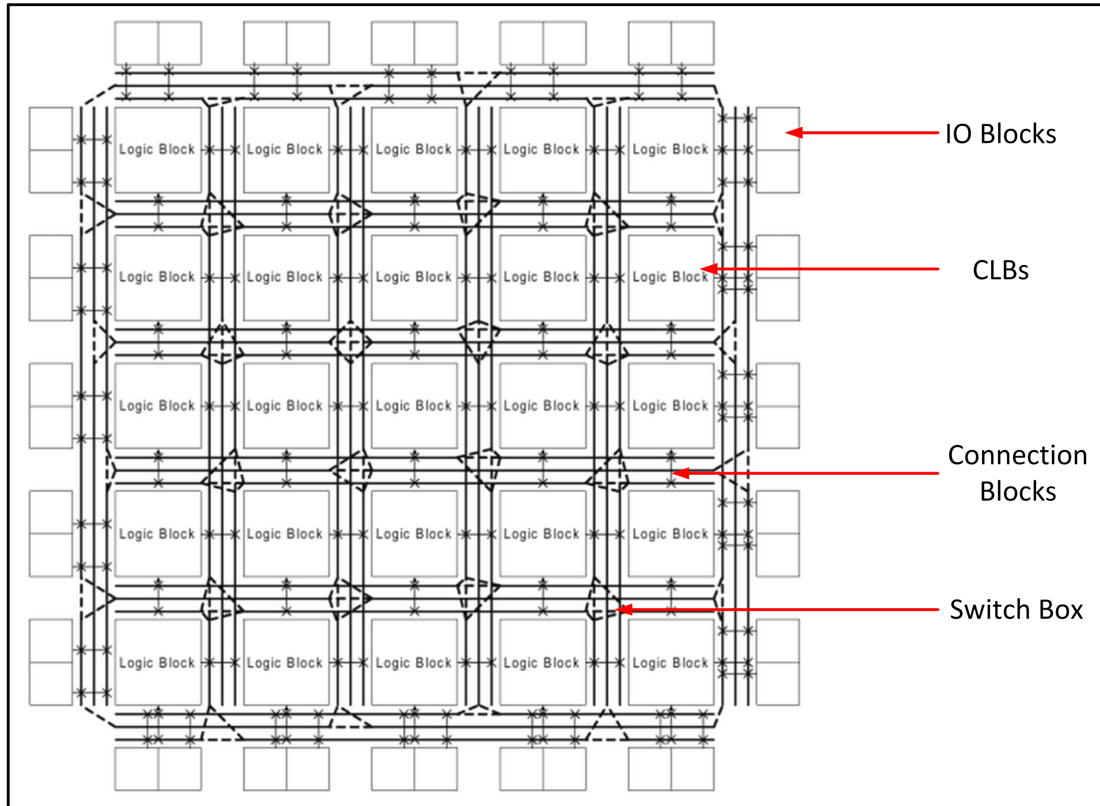

Figure 2.1: Island Style Homogeneous FPGA Architecture

The FPGA architecture having only the fundamental blocks is known as a Homogeneous FPGA Architecture. Apart from these fundamental components, if the FPGA contains one or more hard blocks like multiplier blocks, memory units, DSP blocks etc, then such an architecture is termed as Heterogenous FPGA Architecture. Figure 2.1 and 2.2 show the basic homogeneous and heterogeneous FPGA Architecture configurations respectively. As illustrated in figure 2.1, the homogeneous FPGA fabric has the CLBs and routing resources (switch boxes, connection boxes and interconnects) in the center surrounded by the IO blocks on the chip edges. This mimics the island style of FPGA architecture. In case of heterogeneous architecture, along with the fundamental

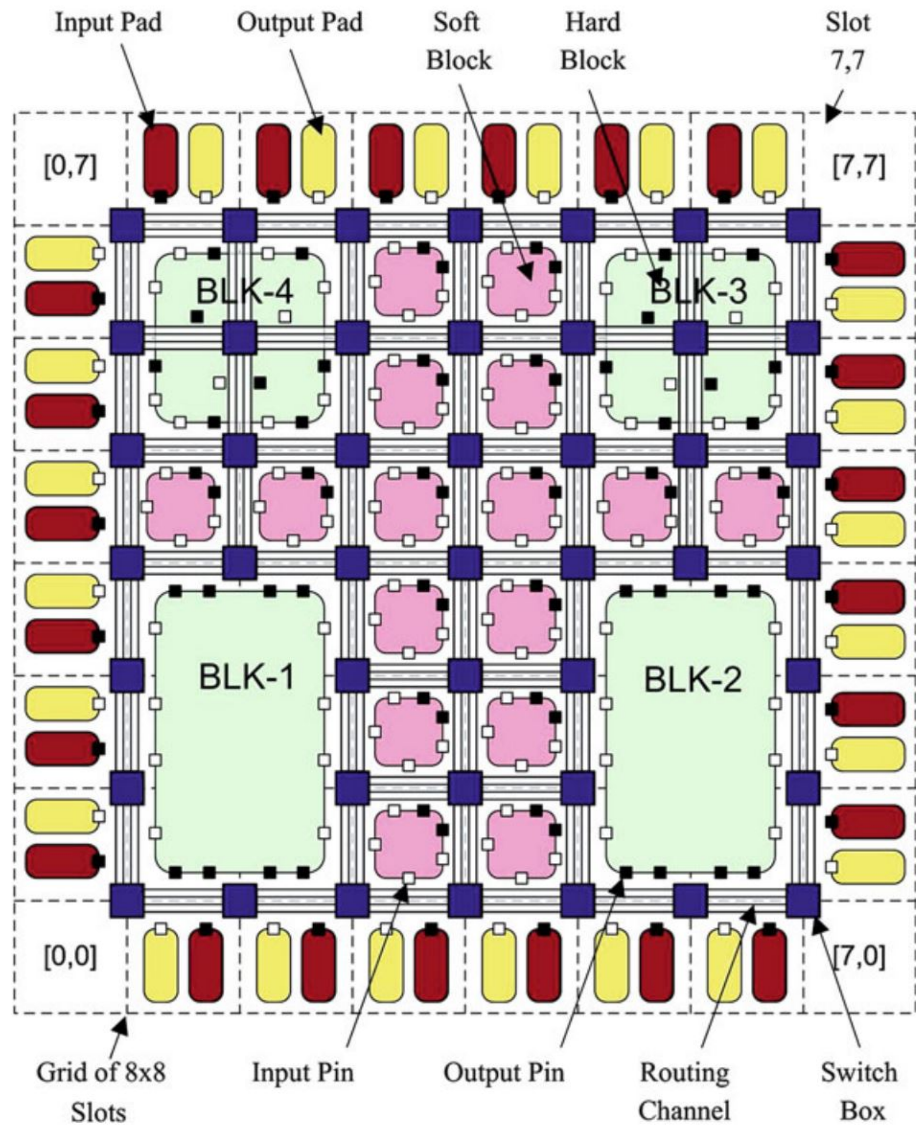

Figure 2.2: Island Style Heterogeneous FPGA Architecture [11]

components, there are islands of multipler blocks, hard marcos, DSPs etc. in the center of the FPGA as shown in figure 2.2. In our research work, we concentrate on the homogeneous island style of FPGA architectures.

A circuit can be realized in FPGA by programming all the logic blocks to implement the logic of the circuit. The I/O blocks serve as the input and output pads of the circuit. The switch boxes connect the wires in one channel to the wire in another channel, whereas, the connection boxes are programmed to connect the logic blocks with their routing channels. Figure 2.3, shows the configuration of the switch boxes and connection boxes working to make two CLBs connect to each other.

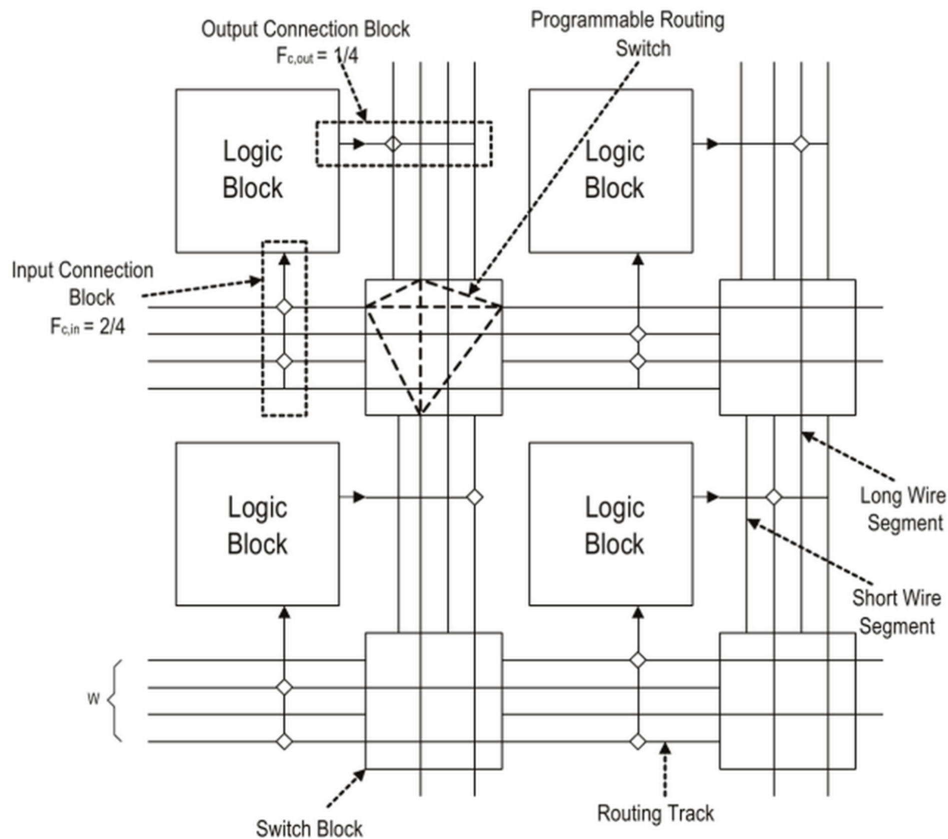

Figure 2.3: Switch and Connection Box Configuration [3]

Configurable Logic Blocks are made up of Basic Logic Elements (BLEs). Depending on the architecture, there can be one or more BLEs packed in one CLB. Each BLE

contains the primitive blocks: a K-input Look-Up Table (LUT) and a flip-flop. The output of the BLE can be the output of LUT or the LUT output can be registered and used as the BLE output. Figures 2.4 and 2.5, show the structures of BLE and CLBs respectively. In this thesis, we have the following architecture:

- CLBs containing ten BLEs packed within them and each BLE constructed using a 6-Input LUT and a flip-flop (k6_N10_40nm)

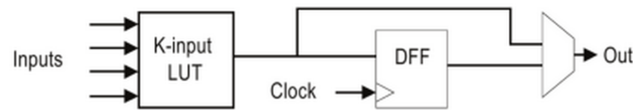

Figure 2.4: Basic Logic Element (BLE)[3]

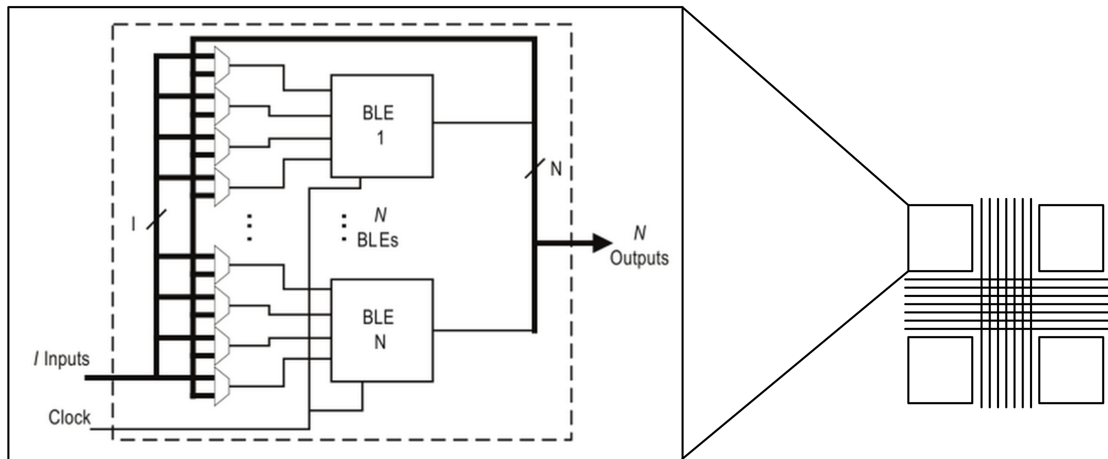

Figure 2.5: Configurable Logic Element (CLB) [3]

2.2 VTR Flow and VPR

Verilog-To-Routing (VTR) [4] is a state-of-the-art academic open source tool for FPGA CAD, designed by the team at University of Toronto. It takes the HDL code, synthesizes it and makes it of the form suitable for placement and routing in FPGA. The entire

VTR flow will be explained in the subsection 2.2.1. Versatile Place and Route (VPR) [2] is a part of VTR framework that maps a technology mapped netlist i.e. a circuit expressed in Look-Up Tables (LUTs), flip-flops, memories etc. to a hypothetical FPGA specified by the user. Basically VPR deals with the packing, placement and routing aspect of the FPGA CAD flow. We are using VTR 7.0 [6] release of the tool in our research work.

2.2.1 VTR CAD Flow

VTR framework basically includes three CAD tools - ODIN II [12] for front end synthesis, ABC [13] for logic optimization and technology mapping and VPR [14], [2], [3], [1] which helps in packing, placement and routing. Figure 2.6 illustrates the VTR CAD flow. First, ODIN II converts the Verilog HDL design into a flattened netlist consisting of logic gates and blackboxes that represent heterogeneous blocks. Next, ABC synthesis package performs technology-independent logic optimization of each circuit and then each circuit is technology-mapped into LUTs and flip flops. The output of ABC is a *.blif* format netlist of LUTs, flip-flops and blackboxes. VPR [2], then packs this netlist into more coarse-grained logic blocks, places the circuit and routes it. The output of VPR includes several files which contain circuit's packing, placement and routing information. It also dumps several echo files, which contain important information regarding VPR's inbuilt data structures, timing slack and criticality information, the inter-block connectivity information etc.

Being an academic open source tool has its own advantages. Each of these stages can be bypassed and replaced by user's own flow for that particular stage. In this thesis, we have replaced the VPR's simulated annealing based placement [1] with our timing driven analytical placement methodology where the placement proceeds by iteratively partitioning the FPGA according to the GORDIAN-like partitioning method [29]. VPR requires two input files viz. the circuit netlist *.blif* file and the architecture file *.xml* file to begin its flow. VPR will perform packing, placement and routing on the circuit to the architecture. VPR provides with various options that can be provided as command line arguments while invoking it to do its work. For eg. if we have the packed netlist and only wish to perform placement and routing, then we need to tell VPR explicitly so that it only undergoes the place and route flows. VPR User Manual [5], provides a detailed

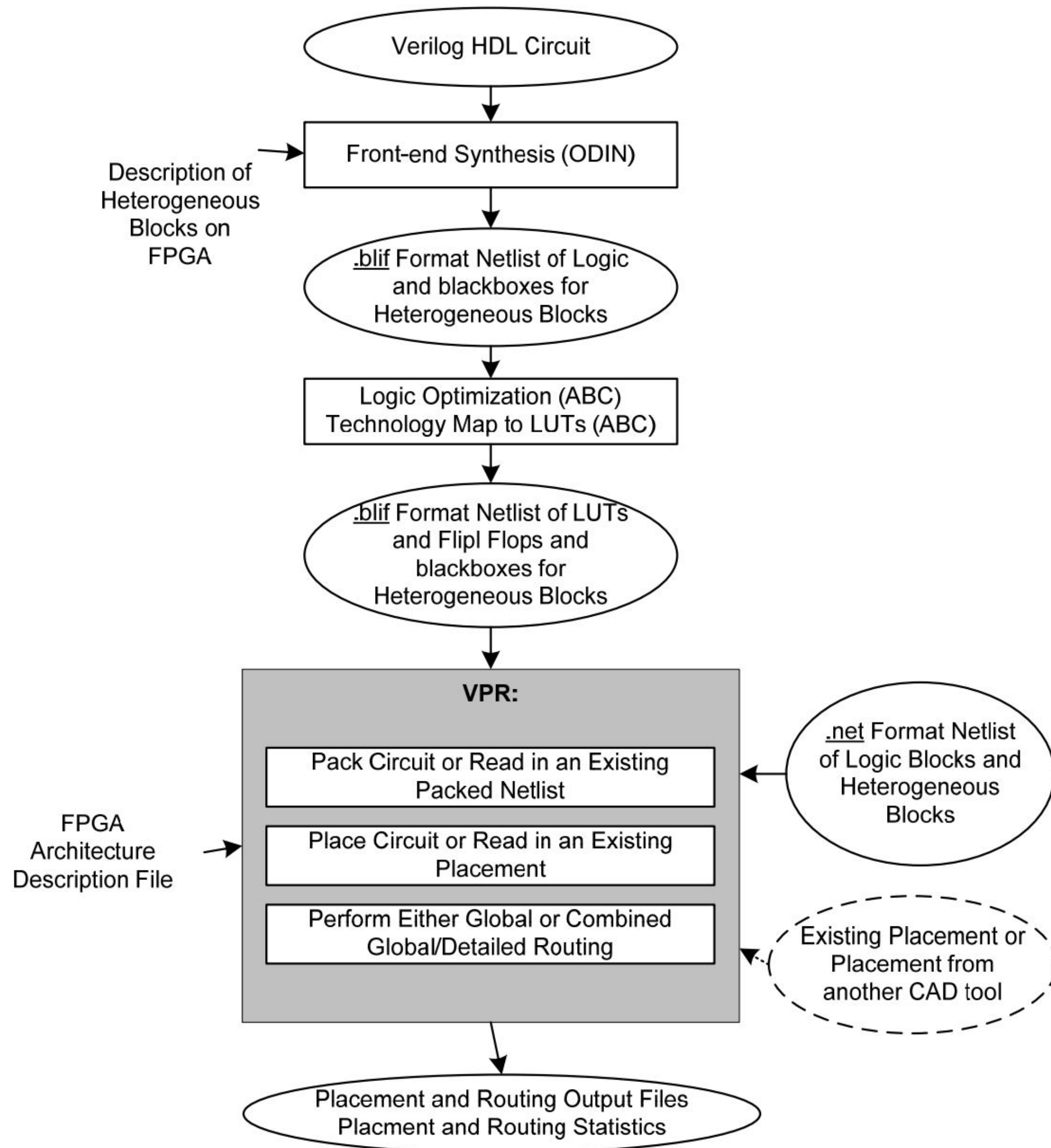

Figure 2.6: VTR FPGA CAD Flow [5]

description of the command line arguments which can be used while invoking the tool. Another powerful feature of VPR is its graphics option. ENABLE_GRAPHICS when set to true, VPR will show a GUI of the FPGA placement area with its CLBs and IOs in grey color. This really helps to better understand the placement methodology employed by VPR. Also, it helps in case you want to debug your placement results. In our thesis, we extensively used the VPR graphics and also its detailed dump of log file to understand the VPR flow and also debug our implementations within the VPR flow.

2.3 Placement Problem Definition

Placement determines which logic block in the FPGA should be used to implement the corresponding logic required by the circuit. The main two optimization goals of a placement algorithm are minimizing the overall wirelength (wirelength driven placement) and maximize the overall circuit speed (timing driven placement). The three major placement algorithms in use today are the iterative techniques of min-cut (Partitioning based) placement [7] [8] [28], analytic placement [35] [39] [37] [36] and the last is the simulated annealing based placement [2] [1].

This section describes about the timing driven quadratic programming approach of placement used in our thesis. We will discuss about the quadratic placement in 2.3.1, which will be followed by timing driven placement methods implemented in the past and the one we chose to implement in 2.3.2. Finally, in 2.3.3, we will give an overview of the GORDIAN-like iterative partitioning and cell-spreading methodology which we have followed in our research work.

2.3.1 Introduction to Quadratic Placement Approach

The connectivity between two or more cells or nodes in a circuit is modeled as a spring in the quadratic placement framework. The total potential energy of the spring is the quadratic function of their lengths (wirelength). It is this potential energy which needs to be minimized to achieve the optimization goal of reduction in wirelength. The most common objective function for placement is the Half Perimeter Wirelength (HPWL) over all nets. First all the multi-pin nets need to be treated as a set of two-pin nets. In order to better explain the quadratic placement methodology, let N be the number of

movable nodes i.e. CLBs in the circuit and (x_i, y_i) be the coordinates of the center of the CLB i. Two N-dimensional vectors X = (x_1, x_2, ..., x_n) and Y = (y_1, y_2, ...,y_n), represent the placement of the CLBs. Consider a net between two CLBs i and j in the circuit. Let Wij be the weight on the net. Then, the total cost of the net between the two CLBs is given by:

$$\Phi(x, y) = \frac{1}{2}W_{ij}[(x_i - x_j)^2 + (y_i - y_j)^2] \tag{2.1}$$

Most analytical placers try to minimize this weighted sum of the squared lengths given in (2.1). The objective function that sums up the cost of all the nets can be written in matrix notation as [10]:

$$\Phi(x, y) = \frac{1}{2}x^T Q_x x + c_x^T x + \frac{1}{2}y^T Q_y y + c_y^T y + constant \tag{2.2}$$

where Q is a NxN symmetric positive definite matrix representing the connections between movable cells (CLBs) and c_x and c_y are N-dimensional vectors representing the connections between movable cells and fixed blocks (IOs). Equation (2.2) is separable into:

$$\Phi(x, y) = \Phi_x + \Phi_y \tag{2.3}$$

For the rest of the discussion, we will concentrate only on the matrices in x-dimension.

$$\Phi_x = \frac{1}{2}x^T Q_x x + c_x^T x + constant \tag{2.4}$$

Minimizing (2.4), involves taking the partial derivative with respect to each variable and setting the resulting system of linear equations to zero.

$$Q_x + c_x = 0 \tag{2.5}$$

The above Eq (2.5), can be solved by a standard linear equation solver. Once solved, the X and Y vectors hold the x and y locations of the CLBs.

To illustrate the Quadratic Placement formulation, consider the figure 2.7, with two fixed blocks (squares) and two movable blocks (circles) i and j. In x-dimension, assuming unit connection weights, the objective function is:

$$\Phi_x = (x_i - 1)^2 + (x_i - x_j)^2 + (x_j - 3)^2 \tag{2.6}$$

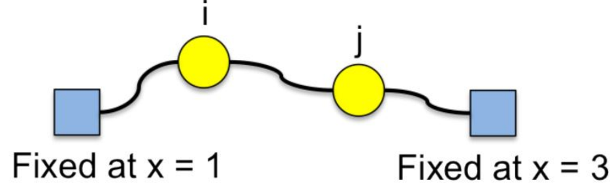

Figure 2.7: Dummy Circuit to illustrate Quadratic Placement Formulation [37]

which can be minimized by taking:

$$\frac{\partial \Phi_x}{\partial x_i} = 2(x_i - 1) + 2(x_i - x_j) = 0 \tag{2.7}$$

and

$$\frac{\partial \Phi_x}{\partial x_j} = -2(x_i - x_j) + 2(x_j - 3) = 0 \tag{2.8}$$

where the linear system is defined by (2.7) and (2.8) can be divided by 2 and expressed in matrix form as:

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} x_i \\ x_j \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

This is a linear system in the form of Eq (2.5), which gives the solution to the unconstrained problem of minimizing the quadratic function in (2.4). Observe that Eq (2.1), does not take any placement constraints into consideration, hence, the generated solution is non-legal with CLBs overlapping one another. In order to remove this overlap and spread the CLBs, we use the GORDIAN-like partitioning technique [29] explained in detail in subsection 2.3.3.

2.3.2 Timing Driven Placers

Timing driven placement is specifically designed to target wires on timing critical paths. Usually, a cell is connected to two or more other cells. Thus, making few targeted nets shorter during placement may sacrifice the wirelength of other nets that are connected through common cells. In other words, the delay on critical path may decrease, however, there might be other paths which become more critical than before. Hence, timing driven placement has to be performed in a very careful and balanced manner [16].

Timing analysis [15] is basically done for two main purposes:

- To determine the speed of circuits which have been completely placed and routed
- To estimate the slack of each source-sink pair during various stages of CAD flow, in order to decide which connections must be made via fast paths to avoid bad slack on them

Many timing-driven placers for ASIC and FPGA designs have been developed [18] [28] [1] [17]. Timing driven placement can be classified into two categories: *net-based placement* and *path-based placement.* In net-based approach, only the nets are dealt with the hope that if the nets on critical paths are handled, then the entire critical path delay will be optimized indirectly. This is achieved using two famous techniques - net weighting [19] [20] [22] [21] and net constraints [24] [25] [33] [26] which guide the timing driven placement engines. The main idea of net weighting is to assign higher net weights to timing critical nets. Higher net weights translate into shorter wirelengths, thus reducing the critical path delays. Net weighting technique gives direction for timing optimization by shortening the critical nets, however, it does not have total control since the objective is the total weighted wirelength. Net constraint generation, on the other end, strives to distribute slack for each path to its constituent nets such that a zero-slack solution is obtained. The delay budget for each net is translated into its wirelength constraint during placement. Both these techniques can be used in iterative placement algorithms and the numbers can be refined in every next iteration.

The path-based approach directly works on all or a subset of paths [27]. Primarily, this approach gets the problem formulated into mathematical programming framework like linear programming. Both net-based and path-based differ from each other with respect to runtime/scalability, ease of implementation, controllability etc. Generally, path-based approaches posses more accurate timing view and control, but suffer from poor scalability. The net-based techniques, in particular net weighting, have low computational complexity and high flexibility. Modern timing driven placement techniques tend to use a mix of both worlds.

In our research work, we concentrate on net-based timing driven placement approach using timing criticality aware net-weighting technique as discussed in subsection 3.3.6.

2.3.3 GORDIAN-like Approach

As pointed in subsection 2.3.1, the quadratic optimization problem does not take into account any placement constraints, thus, resulting in overlap of cells. In order to avoid these overlaps, there are two ways generally used in quadratic placement. The first is to add center-of-mass constraints to prevent cell/CLBs from clustering together and the second is to add forces to pull cells from dense regions to spare regions. In both the ways, the constraints/forces are added in iterative manner to gradually spread out the cells. In our thesis, we use the technique of adding center-of-mass constraints, first introduced by GORDIAN [29] but in a slightly modified manner.

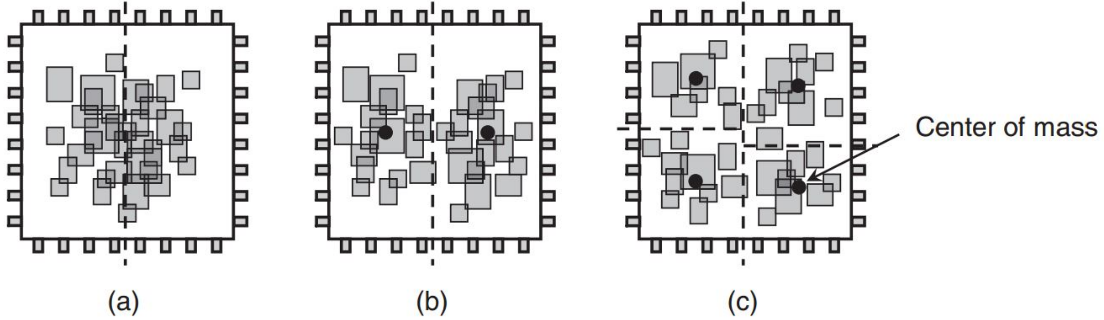

Figure 2.8: Gordian Method of Spreading by Center of Mass Constraints[47]

Our proposed algorithm is implemented as described ahead. Once the initial placement of CLBs is obtained, the FPGA placement region is divided into 2^{2i} sub-regions, where i is the current iteration number, using one horizontal and a vertical cutline. Then, for each sub-region, a constraint in the x-direction and in the y-direction is added to force the center-of-mass of all its CLBs to be at the center of the corresponding region.

$$A_{lin}x = B_{xlin} \tag{2.9}$$

where the entries of matrix A_{lin} are all 0 except for those non-zero entries corresponding to the sub-region that a given CLB belongs to. Additionally, each CLB is provided with the upper and lower bound constraints known as boundary conditions, so that they stay within the specified boundaries after the next placement iteration. The placement problem with these additional constraints is solved again. The center-of-mass constraints

pull the CLBs away from each other near the center-of-mass of each sub-region as shown in fig 2.8. This procedure is applied hierarchically to improve the distribution in each sub-region. Note that, Gordian does not use partitioning to reduce the problem size, but to restrict the freedom of movement of the cells. At each hierarchical level, the placement of all the CLBs is considered together as a single global optimization problem. The coordinates of the center-of-mass are the area weighted mean values (i.e. linear functions) of the CLB coordinates. Hence, the global optimization problem at each hierarchical level is a convex quadratic program, which is equivalent to solving a system of linear equations.

2.4 Previous Work

A tremendous amount of research has been done in the field of placement for FPGAs. Broadly, this research can be classified into simulated annealing based placement, partitioning based placements, analytical placement. Each of these categories can either be wirelength driven placers or timing-driven placers or both. In [1], a simulated annealing based timing driven placement algorithm for FPGAs introduced. This algorithm is an enhancement to the existing placement algorithm within the state-of-the-art simulated annealing based placer - VPR [2] [3]. It uses both wirelength-driven (optimize the Half Perimeter Wirelength - HPWL) and timing-driven (optimize the critical path delay) model with a timing tradeoff factor (default at 0.5) to balance both wirelength and timing costs. VPR achieves very high quality results in terms of wirelength and timing, however, it tends to have long CPU runtimes for larger complex circuits.

Second category of placement algorithms include the partitioning based placements. A routing-aware partitioning based placement for FPGAs is proposed in [28]. The authors have proposed a new alignment cost term in the auto-normalizing cost function of VPR [2] to minimize the delay of the circuit. Low temperature simulated annealing is used towards the end in order to generate higher quality results. Although, this method achieves 4-fold speed up than VPR, it suffers from quality loss.

Analytic/quadratic placement methods are further sub-divided into partitioning based methods [29] [30], density based methods [31] [33] and cell shifting based methods [32]. They are known for being fast and providing good quality results for large complex

ASIC designs. They typically employ a global view of the placement problem [34] [29] [30]. With the advancement of technology and reducing feature size, FPGAs started becoming more and more dense and complex. As a result to achieve faster compilation times, quadratic placement methods for FPGAs became the need of the day. QPF [35] claims to be the first attempt at quadratic placement for FPGAs. It concentrates on optimizing quadratic wirelength by mapping the circuit on the chip and adding dummy nodes to expand the placement and iteratively solving linear equations using conjugate gradient method. Linear adjustment is done to optimize the linear and quadratic wirelength. Finally, low temperature simulated annealing is performed to refine the overall quality of placement. QPF claims to be 5.8 times faster than VPR and provides comparable total estimated wirelength when tested on homogeneous FPGA benchmark circuits. Another analytical placement tool developed for homogeneous FPGA architecture is StarPlace [36]. StarPlace proposes a new near-linear net model called star+ which is a modified version of the star model used in [32] to estimate the wirelength of a net. Star+ tries to minimize the over-estimated squared wirelength by taking the sum of the square roots of the sum of the quadratic distances between each block and the center of gravity of a net. It proposes two different analytical placers, one using conjugate gradient method and another based on successive over-relaxation for solving the resulting system of non-linear equations. StarPlace achieves 8-9% reduction in critical path delay and 5x speedup compared to VPR. HeAP - A work targeting heterogeneous FPGAs comprised of LUT-based logic blocks, multiplers/DSPs and block RAMs is shown in [37]. HeAP adapts the framework of a state-of-the-art ASIC based analytic placer, SimPL [38] to target FPGAs with heterogeneous blocks located at discrete locations throughout the FPGA fabric. It adopts the bound2bound net model used in [38] and first proposed in [31], which gives high quality solution as it directly models the HPWL. It is known to outperform industry standard timing and non-timing driven FPGA tool from Altera in terms of runtime with 4-5% increase in overall wirelength. Another work in the field of analytical placement of homogeneous FPGAs found to achieve better wirelength and ciritcal path delay compared to VPR is [39]. The authors of [39] claim it to be the first academic multilevel timing and wirelength driven analytical placement algorithm. They propose a multilevel framework for global placement with block alignment consideration calculated based on the alignment cost to minimize the wirelength. To enhance the

wirelength estimation accuracy they use Stable Log-Sum-Exponent model proposed in [40]. Conjugate Gradient method is used to solve the system of equations. A look-ahead legalization is performed using Gordian based methodology [29]. For wirelength-driven detailed placement a window-based bipartite block matching technique [41] is used and timing-driven detailed placement is optimized by doing a low temperature simulated annealing in VPR.

2.5 Motivation and Research Goal

With increasing chip density and decreasing feature size, the density of FPGAs is ever-increasing. The FPGAs are becoming more and more complex each year. Placement is one of the most time consuming stages in the CAD flow for FPGAs. Developing a fast and efficient placement algorithm will help lessen the overall CAD tool run-time. Simulated Annealing (SA) based methods [2] provide best results, however, they tend to be a bottleneck for large circuit placements. In ASIC domain, placer must handle designs having millions of gates, and SA based methods have almost been abandoned there. Despite this fact, SA continues to be a popular placement technique for FPGAs [4], [2]. One of the giants in FPGA industry, Altera, too uses simulated annealing based iterative strategy [42]. Two main hurdles lie in the way of developing a better CAD placement algorithm:

- The placement stage should be fast enough and provide better quality of results (QoR) for large complex circuits
- There should not be any overlaps in the final placed circuit

Our research goal is to develop a timing-driven iterative analytical placement algorithm targeting homogeneous FPGAs, which is fast and provides better quality of results in terms of critical path delay and wirelength. Our placement will be a legalized placement which is free from any overlaps, developed using a spiral overlap removal technique.

We chose to implement our work in MATLAB and use VTR [4] framework to carry out the other stages of the FPGA CAD flow. We had planned to modify and use the basic framework developed by the team of HeAP [37]. We asked the authors for the source code, however, we did not hear back from them. Hence, we decided to use VTR

because it is an academic open-source state-of-the-art tool and widely used in FPGA CAD research.

2.6 Summary

In this chapter we discussed about the island style FPGA architectures and described the VTR flow. We also talked about the quadratic programming framework, timing driven placement methodologies and previous research work undertaken in this field. Lastly, we end by outlining our motivation and research goals in carrying out this work.

Chapter 3

Quadratic Placement CAD Flow

In this chapter, we present our entire FPGA quadratic placement CAD flow. We will first provide an overview of the entire timing driven CAD flow that we developed in 3.1. Next, the section 3.2 discusses about the Analytical Placement Engine developed in MATLAB. Towards the end of this chapter, in section 3.3, we will provide the details about the various functions that were used in developing the analytical placement engine and the decision on some of the important parameters used.

3.1 Timing Driven CAD Flow Overview

Figure 3.1 below illustrates the CAD flow developed by us. We start the flow by invoking VPR tool and providing the architecture file and the benchmark circuit. Once VPR finishes packing of the logic elements into clustered logic blocks (CLBs), we generate the connectivity matrix. The connection matrix is a symmetric matrix of size (N X N), where N is the number of blocks. Number of blocks includes all the IOs and CLBs present in the benchmark circuit. This matrix forms one of the inputs given to our MATLAB Analytical Placer. Additional details about generation of connection matrix is provided in 3.3.1. Once VPR completes its placement step i.e. it places the CLBs and IOs in their optimized location based on its simulated annealing mechanism, it generates a file containing the placement information. We term this VPR placement as the Reference Placement. We use the data of IO locations from this reference placement and give it as an input to our MATLAB engine. We also let VPR dump out the final placement

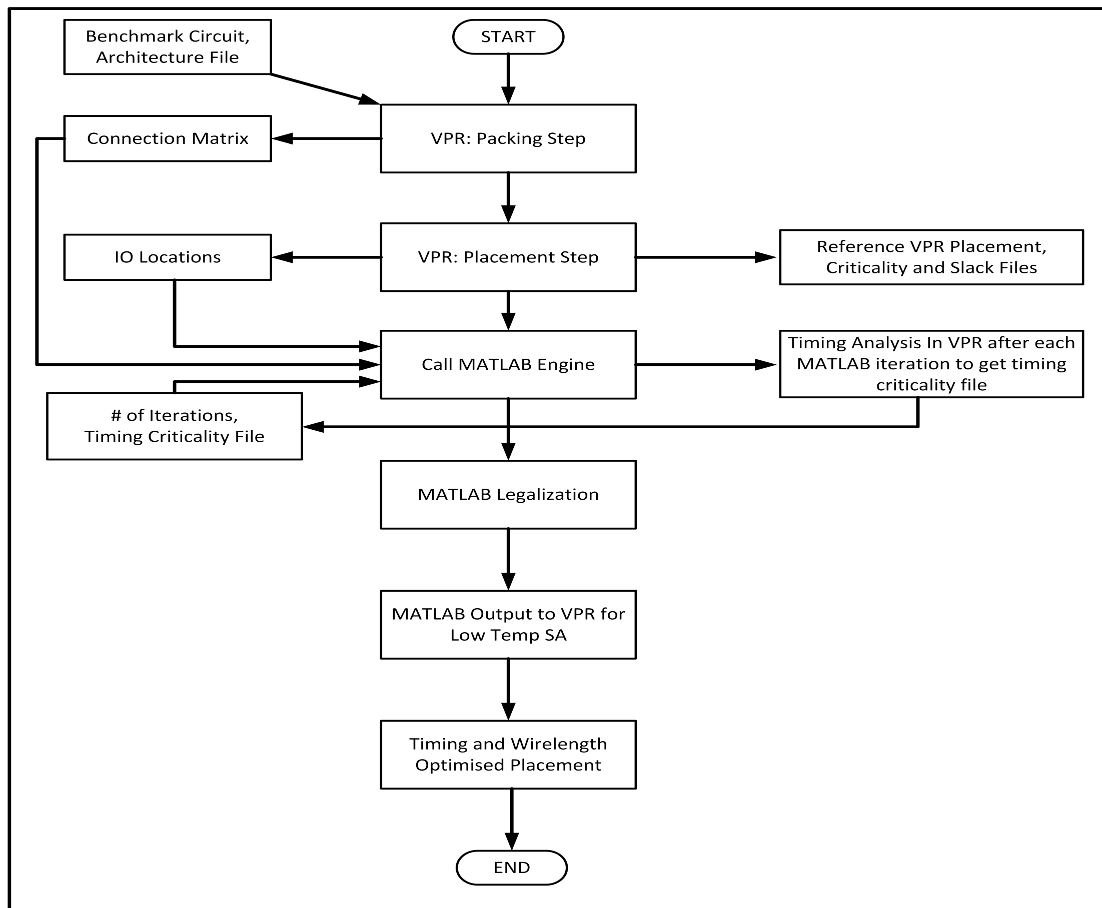

Figure 3.1: Timing Driven CAD Flow Overview

criticality, final placement slack, wirelength and the critical path delay information which we use to compare VPR with our placement engine. This analysis is shown in detail in Chapter 4.

Once VPR is done with its reference placement, the MATLAB engine is invoked. The specific functions used in MATLAB flow are described in 3.3. After each MATLAB iteration, an intermediate placement of IOs and CLBs is generated and given to VPR to do timing analysis on it and obtain the slacks and timing criticalities. This criticality between each CLB block, obtained from the timing analysis, is used as a reference to update the weights in the connection matrix during the next iteration. The final

iteration MATLAB output is a legalized placement of CLBs on the FPGA fabric. This placement is fed back to VPR to do detailed placement via low temperature simulated annealing. The placement obtained at the end of this flow is our final refined legalized placement output.

3.2 MATLAB CAD Flow

The MATLAB CAD flow is shown in figure 3.2. The analytical placement engine takes the connection matrix generated earlier by VPR as the input along with the I/O locations from the VPR reference placement. There is a provision for one more input to the placement engine and it is the timing criticality information of the nets generated by VPR after each placement iteration in MATLAB. This information is used to update the weights in the connection matrix. In every iteration, we provide the criticality information to our placement engine. The decision on calculating the number of iterations for which the placement engine will be called, is explained in detail in 3.3.4.

When MATLAB analytical placement engine is called for the very first time (zeroth iteration), the timing criticality information is unavailable, hence we take the weights of all connections in the connection matrix to be equal to the fanin of the CLB as explained in 3.3.1. We use this weighted connection matrix to run the zeroth iteration and carry on with the algorithm described ahead. The total number of available physical locations on the FPGA grid in which the post packed CLBs can be placed is a function of the number of CLBs that VPR generates after packing step and sideSize. The sideSize is calculated as the ceil of square root of the total movable blocks i.e. the total number of CLBs obtained after the packing step of VPR. This formula is adapted from VPR framework [1] [5] and the main motive behind this is that we want to use as minimum FPGA resources as possible as far as total available physical placement locations are concerned.

$$sideSize \ = \ ceil(sqrt(Total\ CLBs)) \tag{3.1}$$

We want to map each circuit into the smallest FPGA possible, hence, the total number of available locations is equal to sideSize*sideSize. So, if we have the circuit packed into 10 CLBs by VPR, the total number of available physical locations for these 10 CLBs will be equal to 4*4 which is 16 locations.

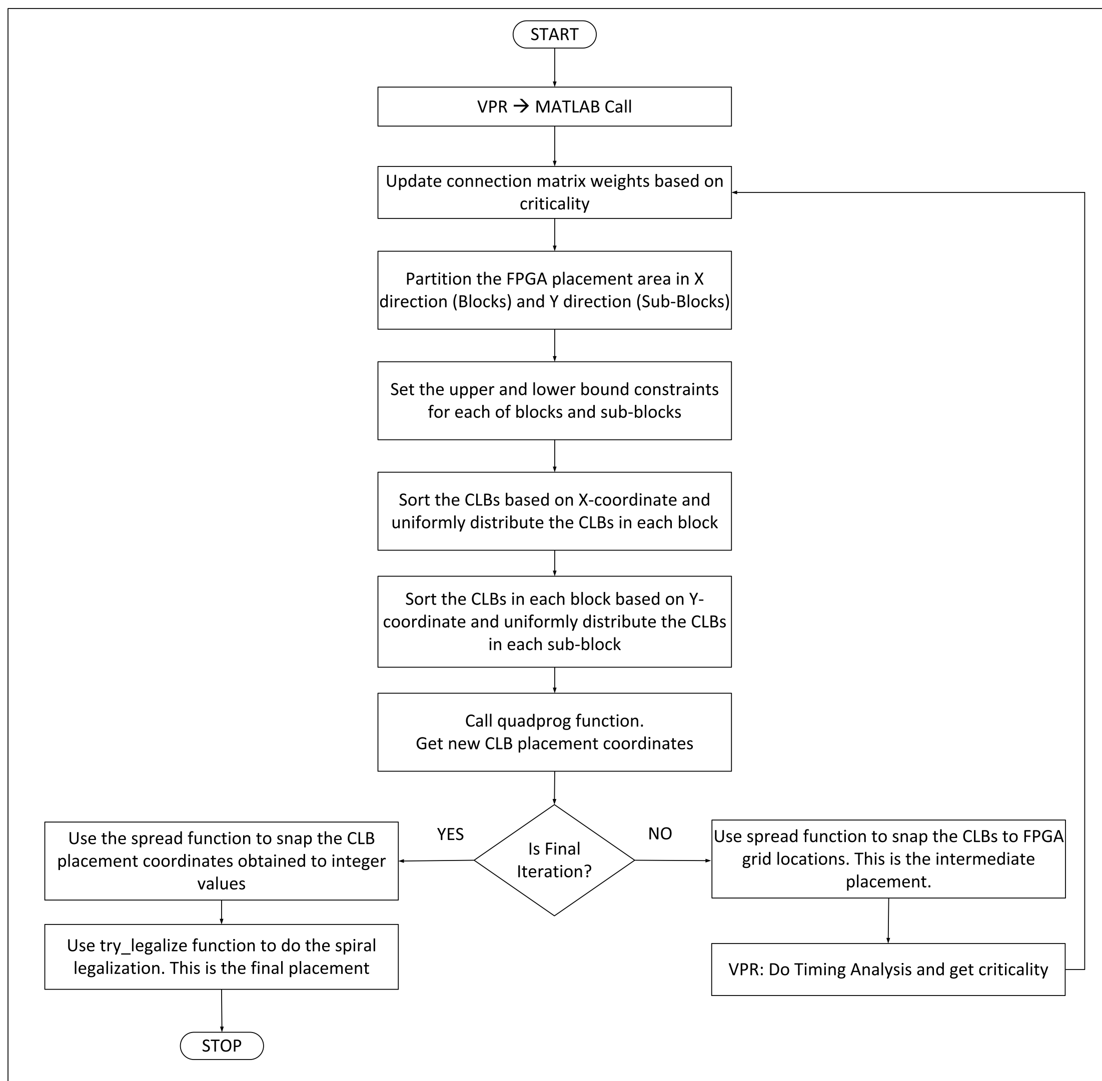

Figure 3.2: MATLAB CAD Flow

Once the connection matrix entries are updated with appropriate weights, we divide the placement area for CLBs on the FPGA chip into different blocks. Here, each block refers to a vertical partition i.e. vertical cutline. The number of blocks on the FPGA fabric for a particular iteration is decided by the formula 2^{I}, where I is the current iteration number.

$$Number\ of\ Vertical\ Partitions\ or\ Blocks \ = \ 2^{\text{Current Iteration Number}} \tag{3.2}$$

Hence, for zeroth iteration, the FPGA will not be divided into any block; it will be the complete FPGA placement area. This is because I=0 for zeroth iteration and going by the Eq (3.2), we have number of blocks equal to 1 i.e. the entire FPGA placement area. For all other iterations, the placement area is divided into number of blocks given by Eq. (3.2). The next step distributes the CLBs to each block on the basis of their X-coordinates sorted in ascending order. The number of CLBs going in each block is calculated using the formula:

$$CLB\ Count\ In\ Each\ Block = \frac{Total\ CLBs}{Number\ of\ Blocks} \tag{3.3}$$

Along with this, we set the boundary conditions of lower and upper bounds in X-direction on each of the CLBs depending on the block they are put into. Now, in the same iteration, each block is further divided into 2^{I} number of sub-blocks i.e. horizontal cutline. The CLBs are now sorted based on their Y-coordinates and distributed in the corresponding sub-blocks of the blocks they were assigned in the previous step. Similar lower and upper bound constraints are now assigned in Y-direction. Additionally, every CLB is associated with a tag which denotes the sub-block in which that particular CLB was made to go. All this data is passed as arguments to quadprog function of MATLAB, which generates a new placement of CLBs in each iteration. This new intermediate placement, however, is not legal and also it does not adhere to the physical FPGA grid locations. Hence, we take the X and Y coordinates of the new placement and use floor and ceil functions on them to get integer coordinates, and snap them to the physical FPGA grid locations. This is mainly done so that VPR can perform timing analysis on this intermediate placement result and generate the timing criticality information. This information will then be used by the next iteration of the placer to update the weights in connection matrix. The quadprog function of MATLAB works

faster for sparse matrices. So, to improve the runtime of the MATLAB analytical placer, all odd iterations undergo an update in the net weights in the connectivity matrix for timing criticality values greater than a threshold limit and remaining values are made 0. In all even iterations, the timing criticality of all the nets is considered and the connectivity matrix weights are updated taking all these nets into consideration. This type of alternate updates to connectivity matrix, thereby making the matrix sparse and dense in subsequent iterations, not only improves the placer runtime, but also takes care of optimizing the timing critical connections.

In the final iteration of the analytical placer, it does a spiral legalization instead of using floor and ceil functions and then MATLAB exits.

3.3 Placement Engine Functions and Parameters

In this section, we describe the important functions developed for our analytical placement engine and also few important parameters and how did we decide on them.

3.3.1 The Connectivity Matrix

The connectivity matrix is a (N X N) sized symmetric matrix, where N is the Number of Blocks in the benchmark circuit. It contains the connectivity information about the clustered logic blocks (CLB) that VPR generates post packing. We generate the connection matrix at the end of packing step in VPR.

Once VPR finishes packing, with the help of data structures in VPR as well as a new data structure s_clb_info created by us within VPR, we develop the connection matrix. Details regarding this new data structure are given in the Appendix B. The procedure for generation of connection matrix is as follows:

- Select the one CLB from the list of CLBs.
- Loop through each input pins of the CLBs and check if they are OPEN or not.
  - If input pin is marked as OPEN, continue to another pin.
  - If input pin is not OPEN, then it'll have an pin index. Use this pin index to search for the net name connected to this pin in vpack_net data structure.

- As per VPR 7.0 User Manual [5], the net name is the name of the leaf level primitive block (LUT or Flip-Flop) driving it. This same net name becomes the input pin name too.
- Once the net name or rather the leaf level primitive name is obtained, check in which CLB this particular primitive is packed into. VPR logical_block data structure's member variable clb_index is used in this case.

For every CLB, as each input pin is parsed, the clb_index of its driver CLB is stored in the connections array member variable of s_clb_info data structure. This information will be used while printing out the connection matrix to a file. In this way the inter-CLB connectivity is obtained.

By default, the connection matrix generated by the above method is such that if two blocks(CLB-CLB, CLB-IO or IO-IO) are connected to each other, they will have 1 in their corresponding row and column. We have also kept a provision where we can put the total number of connections a particular CLB makes with an other CLB or IO as the number in their corresponding row and column. For example, if CLB A has its 4 input pins driven by 4 output pins of CLB B, then in the row and column corresponding to CLB A and CLB B we place the value 4. All the other entries will have 0 in them. Indexing of the rows and columns in the connection matrix is done in a specific order. All IOs occupy the initial indexes and then the CLBs are indexed. For eg: if there are 41 IOs and 20 CLBs, then IOs will have rows and columns indexed from 0 to 40 and then the CLB row and column indexing starts from 41 to 60. The final default style of connection matrix generated at the end of VPR packing stage is made up of 1s and 0s. It is non-weighted matrix of connections. After every iteration, based on the timing criticality of the nets, appropriate weights are set in the connection matrix. The methodology of applying weights is discussed in 3.3.6.

Using the weighted connection matrix, two new matrices are generated - Matrix (A) having only inter-CLB connectivity information along with the weights associated with each net and the matrix (B) with CLB-IO connectivity information. The matrix B is generated separately for X and Y coordinates. They are named Bx and By respectively.

3.3.2 quadprog Function

The heart of our analytical placer is the quadprog function in MATLAB. It is a matrix solver provided by the optimization toolbox in MATLAB. As described in section 2.3.1, the quadratic placement approach uses springs to model the connections in the circuit. The total potential energy of the springs, that is a quadratic function of the length, is minimized. Quadratic programming is the problem of finding a vector x that minimizes a quadratic function subject to linear constrains. The quadprog function uses a interior-point-convex algorithm which attempts to follow a path that is strictly inside the constraints specified.

We separately solve for the X and Y coordinates. The quadprog function helps minimizing the equation of the following type:

$$min\ \frac{1}{2}x^T Ax + B_x^T x$$

$$subject\ to\ A_{lin} * x = B_{xlin}$$

$$B_{xlb} \leq x \leq B_{xub}$$

Here, A and B_x are the matrices described in 3.3.1. Matrices A_{lin} and B_{xlin} help to set a constrain that the average of all X-coordinates of the solution obtained by quadprog is the X-coordinate of the Center of Gravity of that block. This is based on the GORDIAN Placement methodology [29]. B_{xlb} and B_{xub} are the matrices which set the upper and lower bounds for each block and sub-blocks created on the FPGA chip. Similar equations are used to solve for Y-coordinates.

3.3.3 try_legalize Function

One of the problems associated with quadratic placement solvers is that no constraints or information about the legal FPGA grid locations is supplied to the quadratic solvers. As a matter of fact, the solution obtained, contains a lot of overlap of cells. Figure 3.3 shows a 4x4 FPGA grid. The square grey blocks represent the legal FPGA grid locations and the yellow small blocks represent the solution of the final quadprog iteration. Since, there is no constraint regarding the legal grid locations specified as an input to the quadprog, it does a good job to minimize the wirelength but returns a non-legal placement. A good

analytical placement algorithm should be able to handle this issue without deteriorating the placement considerably in terms of timing criticality and wirelength.

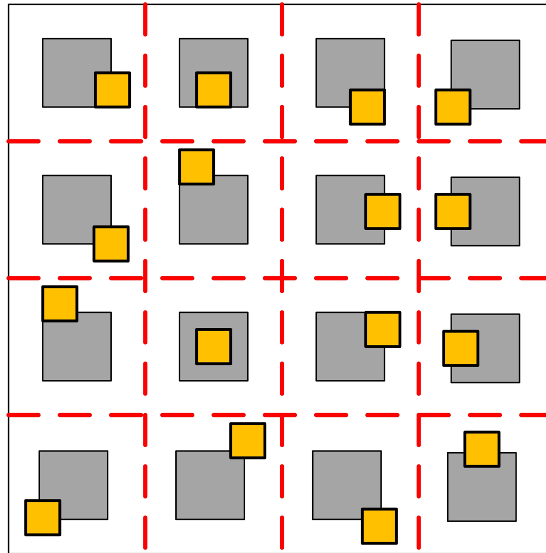

Figure 3.3: 4x4 FPGA g rid with grey blocks representing physical grid locations and yellow blocks representing CLB placement by final quadprog iteration

In an attempt to legalize the final placement, we implemented a couple of different algorithms. First we use floor and ceil functions to snap the CLBs to integer value X and Y coordinates on the FPGA grid. We initially started with an algorithm to randomly place the overlapped cells onto free locations available in the FPGA grid. However, this was not a very good method to be employed for legalization as it didn't take care of wirelength nor timing criticality. Next, we tried to legalize the CLBs by going to an overlapped location, picking the CLBs which are overlapping and spread them to free locations by traversing in zig-zag method diagonally. This method made the wirelength worse than the previous method since in some instances, the CLB of one sub-block ended up in completely far away sub-block. Hence, the initial premise of iterative partitioning of the FPGA chip and dividing the CLBs equally in each block was defeated.

We then came up with an algorithm in which we go to a highly populated overlapped grid location and check for the timing criticalities of the CLBs in that location. We leave the highly timing critical CLB in that same place so that it's criticality is not affected

and pick up the remaining less critical CLBs and move in spiral order starting from the non-legalized position. In this spiral run, whenever an empty location is encountered, we drop the CLB we are carrying in that location and keep moving ahead in spiral order until all the overlapped CLBs we are carrying are depleted. Once, a particular overlapped region is devoid of overlaps, we move to another location on the FPGA grid having the same number of overlaps if they still exits or else go to a location with lesser than previous location overlaps. Basically, starting from highly overlapped FPGA grid location, the algorithm goes on legalizing in descending order of number of overlaps present in the FPGA grid location. Mainly, this method is better than the previous methods because, it checks the timing criticality of the CLB in an overlap location before making an attempt to legalize them. Also, since we move in spiral manner, we observed that the CLBs are placed in empty location which generally lie in the same or adjacent sub-block, thus, not degrading the total wirelength to a very high extent as the random placement. Figure 3.4 illustrates the method described above pictorially. The red spots are the over utilized non-legal locations and the green spots are the free locations. The grey slots are occupied locations. The number in the brackets at each location denote the total number of overlap at that particular location on the FPGA fabric.

Described below is the procedure of our spiral legalization algorithm. The pseudo code is shown in Algorithm 1.

- Output of the quad prog function i.e. the X and Y coordinates are converted to integer values using the floor and ceil functions.
- overlapCLB array is initialized. This array stores the indices of the CLBs which are overlapping.
- legalityMatrix is initialized to zero. This matrix maps the actual FPGA grid locations and its entries denote the amount of overlap present at that particular row-column location in the FPGA grid.
- Parse each CLB X and Y coordinates and go on incrementing the corresponding entry in legalityMatrix. If more than one CLB has the same X and Y coordinates, increment the current entry in the corresponding location of legalityMatrix by

Algorithm 1 try_legalize Function

1: **procedure** TRY_LEGALIZE($XCord, YCord, final_iteration$) ▷ Function to place the CLBs in legal positions
2: $overlap_CLB \leftarrow NULL$
3: $legality_Matrix \leftarrow 0$
4: $i \leftarrow 1$
5: **for** $i : CLBArraySize$ **do**
6: $legality_Matrix(XCord(i),\ YCord(i)) ++$ ▷ Update the XCord, YCord location of legality_Matrix by 1
7: **if** $legality_Matrix(XCord(i),\ YCord(i)) > 1$ **then**
8: $overlap_CLB \leftarrow [overlap_CLB\ i]$ ▷ Location value greater than 1, implies overlap
9: **end if**
10: **end for**
11: **while** $overlap_CLB \neq EMPTY$ **do**
12: $xAddress \leftarrow Xcoordinate\ of\ max\ overlap\ location$
13: $yAddress \leftarrow Ycoordinate\ of\ max\ overlap\ location$
14: $Check\ the\ timing\ criticality\ of\ all\ CLBs\ at\ this\ location$
15: $Pick\ up\ less\ timing\ critical\ CLBs\ and\ rotate\ sprially\ in-counter\ clockwise\ direction\ until\ all\ CLBs\ legalized$
16: **if** $legality_Matrix(Xnew,\ Ynew) \leftarrow 0$ **then**
17: $Found\ empty\ location.\ Put\ theCLB\ at\ this\ location$
18: $legality_Matrix(Xnew,\ Ynew) \leftarrow 1$
19: $legality_Matrix(xAddress,\ yAddress) --$
20: $Remove\ the\ CLB\ Index\ from\ overlapCLB\ array$
21: **end if**
22: **end while**
23: **return** $(XLegal,\ YLegal)$ ▷ Output of try_legalize function
24: **end procedure**

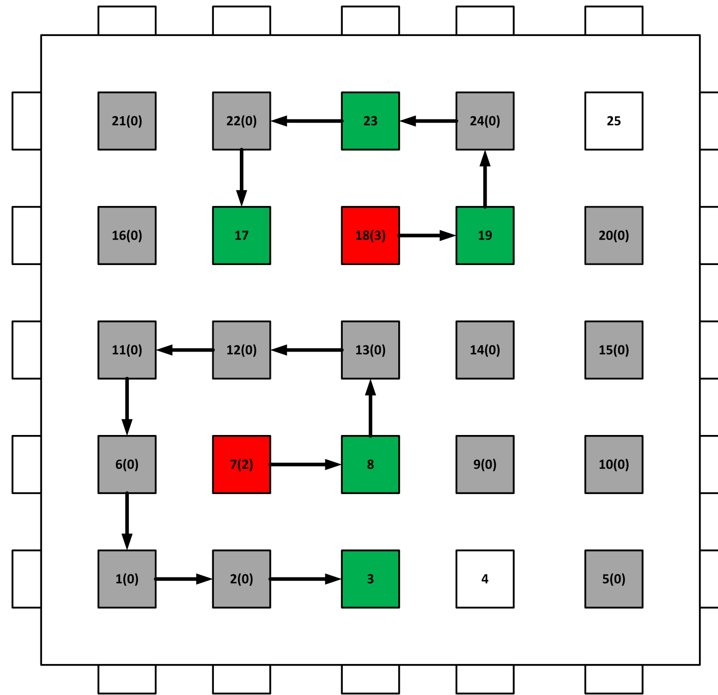

Figure 3.4: Spiral Legalization Pictorial Representation

1, each time a same entry is obtained. Also add the index of such CLBs in the overlapCLB array.

- The CLB index is same as the row or column index of that CLB in the connection matrix.

- Find the location having the maximum CLB overlap and check for timing criticality of all CLBs at that location. Leaving behind the CLB which is highly timing critical, pick up the remaining and start spiral legalization from that location. Spiral movement is in anti-clockwise direction starting from the overlapped region.

- If all the CLBs of a particular location are legalized, break from the loop and go to the next overlapped location on the FPGA.

- As the CLBs find empty locations and get legalized, the legalityMatrix is updated. The overlapped location entry in the matrix is decremented by 1 and the empty

location where the CLB was placed for legalization is incremented by 1.

- If all the CLB indexes in the overlapCLB array are legalized, exit from the procedure.

In this manner, we achieve a legalized placement as the output of final iteration of the analytical placement engine. This legalized placement is then given to VPR for low temperature simulated annealing detailed placement.

3.3.4 Number of Iterations

Quadratic placement with partitioning is an iterative algorithm. In each iteration after the zeroth iteration, the FPGA chip is partitioned into 2^{2*I} partitions. Once the number of partitions is decided and CLBs are distributed uniformly among all partitions, the boundary conditions are set and analytical placement engine is run to give the CLB placement. The number of iterations for which the MATLAB engine is called depends on the total number of movable blocks a.k.a. the CLBs in which the circuit is packed into by VPR. This is calculated using the following formula:

$$Number\ of\ Iterations\ =\ floor(\frac{log_2(Number\ of\ Movable\ Blocks)}{2}) \tag{3.4}$$

We aim to put atleast one or at the most four CLBs in each partition. The upper limit to the number of CLBs going in a particular partition also depends on the size of the FPGA. For instance, if the circuit is packed into 5 CLBs, the size of the FPGA will be 3x3 i.e. 9 available physical grid locations based on the Eq (3.1). By the above Eq (3.4), it will have one iteration following the default zeroth iteration, which implies there will be a total of four partitions (sub-blocks) in the first iteration. Since, our CAD flow is designed to make the CLB density in each partition almost uniform, we will see that there will be one CLB each in three sub-blocks, whereas, the last sub-block will have two CLBs. In case, in some bigger circuit, if all the partitions are filled with 4 CLBs each, the number of iteration is increased by 1 in an attempt to reduce the number of CLBs from four to less than four in each partition of the next iteration. This technique of deciding the number of iteration helps us to make the legalization problem easier during the final iteration because, we might end up only with less than 4 CLBs in most of the sub-blocks.

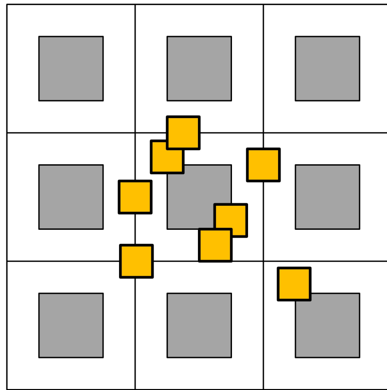

Figure 3.5: 8 CLBs - Iteration 0

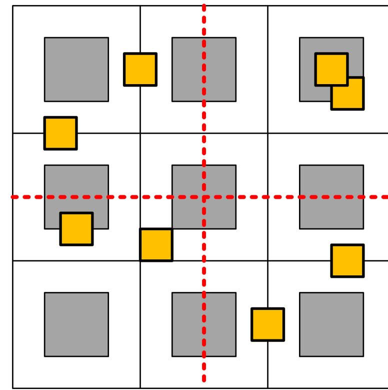

Figure 3.6: 8 CLBs - Iteration 1

To better explain this, lets take three examples - a circuit packed into 8 CLBs, a circuit packed into 15 CLBs and last about a circuit packed into 16 CLBs.

- **Circuit Packed into 8 CLBs**

  In this example, we have 8 movable blocks i.e. 8 CLBs in the packed circuit. From Eq. (3.1), we know that there are 3*3 = 9 physical CLB locations available on the FPGA chip. Using Eq. (3.4), we have the MATLAB engine undergo 1 iteration after the default zeroth iteration. From Eq. (3.2), the FPGA will be divided into two vertical blocks and each of these vertical blocks will be further divided in to two sub-blocks. So, in total the FPGA placement area will be divided into four sub-blocks in the iteration after the zeroth iteration. This will also be the final iteration in this case. Finally, to decide the number of CLBs in each sub-block, from Eq. (3.3), we derive that there will be 4 CLBs going in each vertical block. Further, these 4 CLBs in each vertical block will be divided into 2 CLBs each going to each of the sub-blocks.

  In summary, for a 8 CLB packed circuit, there will be one iteration after the default zeroth iteration and we have the FPGA placement area divided into 4 sub-blocks as shown in the figure 3.6 and each of the sub-block contains 2 CLBs within them.

- **Circuit Packed into 15 CLBs**

  We take the similar approach as the first case above and find that for a 15 CLB

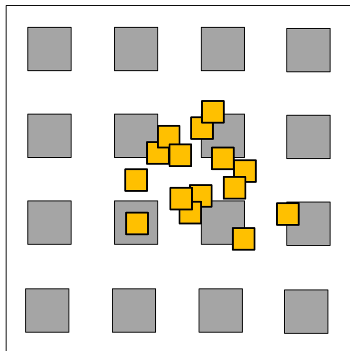

Figure 3.7: 15 CLBs - Iteration 0

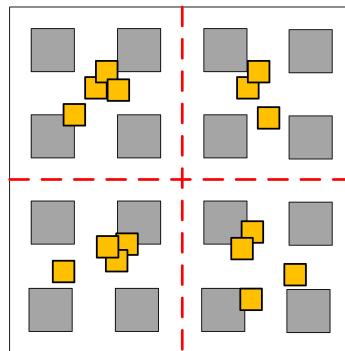

Figure 3.8: 15 CLBs - Iteration 1

packed circuit, we will have 16 physical locations available for the CLBs on the FPGA grid. There will be one iteration after the default zeroth iteration which means that the FPGA placement area, in this case too, will be divided into four sub-blocks. Out of the four sub-blocks, three of them will have 4 CLBs each and the last sub-block will have 3 CLBs as shown in the figure 3.8.

- **Circuit Packed into 16 CLBs**

  In this case, there will be 16 physical locations available for the CLBs on the FPGA grid. However, based on the Eq (3.4), there will be 2 iterations following the default zeroth iteration. Hence, the FPGA placement area will be divided into 4 sub-blocks in the first iteration and each of these 4 sub-blocks will be divided into 4 more sub-blocks in the second iteration, making a total of 16 sub-blocks in the final iteration. This will result in one CLB per sub-block at the end of analytical placement, making the legalization step much easier. Had it been only four partitions i.e. one iteration following the default zeroth iteration as in the case of circuit packed into 15 CLBs shown above, every CLB would have to deal with three other CLBs in each sub-block during legalization. This is because, for 16 CLB circuit and 4 sub-blocks, there would have been 4 CLBs in each sub-block. However, since we added one more iteration of analytical placement, the FPGA placement area got sub-divided into more partitions and each partition now has less than 4 CLBs within them. Figure 3.9, 3.10 and 3.11 show the zeroth, first and second iteration of analytical placement for this case.

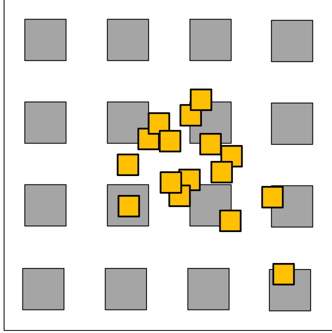

Figure 3.9: 16 CLBs - Iteration 0

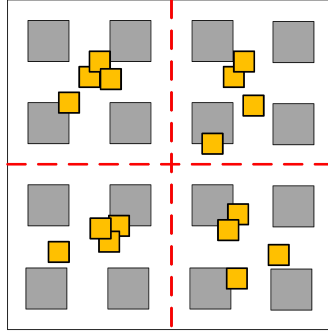

Figure 3.10: 16 CLBs - Iteration 1

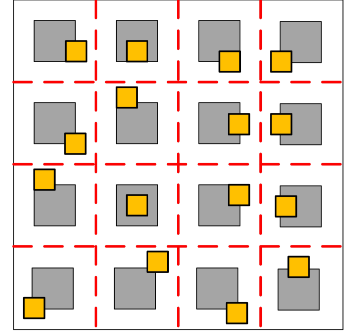

Figure 3.11: 16 CLBs - Iteration 2

3.3.5 Timing Analysis Methodology in VPR

Timing analysis is the soul of our timing driven analytical placement algorithm. The way in which timing analysis is carried out is that initially, the circuit under consideration is presented as a directed graph. Nodes in the graph represent the input and output pins of the circuit elements like LUTs, flip-flops and I/O pads. Connections between these nodes are modeled with edges in the graph. Between the inputs and outputs of LUTs, edges are added and annotated with a delay corresponding to the physical delay between these nodes. Flip-flop input pins are not joined to their output pins. In order to determine the delay of the circuit, a breadth first traversal is performed on the graph starting at sources (input pads and flip-flop outputs). The arrival time ($T_{arrival}$) at all nodes in the circuit is computed using the below equation:

$$T_{arrival}(i) = max_{j \epsilon fanin(i)} T_{arrival}(j) + delay(j,i) \tag{3.5}$$

where node i is the node currently being computed and delay(j, i) is the delay value of the edge joining the node j to node i. The maximum arrival time, D_{max} of all nodes becomes the maximum delay of the circuit.

To guide a placement algorithm, it is important to know the amount of delay that may be added to a connection before the path that the connection is on becomes critical. The amount of delay that may be added is called the slack of that connection. In order to compute the slack of a connection, the required time $T_{required}$ should be computed at every node in the circuit. Initially, $T_{required}$ is set to be D_{max} at all sinks (i.e. Output

pads and flip-flop inputs). Required time is then propagated backwards starting from the sinks with the below equation:

$$T_{required}(i) = min_{j \epsilon fanout(i)} T_{required}(j) - delay(j, i) \quad (3.6)$$

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

$$Slack(i, j) = T_{required}(j) - T_{arrival}(i) - delay(i, j) \quad (3.7)$$

Since, VPR already has a timing analysis engine developed, we decided to use the same. According to our timing driven placement flow, we perform one entire pass of timing analysis after every iteration of MATLAB. In each iteration, the analytical placer generates a new placement of configurable logic blocks. These CLBs, however, do not have integer X and Y coordinate values. So, in order to snap them to the nearest physical grid locations, the spread function (floor and ceiling operator) is used. The output of this function becomes the intermediate CLB placement.

On the completion of each MATLAB iteration, the CLB index, the location of CLBs on the FPGA and their names are written to a file. This file is read in VPR and based on the CLB index, s_block data structure's member variables (x, y, z) are updated. The way in which timing analysis is done in VPR is as follows:

- Calculate the path delays between CLB-CLB or CLB-IO or IO-CLB or IO-IO using the delay matrix already generated by VPR.
- Update the net_delay array with the path delay values calculated. The net_delay [0..num_nets-1] [1..num_pins-1] is a 2D array which stores the delay on a net from its driver to all its sinks.
- When all the net delays have been calculated and loaded in the net_delay array, the timing graph nodes in VPR are updated. The timing analysis in VPR is done with the help of timing nodes called tnodes. The pins in the circuit are converted to tnodes and the nets are transformed into tedges in VPR framework [3].
- Now call the do_timing_analysis function. The arguments passed to this function are slacks data structure pointer, is_prepacked = FALSE, do_lut_rebalancing = FALSE and is_final_analysis = FALSE. This function does a forward pass and

updates the arrival time on all the tnodes. Once the forward pass is completed, a backward pass is done to update the required time on every node. It finally calculates the slacks and timing criticality and updates the corresponding data structures.

- Read the timing criticality generated by the timing analysis pass and in the next iteration of the analytical placement and use this information to update the weights in the connection matrix.

In this way, we perform the timing analysis at the end of every iteration of our analytical placement engine.

3.3.6 Net Weighting Scheme

Conventional placement algorithms optimize the total wirelength. These can be modified quite easily into timing driven using the net weighting scheme. The way this works is, different weights are assigned to different nets such that the total placement weighted wirelength is minimized. A proper net weighting scheme would assign higher weights to more timing critical nets, hoping that the placer will reduce the wirelength of these critical nets and hence, the delays on these paths can be reduced. In this way, better overall timing can be achieved. Net weighting based timing-driven placement is very easy to implement and less computational intensive. Almost all placement algorithms support net weighting. Quadratic placement can optimize the weighted quadratic wirelength, partition-based placement optimizes the weighted cut size and simulated annealing based placements can optimize the weighted linear wirelengths.

While net weighting appears to be simple, it is not easy to generate a good net weighting scheme. Higher net weights on a particular set of critical nets implies, their wirelengths and thus, their delays will be reduced, but other nets may become longer and more critical. In this subsection, we will discuss the net weighting scheme employed by us in our quadratic placer.

We experimented several different ways of net weighting schemes. We started with unity weights in the connection matrix and updated them as the placement proceeded based on the slack values obtained after timing analysis. We considered only those nets where the slacks came out to be negative. However, this method was not the

most optimum method because VPR dumps slacks which are normalized based on the worst negative slack value in the design. So, to overcome this shortcoming, we shifted our concentration on timing criticalities of the nets. Timing criticality is the obvious choice as VPR's timing cost function [1] also uses timing criticality information to optimize timing of the placement. Timing criticality is calculated as per Eq. (4.1). We carried out several experiments in which we considered taking the average of timing criticality of all nets between two connected blocks and use this average value to update the connectivity matrix weights. Results of this average net weighting scheme were promising, however, they were not consistent for all the benchmark circuits. We then targeted to just concentrate on the worst case timing criticality between two connected blocks i.e. consider the maximum timing criticality among all nets between two blocks. This resulted in good consistent results. We tested it by taking different seed values to generate VPR reference placement so that we obtain a different I/O placement to start with. Based on this we run our analytical placer and obtain a different optimized placement each time.

Timing criticality values lie between 0 and 1, 0 being least critical and 1 being the most critical. So, in order to enhance the effect of net weights, we multiply the Max_Criticality by 100 so that the final value comes between 0 and 100. We use 100*Max_Criticality and multiply this with the total number of connections between the two CLBs to generate a new high value of net weight in the connection matrix in order to guide the placement engine ahead.

$$New\ Weight = Max_Criticality * 100 * No\ of\ Nets\ Connected \tag{3.8}$$

In this way, we have used timing criticality information to update the weights in connectivity matrix and drive the placement engine ahead. We have observed that, on an average, by employing the above net weighting scheme, the sum of criticality of all nets for the final legalized placement across benchmarks from the analytical engine when compared to VPR reference placement's sum of criticality of all nets, a reduction of 9-10% is seen. Detailed results are presented in chapter 4.

3.3.7 Low Temperature Simulated Annealing

Detailed placement is performed using low temperature simulated annealing [44] to get optimized critical path delay and wirelength. Since VPR [1], [2] has a state-of-the-art simulated annealing environment already developed, we choose to use VPR in order to carry out detailed placement in our flow. In this subsection, we shall outline the important parameters connected to the simulated annealing methodology used in VPR and then discuss about our implementation of low temperature simulated annealing and decision of the starting low temperature.

Algorithm 2 Simulated Annealing Based VPR Placer Algorithm

1: **procedure** SA_PLACER ▷ Simulated Annealing Procedure
2: $S \leftarrow RandomPlacement()$
3: $T \leftarrow InitialTemperature()$
4: $D_{limit} \leftarrow InitialD_{limit}$
5: **while** $ExitCriterion() == false$ **do**
6: **while** $InnerLoopCriterio() == false$ **do**
7: $S_new \leftarrow GenerateViaMove(S, D_{limit})$
8: $\Delta C \leftarrow Cost(S_{new}) - Cost(S)$
9: $r \leftarrow random(0, 1)$
10: **if** $r \leq e^{-\frac{\Delta C}{T}}$ **then**
11: $S \leftarrow S_{new}$
12: **end if**
13: **end while**
14: $T \leftarrow UpdateTemp()$
15: $D_{limit} \leftarrow UpdateD_{limit}()$
16: **end while**
17: **end procedure**

Algorithm 2 shows the algorithm implemented in VPR. It begins by placing the logic blocks (CLBs) and IOs randomly on the FPGA chip. This is the initial placement as done by VPR. Next, VPR performs random swaps and calculates the objective cost function. If the cost reduces by the swap, the move is accepted. If the cost increases compared to the earlier case, then decision to accept the swap depends on the probability of acceptance given by Eq (3.9) and a uniform random number (R) generated between 0 and 1. Initially, T is high enough so almost all moves are accepted. It is gradually decreased as the placement improves, in such a way that eventually the probability of

accepting a worsening move is very low. This hill climbing ability allows SA to not converge at a local minima and provide global optimization.

$$e^{-\frac{\Delta C}{T}} > R \ \ then accept, \ else \ reject \ swap \tag{3.9}$$

where, ΔC is the change in cost as described next under auto-normalizing cost function and T is a parameter called temperature that controls probability of accepting moves that worsen the placement.

The important parameters involved in this algorithm are listed below:

- **Wirelength Cost**

  The wirelength cost function is defined according to the Eq. (3.10) and is estimated using a semi-perimeter metric. The wirelength is an estimate of the routing resources needed to completely route all the nets in the design. It is the parameter that defines the quality of placement. This linear congestion cost function provides the best result in reasonable computation time.

$$Wirelength \ Cost = \sum_{n=1}^{N_{nets}} q(n)[\frac{BB_x(n)}{C_{av,x}(n)} + \frac{BB_y(n)}{C_{av,y}(n)}] \tag{3.10}$$

  where, summation is over all the nets in the circuit. BBx and BBy are the horizontal and vertical span of bounding box of each net. q(n) is the factor which helps in compensating the underestimation of wire length for net with more than three terminal as described in [1]. $C_{av,x}$(n) and $C_{av,y}$(n) are the average channel capacities (in tracks) in X and Y direction respectively, over bounding box of net n.

- **Timing Cost**

  Timing Cost function for a connection from i to j is defined as per Eq. (3.11). This portion of cost function is responsible for minimizing the critical path delay. Timing cost is based on criticality as given in Eq (4.1).

$$Timing_Cost(i,j) \ = \ Delay(i,j) * Criticality(i,j)^{Criticality_Exponent} \tag{3.11}$$

  where Timing_Cost is of a connection (i,j). The total Timing Cost for a circuit is

the sum of the Timing Cost of all of its connections [2]:

$$Timing_Cost \;=\; \sum_{\forall i,j \subset circuit} Timing_Cost(i,j) \tag{3.12}$$

- **Auto-Normalizing Cost Function ΔC**

  The auto-normalizing cost function ΔC is given by Eq. (3.13). It depends on the change in Timing_Cost and Wiring_Cost. It uses a tradeoff variable call λ to determine how much weight to give each component.

$$\Delta C = \lambda \frac{\Delta Timing_Cost}{Previous_Timing_Cost} + (1-\lambda)\frac{\Delta Wiring_Cost}{Previous_Wiring_Cost} \tag{3.13}$$

- **Initial Temperature**

  As SA starts with random placement, it targets to avoid local minima by hill climbing for which it needs good high initial temperature. VPR follows [43] to obtain the high starting temperature. After the initial random placement, VPR makes N moves (pairwise swaps) of CLBs and IOs, where N is the total number of blocks (CLB + IO) for that particular circuit. It sets the temperature to a very high value so that all these N swaps are accepted. For every swap, it calculates the ΔC cost of the move. Finally, once N moves are complete, the initial temperature is assigned as 20 times the standard deviation of final cost of these N moves.

- **Number of Moves and Temperature Update**

  Number of moves at each temperature is define as per Eq. (3.14), where, InnerNum is default at 10 and R_{accept} is the rate of acceptance of the move at each temperature.

$$Moves\ Per\ Temperature \;=\; InnerNum * (N_blocks)^{\frac{4}{3}} \tag{3.14}$$

  [3] proposed a new temperature update scheme as per Eq. (3.15), where, α depends on the value of R_{accept} as shown in the table 3.1. It ensures that at high temperature almost every move is accepted avoiding local minima and spends enough time at a temperature where significant fraction of, but not all, moves are being accepted.

$$T_{new} = \alpha * T_{old} \tag{3.15}$$

Table 3.1: Temperature Update Schedule [2]

Fraction of Moves Accepted (R_{accept})	α
$R_{accept} >0.96$	0.5
$0.8 < R_{accept} \leq 0.96$	0.9
$0.15 < R_{accept} \leq 0.8$	0.95
$R_{accept} <0.15$	0.8

- D_{limit}
  [45] and [46] suggests to keep R_{accept} near to 0.44 as long as possible, which can be achieved if the blocks are interchanged in the range of D_{limit}. Initially this limit is set to the FPGA dimension. It varies as per the Eq. (3.16) given below:

$$D_{limit}^{new} = D_{limit}^{old} * (1 - 0.44 + R_{accept}^{old}) \tag{3.16}$$

- **Exit Criteria**
  Finally annealing is terminated when $T \leq 0.005 * cost/N_{nets}$

Now we shall discuss the methodology in which we calculated the initial temperature for simulated annealing. This temperature should not be too high that it will destroy the placement generated from our MATLAB placer. Also it should not be too low, else the optimization will not be sufficient and placement might get trapped in local minima. Hence, we decided to calculate this value on the same lines as that of VPR. We decided to accept a move which causes a degradation of 5% in cost with a probability of acceptance equal to 10%. So, we have change in cost $\Delta C/C = 0.05$ and $R = 0.1$. Hence, based on Eq. (3.9), we get the temperature parameter as given by Eq. (3.17).

$$T = -\frac{\Delta C}{C} * \frac{C}{ln(R)} = -\frac{0.05}{1} * \frac{1}{ln(0.1)} = 0.0217 \tag{3.17}$$

C is equal to 1 in this case because it is the normalized cost and since no swaps are performed before this step as far as the placement obtained from MATLAB is concerned, it can be conveniently initialized to 1.

Since, we want VPR to only accept a subset of moves unlike the way explained above, we provide this small value of T to VPR. We also provide the placement generated by

our MATLAB placer to the starting_t function in VPR. This function undergoes N perturbations, where N is the number of blocks of the circuit under test and calculates the average cost of the accepted moves. Note that, since, we supplied a smaller value of T to starting_t function, we do not see all the moves taken being accepted. There are certain moves which do get rejected in this case. We finally use the average cost as C in the Eq. (3.17) and divide it by 120 to obtain the starting temperature for detailed placement step. Using the method described here, enables us to get a low initial temperature where acceptance rate approximately starts between 28-30%. Once initial temperature is calculated, we reset all the swaps taken by VPR in the process to get back our original placement obtained from MATLAB. Now, using this MATLAB placement and initial temperature calculated, we perform the detailed low temperature simulated annealing step.

Chapter 4

Experiment and Result Analysis

In this chapter, we shall first go over some of the essential inputs which the CAD flow requires as a part of experimental setup in section 4.1. This will be followed by a discussion and analysis of the results obtained by running the 20 homogeneous MCNC Benchmarks [9] on an architecture with 6-input LUTs and flip-flop packed into a CLB.

4.1 Discussion on Experimental Setup

In this section, we describe about the various important inputs provided to our CAD flow. In 4.1.1, a brief overview about the types of architectures available in VPR and the ones we are using to analyse the results (k6_N10_40nm architecture) is discussed. In 4.1.2, the benchmark circuits used to test our analytical placer are described. This is followed by a discussion on the format of the timing criticality file in 4.1.3, which is generated after each iteration of quadratic placement and how we make use of this data in our flow. Lastly, in 4.1.4, we point out about the location of the IO blocks used before the start of the analytical placement engine.

4.1.1 Architecture File

VTR 7.0 [6] comes with different flavours of FPGA architectures. The *Comprehensive Architecture* is the flagship architecture for VTR 7.0 release. It describes a number of modern features in FPGA like fracturable LUTs, carry-chains, fracturable multipliers and configurable memories. This is a kind of heterogeneous architecture for FPGAs and

Table 4.1: Major Architecture Files in VTR 7.0 Release [6]

File name	**Description**
k6_frac_N10_frac_chain_mem32K_40nm	Comprehensive Arch: ten fracturable 6-LUTs with carry chains, 32kb RAM and hard mutipliers
k6_frac_N10_frac_chain_depop50_mem32K_40nm	Comprehensive Arch with depopulated crossbar
k6_frac_N10_mem32K_40nm	Comprehensive Arch without carry chains
k6_frac_N10_40nm	Comprehensive Arch without carry chains or hard logic
k4_N4_90nm	Classical Arch: four 4-LUTs per logic cluster and no hard blocks
k6_N10_40nm	Classical Arch: ten 6-LUTs per logic cluster and no hard blocks
hard_fpu_arch_timing	Classical Arch with hardened floating point block

closely resembles the modern day FPGA. Also, it includes the *Classical Architecture* which describes a much simpler version of FPGAs having only LUTs, flip-flops and I/Os. Since, this is our first step into the FPGA placement work, we have targeted the classical architecture provided by VTR 7.0 in this thesis. Figure 4.1, illustrates the classical soft logic block.

It consists of N basic logic elements (BLEs), where each BLE is a LUT with an optionally registered output [2], [Betz et al. 1999]. This architecture comes in two variants, one having ten general inputs and four BLEs per cluster (N=4) and each of the LUTs has four inputs. All the routing wires are length 1, single-driver, with Fc_{in} = 0.15 and Fc_{out} = 0.25 and Fs = 3. There are three I/O pins per I/O block. The area and delay models come from a 90nm transistor-optimized architecture from the iFAR repository [Kuon and Rose 2008].

The second variant includes 40 general inputs and ten BLEs per cluster (N=10)

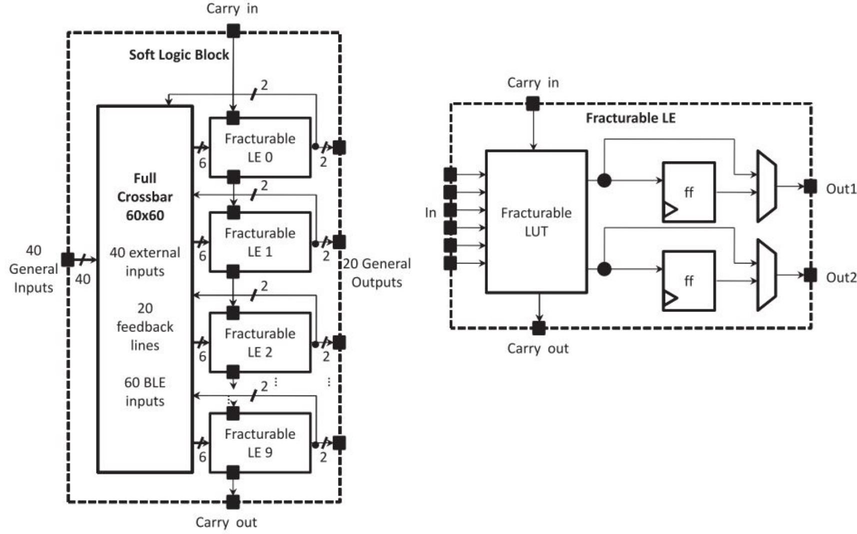

Figure 4.1: Classical Soft Logic Block [6]

and each of the LUTs has six inputs. All the routing wires are length 4, with Fc_{in} = 0.15 and Fc_{out} = 0.1 and Fs = 3. This architecture is based on the flagship k6_frac_N10_mem32k_40nm architecture without any fracturable LUTs nor any heterogeneous blocks. There are eight I/O pins per I/O block.

Table 4.1, gives a summary of the major architecture files available in VTR 7.0 release. These include an architecture that uses a depopulated crossbar with the logic cluster to save area and simplified architectures without fracturable LUTs, without carry chains and without hard logic. An architecture with hard floating point units is also included in this version of VTR release [6].

4.1.2 Benchmark Circuits

Microelectronics Center of North Carolina (MCNC) benchmark suite [9] is used as logic synthesis and optimization benchmark. The benchmark suite has standardized libraries with representative circuit designs ranging from simple circuits to advanced circuits. MCNC benchmarks are very popular in academic research. These MCNC benchmarks are available in the VTR 7.0 release and are in the *.blif* format (Berkeley Library Interchange Format). In our CAD flow, we use 20 of the largest MCNC benchmark circuits on the homogeneous architectures discussed in 4.1.1. We also experiment with

Table 4.2: Statistics of Benchmark Circuits for k6_N10_40nm Architecture

Circuit	# Blocks	# IOs	# CLBs	# Nets	Chip Area
alu4	175	22	153	697	13 * 13
apex2	229	41	188	969	14 * 14
apex4	155	28	127	699	12 * 12
bigkey	596	426	170	1024	14 * 14
blob_merge	739	136	603	3113	25 * 25
clma	982	144	838	4815	29 * 29
des	661	501	160	997	13 * 13
diffeq	253	103	150	943	13 * 13
dsip	563	426	137	691	12 * 12
elliptic	606	245	361	1907	19 * 19
ex1010	480	20	460	2572	22 * 22
ex5p	179	71	108	669	11 * 11
frisc	492	136	356	1748	19 * 19
misex3	168	28	140	716	12 * 12
pdc	514	56	458	2292	22 * 22
s298	204	10	194	722	14 * 14
s38417	771	135	636	3567	26 * 26
s38584.1	977	342	635	3641	26 * 26
seq	251	76	175	879	14 * 14
sha	303	74	229	1322	16 * 16
spla	431	62	369	1808	20 * 20
stereovision3	61	41	20	125	5 * 5
tseng	279	174	105	588	11 * 11

three benchmarks - blob_merge, sha and stereovision3 available with the VTR 7.0 release in our flow.

Table 4.2, gives the statistics of these benchmark circuits based on k6_N10_40nm architecture. The first column lists the name of the benchmark circuits. The second column provides the data about the total number of blocks i.e. combined count of I/O Blocks and CLBs present in the circuit. The third and fourth columns give the details about the number of I/O Blocks and number of CLBs respectively in the benchmark circuit. The fifth column list the total number of nets connecting each CLB-CLB or CLB-IO pairs in the benchmark circuit. The sixth column gives the information about

the chip area i.e. the total number of available locations on the FPGA grid where the CLBs can be placed.

4.1.3 Timing Criticality File

Timing driven placement calls for the need to do timing analysis and generate the timing criticalities as the placement progresses. In VPR, timing analysis generates two important files:

- Slack File
- Timing Criticality File

The way in which VPR generates this file is that it loops through all the nets in the design (all nets connecting CLB-CLB or CLB-IO pairs) and finds the driver of that particular net. Based on the VPR's internal data structures, it finds out the timing graph node corresponding to the driver pin which is driving this particular net. Next, it finds the sink nodes connected to the net. Based on this connectivity information obtained, VPR dumps the slack and timing criticality information in the file. VPR calculates the timing criticality based on the following formula:

$$Timing\ Criticality = 1 + \frac{Slack}{D_{max}} \tag{4.1}$$

where D_{max} is the delay of the longest path in the design and Slack is the normalized value of slack calculated by VPR. The worst negative slack value is added to the slacks of all the nets so that all the slack values are non-negative. This is the way VPR calculates the normalized slack values.

We use the timing criticality file in each iteration to update the weights in the connectivity matrix. The timing criticality file contains the driver CLB pin node index and the corresponding sink pin node indices. Each driver-sink pair has a criticality value associated with it. Since, the timing criticality generated is per net basis, and our connectivity matrix considers all the nets going from one CLB to another as one single net, we use the maximum criticality value between two CLBs to model the worst case. This maximum value of timing criticality is used to update the weights in the connection matrix as discussed in 3.3.6.

4.1.4 VPR Reference Placement IO Locations

The way in which our analytical placement flow works is, first VPR runs and generates a reference placement as discussed in 3.1. Next, the MATLAB engine is called from within VPR and our timing driven quadratic placer runs and generates a placement of CLBs on the FPGA physical grid locations. Lastly, the placement generated from MATLAB is refined further using low temperature simulated annealing in VPR. An analytical placer will concentrate on finding the optimized placement locations for the movable blocks i.e. the CLBs. We also need to take care of the I/O Blocks. There are two ways to decide on the placement of the I/O blocks:

- Place the I/O blocks randomly around the periphery of the FPGA chip
- Use the I/O locations from the reference placement generated by VPR and fix them

Random placement of I/O locations in MATLAB flow will cause mismatch between the VPR reference placement and our final placement. This will make it difficult for us to make apples-to-apples comparison between the two. However, using the I/O locations obtained from the reference placement of VPR and fixing them at those positions till the end of the entire timing driven CAD flow will make it easier to compare and obtain accurate results between the two placement engines. In order to test the quality of our analytical placement engine, we run VPR multiple times with different seed values, so that everytime a different reference placement is obtained. This implies at every seed, a different set of I/O locations is obtained when VPR finishes its reference placement. We use this I/O placement and drive our analytical placer to get the final placement and calculate the critical path delay and wirelength based on it.

4.2 Results and Analysis

In this section, we describe the results obtained after comparison between our placer and VPR. All the benchmarks are run on an unloaded Intel Core 2 Duo CPU running at 3.00 GHz. The MATLAB version used is 64-bit R2013a (8.1.0.604). In order to account for the CAD tool noise, we run the same benchmarks with ten different seed values

and take the average of all the ten runs to obtain the final results. The comparison is classified into two categories:

- Post Placement Results
  - Sum of Timing Criticality across all nets
  - Critical Path Delay
  - Total Estimated Wirelength (HPWL)
  - Placer Runtime
- Post Route Results
  - Critical Path Delay
  - Routed Wirelength
  - Best Routing Channel Factor

For all the runs, VPR reference placement is done with default timing tradeoff of 0.5 and for low temperature simulated annealing, we set the timing_tradeoff factor to 0.75 to model the VPR placer as a timing driven placer during detailed placement. Also, for low temperature simulated annealing detailed placement step, we use the start temperature as described in 3.3.7 and a D_{limit} of nx/2 and ny/2 to start with, where nx and ny are the X and Y dimension of the FPGA chip. In subsection 4.2.1, the analysis for post placement results is presented and in subsection 4.2.2, post route results analysis is discussed.

4.2.1 Post Placement Results

Given below are the post placement step results. For better analysis, we have presented the results in a tabular fashion and a corresponding graph for the same. The table columns are in the following order: The first column shows the benchmark circuit names, the second, third and fourth columns show the values obtained after placement from VPR, MATLAB and low temperature simulated annealing step (Cool SA) respectively. The fifth and sixth column provides the ratio between Cool SA to VPR and MATLAB to VPR values respectively. In the fifth and sixth column, a value less than one implies improvement in the results, and greater than one implies otherwise.

- **Sum of Timing Criticality Across Nets**

  As discussed in 4.1.3, VPR calculates the timing criticality per net basis. To view the global picture of the timing criticality in the design, we sum up the criticality of all the nets for each benchmark and compare the same with baseline VPR reference placement. The results show, MATLAB as a global placer achieves on an average a 10% reduction in the sum of timing criticality of all nets across all benchmarks compared to VPR reference placement. Although, the critical path delay from MATLAB placement is slightly higher than VPR reference placement, MATLAB however, does a very good job in global placement, reducing the total timing criticality to a very huge extent (maximum reduction observed is 21% in case of s38584.1 benchmark). This high critical path delay can be reduced by using any detailed placement methods like the one used by us in the thesis - low temperature simulated annealing. Table 4.3, shows the comparison of sum of timing criticality between VPR placement, MATLAB placement and placement after low temperature simulated annealing. Figure 4.2 illustrates the data presented in table in a graphical way.

- **Critical Path Delay (CPD)**

  Critical path delay is the delay of the path having the least negative slack in the design. Table 4.4, shows the CPD information from VPR, MATLAB and low temperature simulated annealing placement. Results show that on an average, the CPD obtained after low temperature simulated annealing detailed placement is almost comparable to VPR's reference placement CPD. The CPD from global placement is 30% degraded compared to VPR, hence the need to do detailed placement step. This degradation is due to the fact that MATLAB considers quadratic objective functions and optimizes it. However, we know that improvement in quadratic objective function does not directly improvise the linear objective, hence the higher critical path delay. Also, AP does not consider the order in which CLBs are connected. If CLB A, B, C are connected in order, then they might be less critical. If the order changes to A, C, B then the criticality might increase and AP does not consider this effect while doing placement. The maximum improvement in CPD after low temperature annealing is seen for *seq* benchmark circuit - a 3% improvement over VPR CPD. Figure 4.3 shows the data presented in table 4.4 in

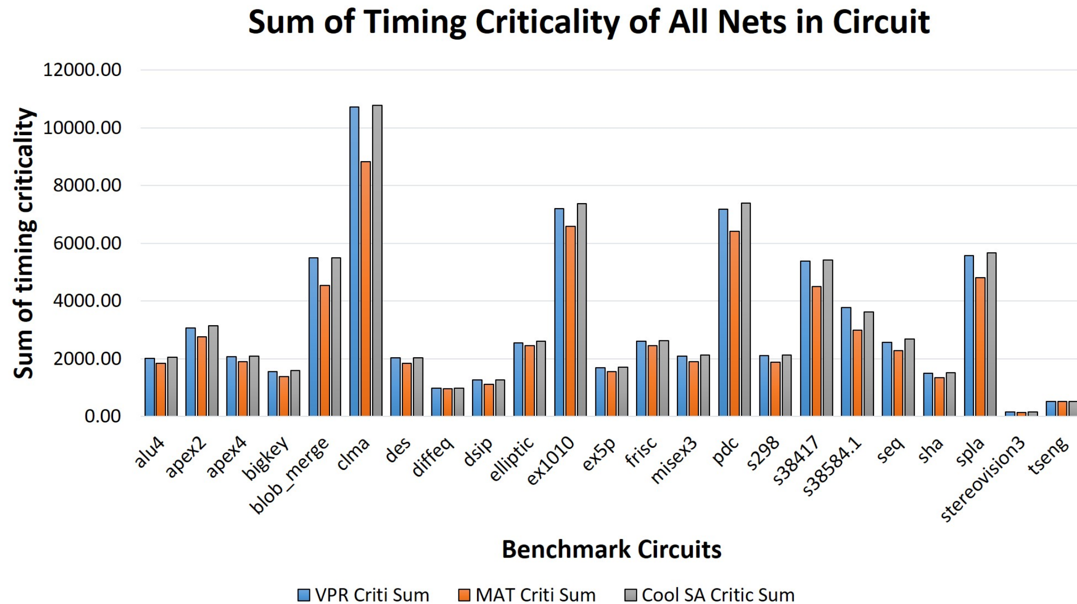

Figure 4.2: Graph showing comparison of sum of timing criticality of all nets per benchmark

a graphical format.

- **Total Estimated Wirelength (HPWL)**
  Total wirelength calculation is based on building bounding boxes and calculating the half-perimeter wirelength as done in VPR. Table 4.5, shows the comparison between VPR, MATLAB and cool simulated annealing wirelength after legal placement. It is observed that the wirelength after low temperature simulated annealing step is 9% degraded than the VPR reference placement. Figure 4.4, shows the graph for the wirelength comparison for the three different placements.

- **Placer Runtime**
  The runtime for VPR reference placement is calculated directly from the time VPR begins the try_place function till the end of the simulated annealing freeze out condition. We calculate the total time taken using the time() function in C. So basically, we are calculating the wall clock time taken for the placement step to complete. Consideration of wall clock time in C allows us to directly add the

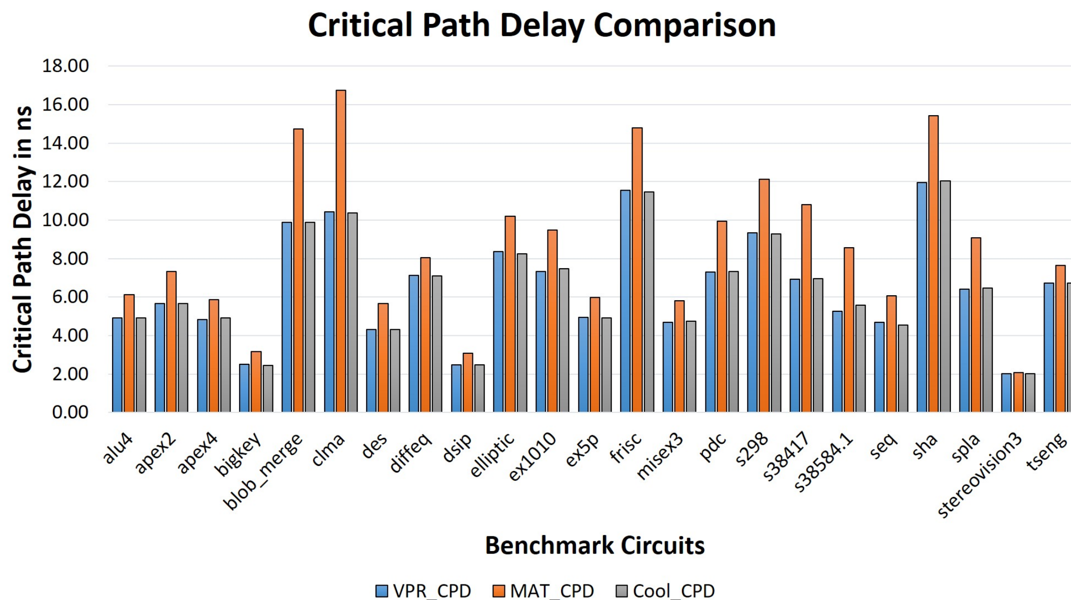

Figure 4.3: Graph showing comparison for Critical Path Delay (CPD)

MATLAB runtime calculated using the tic and toc functions to time_t value in C. Since, MATLAB cputime function and the clock() function in C are not on the same baseline i.e. it gives apples to oranges comparison, we decided to go ahead with wall clock time for runtime comparison of both placers.

The calculation for runtime of the analytical placer (MATLAB + Low Temperature Annealing) is done in various stages. As MATLAB framework is modularized into different functions to achieve the required work, the runtime calculation is divided into 4 different stages for MATLAB. These are as follows:

1. Time taken to read the timing criticality file and update the connection matrix (Conn MAT RT)
2. Time taken by quadprog to generate a placement in every iteration i.e. generate the X and Y coordinates for every iteration (Placer RT)
3. Time taken by spread function (floor and ceiling) to generate intermediate placement required for timing analysis after every quadprog placement iteration, except the last (Spread RT)

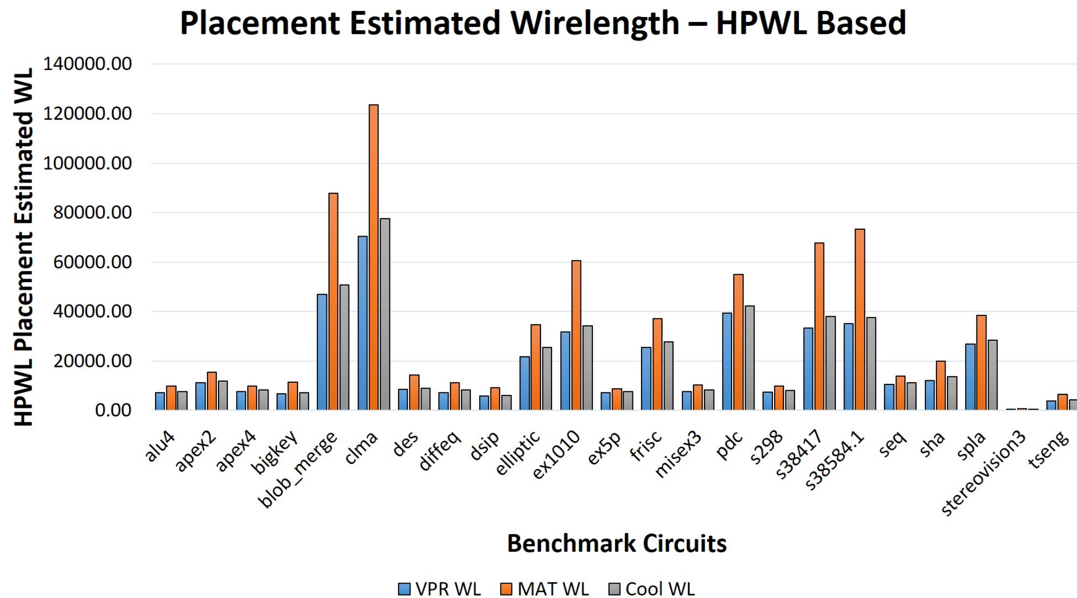

Figure 4.4: Graph showing comparison for estimated wirelength after placement (HPWL)

4. Time taken to legalize the final iteration placement (Legal RT)

In every iteration, we take the sum of stages (2), (3) and (4) shown above and calculate the MATLAB placer runtime. Note that (4) will be a non-zero value only for the final iteration of placement and (3) in that case will be zero as per our algorithm. The reason for doing this kind of timing calculation and not considering stage (1) in the total runtime is because of the fact that our placer is not written in C and hence stage (1) above has to read two files to get the timing criticality information and the connection matrix row and column information respectively. Once read, it has to update the connectivity matrix rows and columns based on the timing criticality and again save it in a file for the next iteration. We believe that if the placer was written in C, this bottleneck in runtime would have been much less than the current time taken. All this data would have been stored in the C data structures and it would have resulted in tremendous reduction in runtime not only for stage (1) alone but also other stages too.

Every MATLAB iteration is followed by a timing analysis step. This is done in VPR as explained in subsection 3.3.5. We calculate the time taken to do timing analysis from the point where we receive the intermediate placement from MATLAB to the point timing analysis is completed. We add this timing analysis runtime to Stage (2) described above. Once, MATLAB based global placement is over, low temperature simulated annealing is done for detailed placement. The time for doing this is the stage (5) time which is calculated from the moment MATLAB global placement is available till the end of the simulated annealing freeze out step.

Figure 4.5 illustrates the relationship between the runtime of MATLAB global placement, VPR placement and the size of the circuit i.e. total number of blocks (CLBs + I/Os) of a benchmark circuit. It can be inferred from the graph that on an average MATLAB global placer runtime outperforms VPR simulated annealing based placement runtime by 30%.

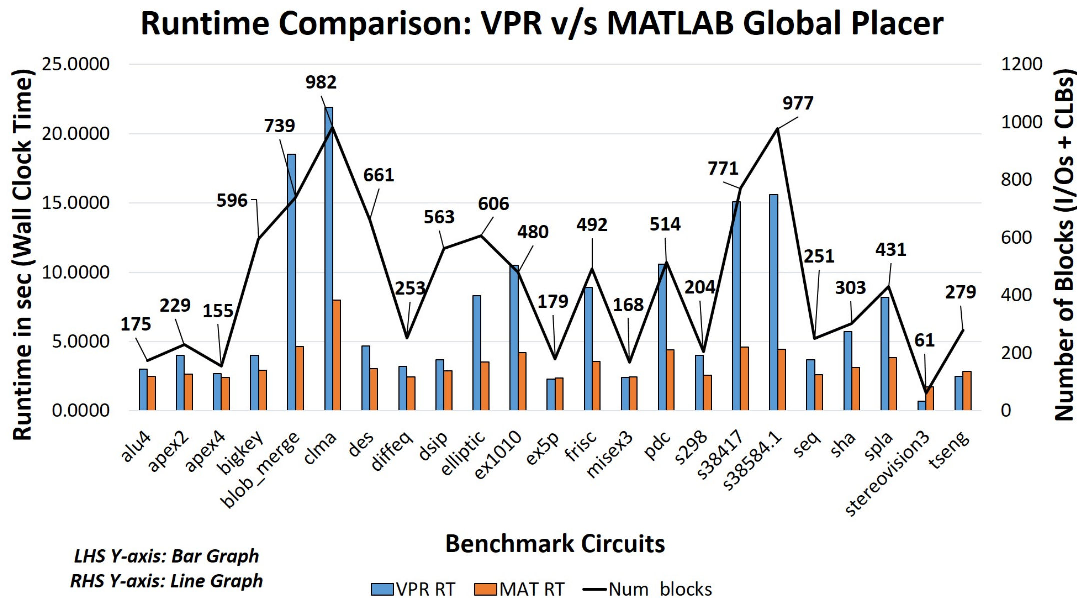

Figure 4.5: Graph showing relationship between VPR and MATLAB Runtime

Figure 4.6, shows a pie-chart with the break-up of the our analytical placer's

(MATLAB + Cool SA) runtime divided in different stages for one of the largest benchmarks - **blob_merge**. It can be observed that time taken for low temperature simulated annealing detailed placement i.e. stage (5) has 66% share in the total time consumption. Also, the MATLAB placement step (i.e. Stage (2) described above) occupies 31% of the total runtime for MATLAB and Cool SA combined. Note that timing analysis hardly takes any significant portion of runtime to generate the slack and timing criticalities, hence, as far as stage (2) time is concerned, it is safe to ignore the timing analysis runtime from stage (2). Other factors like the spreading function and legalization stage take almost negligible time. It can be inferred from this pie-chart that further refining the algorithm of global placement will help in reducing the overall runtime.

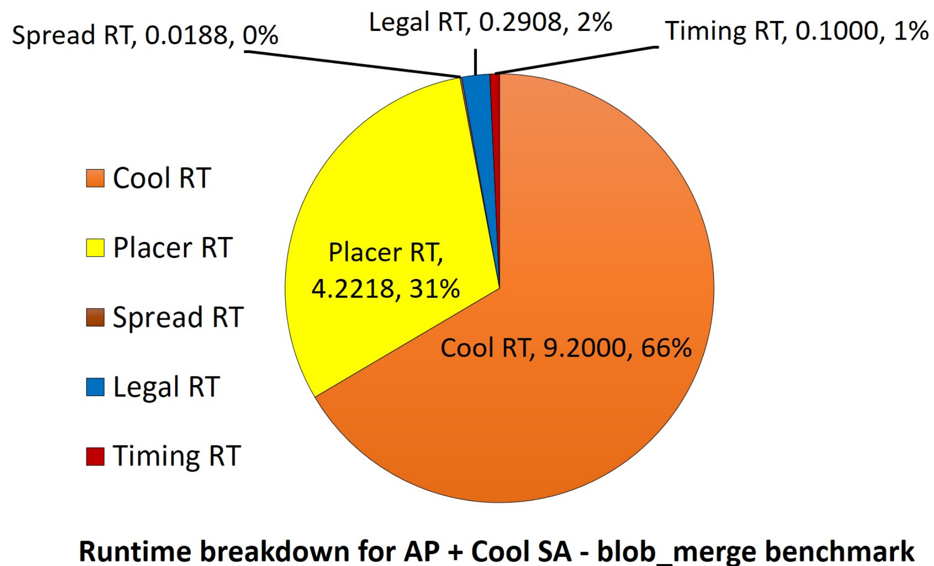

Figure 4.6: Pie Chart showing the break-up of MATLAB and Cool SA Placement Runtimes

Table 4.6 gives detailed numbers for runtime (RT) taken by VPR, MATLAB and Cool Simulated Annealing stages for all benchmark circuits. The last two columns show the ratio of sum of global and detailed placement (AP) to VPR reference placement and global placement to VPR reference placement runtime for each

benchmark respectively. The maximum improvement in global placer runtime of 75% is seen for blob_merge benchmark circuit.

Table 4.7 gives a breakdown of the different stages of MATLAB based global placement engine for reference. The first column denotes the circuit names. The remaining columns contain the time taken for Stages (1), (2), (3) and (4) respectively.

4.2.2 Post Route Results

Given below are the post route results. We ran the entire the VPR flow from packing till the end of routing stage for 10 seed values with a timing tradeoff factor of 0.75. The post route results obtained from this run are termed as golden VPR run results. The Analytical Placement - AP (i.e. MATLAB global placement followed by low temperature simulated annealing detailed placement) post route results are termed as AP run results. For better analysis, we have presented these results in a tabular fashion and a corresponding graph for the same. The table columns are in the following order: The first column shows the benchmark circuit names, the second, third columns show values obtained from golden VPR routing stage and Analytical Placement (AP) stage respectively. The fourth column contains the ratio of AP to golden VPR numbers. A value less than one in the fourth column, implies improvement in the results, and greater than one implies otherwise. Similar stands true for the fifth to seventh columns as well.

Note that for seed value of 9, golden VPR run for **sha** benchmark failed the routing step, whereas it successfully routed the circuit with our AP flow for the same settings in VPR. Also, for seed 2, stereovision3 did not get routed in our flow. Hence, we have provided post route results for all benchmark except sha and stereovision3.

- **Post Route Critical Path Delay (R-CPD)**
  Given below is the comparison between the post route critical path delay from VPR's reference placement and our Analytical Placer (AP) placement. It is observed that average of AP routed CPD across all benchmarks and normalized to VPR post route CPD shows 2% degradation compared to that of VPR's reference placement. Table 4.8 shows the critical path delay for golden VPR placement and AP placement. It can be seen that for the benchmark circuit *elliptic*, 3%

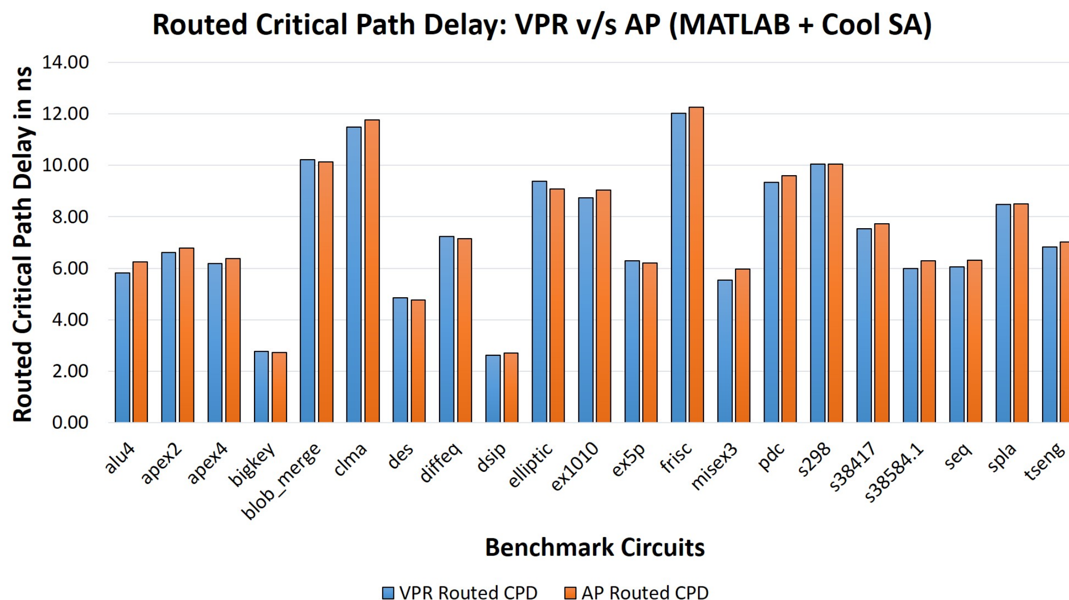

Figure 4.7: Graph showing comparison post route critical path delay (R_CPD))

improvement in the routed critical path delay is obtained, whereas, for *alu4*, 7% degradation in post route critical path delay is observed. Figure 4.7, depicts a graph showing this comparison across all benchmarks.

- **Post Route Wirelength (R-WL)**
  Here, we compare the post route wirelength obtained from golden VPR flow and AP flow. Table 4.8 shows the comparison between the wirelength values in the two runs. The circuit sha was non-routable for seed 9 in the golden VPR flow, however, the same was successfully routed in our AP flow. We show the results taking all benchmarks except for sha in this case. Figure 4.8 illustrates this data with the help of graphical representation. The data shows an average 3% degradation in AP post route wirelength compared to golden VPR post route results.

- **Routing Channel Factor (CF)**
  The routing channel factor is defined as the maximum tracks per channel used by the router to successfully route the design on the FPGA fabric. We compared the golden VPR's routed results with our AP run results. It can be observed that

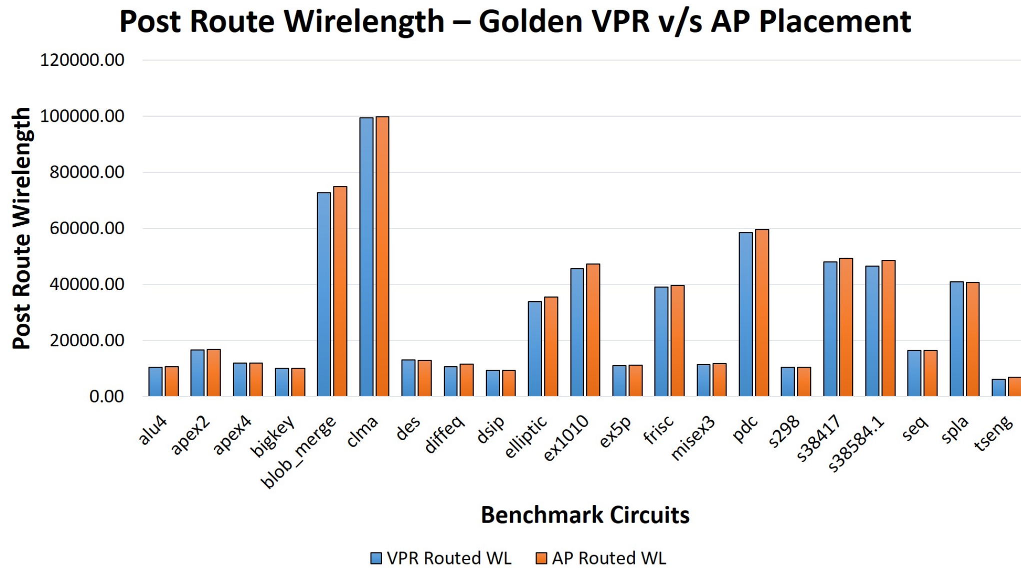

Figure 4.8: Graph showing comparison post route wirelength (R_WL))

there is 3% increase in the channel factor in AP's routed result compared to VPR's golden results. The average of the channel factor across all benchmarks for run with 10 seed different values is shown in Table 4.9. The first column represents the benchmark circuit names. The second and third column provide the average value of channel factor from the 10 runs of golden VPR and AP routing results respectively.

Table 4.3: Comparison: Sum of timing criticality across all nets after placement for VPR, MATLAB and after low temperature SA

Ckt Name	**VPR**	**MATLAB**	**Cool SA**	**Cool SA/VPR**	**MAT/VPR**
alu4	2023.67	1850.54	2047.68	1.01	0.91
apex2	3078.92	2762.12	3144.15	1.02	0.90
apex4	2071.56	1896.13	2086.35	1.01	0.92
bigkey	1567.29	1392.91	1592.63	1.02	0.89
blob_merge	5500.21	4548.35	5505.29	1.00	0.83
clma	10727.12	8829.57	10782.57	1.01	0.82
des	2044.38	1841.70	2042.13	1.00	0.90
diffeq	979.62	974.75	989.56	1.01	1.00
dsip	1262.56	1112.36	1262.27	1.00	0.88
elliptic	2560.18	2463.02	2609.83	1.02	0.96
ex1010	7212.33	6596.42	7374.09	1.02	0.91
ex5p	1691.60	1559.23	1715.60	1.01	0.92
frisc	2609.90	2455.27	2624.87	1.01	0.94
misex3	2088.52	1911.54	2127.83	1.02	0.92
pdc	7179.37	6418.88	7395.78	1.03	0.89
s298	2109.11	1883.90	2124.87	1.01	0.89
s38417	5393.71	4507.38	5418.42	1.00	0.84
s38584.1	3775.74	2996.50	3628.78	0.96	0.79
seq	2572.77	2291.61	2682.32	1.04	0.89
sha	1502.11	1347.02	1512.73	1.01	0.90
spla	5574.88	4812.20	5674.06	1.02	0.86
stereovision3	152.64	151.21	152.98	1.00	0.99
tseng	529.95	520.32	533.73	1.01	0.98
Overall Average				1.01	0.90

Table 4.4: Comparison: Critical Path Delay (CPD) after placement for VPR, MATLAB and after low temperature SA

Ckt Name	**VPR CPD**	**MATLAB CPD**	**Cool SA CPD**	**Cool SA/VPR CPD**	**MAT/VPR CPD**
alu4	4.92	6.12	4.91	1.00	1.25
apex2	5.67	7.32	5.66	1.00	1.29
apex4	4.85	5.88	4.93	1.02	1.21
bigkey	2.51	3.17	2.46	0.98	1.26
blob_merge	9.90	14.75	9.90	1.00	1.49
clma	10.42	16.76	10.38	1.00	1.61
des	4.31	5.67	4.33	1.01	1.32
diffeq	7.12	8.04	7.11	1.00	1.13
dsip	2.48	3.09	2.48	1.00	1.25
elliptic	8.37	10.19	8.26	0.99	1.22
ex1010	7.35	9.49	7.46	1.02	1.29
ex5p	4.95	5.98	4.93	1.00	1.21
frisc	11.56	14.79	11.48	0.99	1.28
misex3	4.69	5.80	4.76	1.02	1.24
pdc	7.32	9.95	7.33	1.00	1.36
s298	9.34	12.13	9.29	1.00	1.30
s38417	6.94	10.81	6.97	1.00	1.56
s38584.1	5.26	8.58	5.58	1.06	1.63
seq	4.69	6.07	4.56	0.97	1.30
sha	11.96	15.42	12.04	1.01	1.29
spla	6.41	9.10	6.48	1.01	1.42
stereovision3	2.03	2.08	2.03	1.00	1.03
tseng	6.73	7.64	6.73	1.00	1.14
Overall Average				1.00	1.31

Table 4.5: Comparison: Estimated Wirelength after placement for VPR, MATLAB and after low temperature SA

Ckt Name	**VPR WL**	**MATLAB WL**	**Cool SA WL**	**Cool SA/VPR WL**	**MAT/VPR WL**
alu4	7309.54	9813.85	7794.09	1.07	1.34
apex2	11362.37	15563.28	11989.22	1.06	1.37
apex4	7750.66	9997.65	8349.35	1.08	1.29
bigkey	6776.50	11484.95	7252.34	1.07	1.69
blob_merge	46980.02	87888.00	50798.58	1.08	1.87
clma	70331.59	123682.60	77687.62	1.10	1.76
des	8614.38	14378.32	8919.71	1.04	1.67
diffeq	7291.86	11291.06	8293.28	1.14	1.55
dsip	5845.72	9174.58	6168.24	1.06	1.57
elliptic	21764.80	34607.36	25661.99	1.18	1.59
ex1010	31907.66	60571.60	34248.11	1.07	1.90
ex5p	7176.37	8706.52	7708.51	1.07	1.21
frisc	25481.06	37249.00	27874.93	1.09	1.46
misex3	7759.50	10377.26	8369.06	1.08	1.34
pdc	39331.05	55038.21	42217.58	1.07	1.40
s298	7427.17	9950.88	8089.63	1.09	1.34
s38417	33393.49	67763.68	38009.09	1.14	2.03
s38584.1	35105.41	73277.13	37581.14	1.07	2.09
seq	10557.81	13958.03	11322.89	1.07	1.32
sha	12173.20	19997.38	13608.82	1.12	1.64
spla	26802.82	38555.95	28475.06	1.06	1.44
stereovision3	500.53	666.76	550.87	1.10	1.33
tseng	3978.25	6529.97	4439.82	1.12	1.64
Overall Average				1.09	1.56

Table 4.6: Comparison: Placement runtime for VPR, MATLAB and low temperature SA

Ckt Name	**# Blocks**	**VPR RT**	**MATLAB RT**	**Cool SA RT**	**AP/VPR RT**	**MAT/VPR RT**
alu4	175	3.0000	2.4709	1.0000	1.1570	0.8236
apex2	229	4.0000	2.6354	1.1000	0.9339	0.6589
apex4	155	2.7000	2.4234	1.0000	1.2679	0.8975
bigkey	596	4.0000	2.9226	1.2000	1.0306	0.7306
blob_merge	739	18.5000	4.6314	9.2000	0.7476	0.2503
clma	982	21.9000	7.9767	10.3000	0.8346	0.3642
des	661	4.7000	3.0434	1.8000	1.0305	0.6475
diffeq	253	3.2000	2.4556	1.1000	1.1111	0.7674
dsip	563	3.7000	2.8782	1.1000	1.0752	0.7779
elliptic	606	8.3000	3.5231	3.2000	0.8100	0.4245
ex1010	480	10.5000	4.2238	4.0000	0.7832	0.4023
ex5p	179	2.3000	2.3847	0.8000	1.3846	1.0368
frisc	492	8.9000	3.5809	3.5000	0.7956	0.4023
misex3	168	2.4000	2.4595	0.8000	1.3581	1.0248
pdc	514	10.6000	4.3928	4.8000	0.8672	0.4144
s298	204	4.0000	2.5639	1.4000	0.9910	0.6410
s38417	771	15.1000	4.5941	6.0000	0.7016	0.3042
s38584.1	977	15.6000	4.4541	7.0000	0.7342	0.2855
seq	251	3.7000	2.6058	1.4000	1.0826	0.7043
sha	303	5.7000	3.1216	2.5000	0.9863	0.5477
spla	431	8.2000	3.8387	3.3000	0.8706	0.4681
stereovision3	61	0.7000	1.7277	0.2000	2.7538	2.4681
tseng	279	2.5000	2.8444	0.7000	1.4178	1.1378
Overall Average					1.0750	0.7035

Table 4.7: MATLAB Global placement runtime break-up for each stage

Ckt Name	**Conn Mat RT**	**Placer RT**	**Spread RT**	**Legalize RT**	**Timing Anls RT**
alu4	4.05	2.29	0.01	0.17	0.00
apex2	5.75	2.33	0.01	0.19	0.10
apex4	3.86	2.26	0.01	0.16	0.00
bigkey	7.40	2.75	0.01	0.16	0.00
blob_merge	28.24	4.22	0.02	0.29	0.10
clma	35.03	7.28	0.02	0.47	0.20
des	9.08	2.75	0.01	0.18	0.10
diffeq	3.62	2.26	0.01	0.18	0.00
dsip	6.94	2.72	0.01	0.15	0.00
elliptic	13.09	3.26	0.02	0.24	0.00
ex1010	17.26	3.74	0.02	0.27	0.20
ex5p	3.33	2.22	0.01	0.16	0.00
frisc	12.92	3.24	0.02	0.22	0.10
misex3	4.09	2.27	0.01	0.18	0.00
pdc	18.21	4.12	0.02	0.26	0.00
s298	4.42	2.35	0.01	0.20	0.00
s38417	18.40	4.25	0.02	0.33	0.00
s38584.1	21.51	4.11	0.02	0.32	0.00
seq	5.21	2.32	0.01	0.18	0.10
sha	7.83	2.86	0.01	0.14	0.10
spla	13.26	3.51	0.02	0.21	0.10
stereovision3	0.71	1.58	0.00	0.14	0.00
tseng	3.92	2.69	0.01	0.14	0.00

Table 4.8: Comparison: Post Route Critical Path Delay (R_CPD) and Wirelength (R_WL) for Golden VPR and AP Placement

Ckt Name	**Golden VPR R_CPD**	**AP R_CPD**	**AP/VPR R_CPD**	**Golden VPR R_WL**	**AP R_WL**	**AP/VPR R_WL**
alu4	5.82	6.26	1.07	10344.40	10659.50	1.03
apex2	6.61	6.78	1.03	16574.40	16747.20	1.01
apex4	6.19	6.38	1.03	11924.90	11984.10	1.00
bigkey	2.77	2.73	0.98	9983.00	10056.10	1.01
blob_merge	10.23	10.14	0.99	72792.90	75017.00	1.03
clma	11.49	11.77	1.03	99443.60	99854.10	1.00
des	4.84	4.77	0.99	13026.50	12888.20	0.99
diffeq	7.23	7.15	0.99	10640.70	11540.60	1.08
dsip	2.61	2.70	1.03	9289.80	9373.70	1.01
elliptic	9.39	9.08	0.97	33801.40	35458.90	1.05
ex1010	8.73	9.04	1.04	45568.80	47256.40	1.04
ex5p	6.30	6.21	0.99	11045.00	11189.90	1.01
frisc	12.04	12.26	1.02	39082.10	39708.30	1.02
misex3	5.54	5.97	1.08	11317.90	11654.10	1.03
pdc	9.35	9.59	1.03	58599.20	59724.10	1.02
s298	10.05	10.05	1.00	10360.80	10385.70	1.00
s38417	7.54	7.73	1.02	48116.90	49318.40	1.02
s38584.1	5.98	6.30	1.05	46597.30	48625.00	1.04
seq	6.06	6.32	1.04	16334.40	16405.40	1.00
spla	8.48	8.50	1.00	40881.40	40810.30	1.00
tseng	6.82	7.02	1.03	6044.90	6800.70	1.13
Overall Average	7.34	7.46	1.02	29608.11	30259.89	1.03

Table 4.9: Comparison: Post Route Channel Factor for Golden VPR and AP Placement

Ckt Name	**Golden VPR Chan Fact**	**AP Chan Fact**	**AP/VPR Chan Fact**
alu4	38.60	37.00	0.96
apex2	48.60	48.80	1.00
apex4	49.60	50.00	1.01
bigkey	46.20	43.40	0.94
blob_merge	72.20	74.20	1.03
clma	74.20	79.40	1.07
des	39.00	41.00	1.05
diffeq	37.80	41.00	1.08
dsip	42.00	38.60	0.92
elliptic	55.60	58.60	1.05
ex1010	60.40	61.00	1.01
ex5p	52.80	54.20	1.03
frisc	64.80	67.60	1.04
misex3	45.80	46.40	1.01
pdc	76.00	75.60	0.99
s298	33.00	40.40	1.22
s38417	46.40	50.40	1.09
s38584.1	48.80	49.20	1.01
seq	47.20	48.20	1.02
spla	63.20	64.00	1.01
tseng	34.00	34.60	1.02
Overall Average	51.25	52.55	1.03

Chapter 5

Conclusion and Future Ideas

5.1 Conclusion

In this thesis, we presented a fast and efficient timing driven analytical placement flow for FPGAs. It does the work of fast global placement in MATLAB using Gordian-like methodology [29] and undergoes detailed placement by low temperature annealing in VPR to achieve optimized critical path delays and wirelengths. We formulated the quadratic equation from the connectivity matrix of CLBs and I/Os and solved it iteratively by providing linear constraints and boundary conditions in a way similar to Gordian technique [29], to come to a global placement solution. In order to achieve faster results, we partitioned the placement area into 2^{2I} sub-partitions, where I denotes the current iteration number. Also, we performed alternate iterations of updating the connectivity matrix, thereby making it sparse in odd iterations and dense in even iterations in order to improve the runtime of the global placer. Timing analysis was performed at the end of every iteration to calculate the timing criticality value and guide the placer ahead.

An important task undertaken was formulating a legal output from the MATLAB global placer such that there are no overlaps and the timing or wirelength does not degrade to a large extent. One of the main problems with quadratic placement methods is that they do not consider the overlap constraints during the equation solving process. Hence, the solution results in overlap of CLBs, giving a non-legalized placement. We overcame this problem using a spiral legalization technique where we search for locations

with maximum overlap and based on the timing criticality of all CLBs at that location, pick up the least timing critical CLBs and legalize them by travelling in a spiral fashion. Although, the delay on the critical path based on this legalized global placement came out to be slightly higher than that of VPR, we achieved a remarkable reduction in the overall sum of timing criticality for all nets in the circuit across all benchmarks. This higher critical path delay and wirelength was optimized by performing detailed placement using low temperature simulated annealing process in VPR.

We conclude by saying that we were successful in achieving our goal of making the global placement faster by almost 30% compared to VPR's simulated annealing placement and also obtained a legalized placement of CLBs without any overlaps on homogeneous FPGA fabric.

5.2 Future Ideas

This is our first attempt in developing a timing driven FPGA placement framework using quadratic programming approach. There are many ideas which can be implemented to enhance the quality of our placer. We believe that we can get further improvement in speed and QoR by pursuing these ideas.

First and foremost, if the same code is written completely in C beside the VPR framework, the runtime would reduce more. This is because, in C, all the functions will readily have access to internal data structures and reading the information from the data structures is always faster than reading from a file as we do in MATLAB at present. Moreover, we can have the liberty to check the connectivity of the entire circuit at any instant we like which is not true for MATLAB, which only has the visibility of the connectivity matrix.

Second thing would be to understand and use an appropriate net model, either a Star Model or Clique Model or formulate a completely new model to suit the FPGA needs. Also, an attempt should be made to replace the quadprog solver with any other quadratic solver optimized for dense circuits which can work with the net models and provide quick solutions. This modification shall help reduce the present cpu time taken for solving the connectivity matrix with the quadprog function. Higher the number of non-zero elements in the connectivity matrix, faster is the time to arrive at the solution

and get results. This is can be made possible using the same equation solver as used in [32]. Along with this, we can come up with a more efficient net weighting scheme which can model the inter-block delay based on the linear wirelength within the quadratic wirelength base.

In this work, during each iteration, the FPGA is partitioned and the CLBs are distributed uniformly in all partitions based on only their sorted values of X and Y coordinates. By doing so, few CLBs at the boundary which can be highly critical, end up in completely opposite partitions as we do not consider any timing criticality information while partitioning and distributing them. The improvement that can be done here is in the first iteration of placement, distribute the CLBs based on the current scheme. Once distributed, check for the timing criticality between all the CLBs at this iteration of placement. If two highly critical CLBs are placed in completely opposite partitions, then swap them for other non-critical CLBs to balance out the number of CLBs in each partition. All further iterations can do the distribution based on the current scheme of sorted X and Y coordinates. This will make sure that all the highly timing critical CLBs will always stay near by each other, thus reducing wirelength and also the timing criticality between them.

Moreover, in case of legalization method, we had a constraint of not being able to keep a track of any cost metric associated with the overlap removal process that we perform using the spiral technique. This is due to the fact that, say, if we want to calculate the timing criticality of the block which is moved to a new empty location, we would have to leave the MATLAB procedure and invoke VPR's timing analysis function and return back to MATLAB with this information. This back and forth would consume a lot of runtime and would defeat the whole purpose of achieving a fast global placement. If the code is written in C, we can very well achieve this and also by using such a cost metric, we can end up getting better critical path delay and overall wirelength at the end of global placement itself. Thus, the start temperature of the low temperature simulated annealing can be lowered further resulting in an additional reduction in runtime.

Finally, after testing the above implementations and certifying them for homogeneous FPGAs, an attempt should be made to apply the same technique with some modifications and test it for heterogeneous FPGA architectures.

References

[1] Marquardt, Alexander, Vaughn Betz, and Jonathan Rose. Timing-driven placement for FPGAs. Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays. ACM, 2000.

[2] Betz, Vaughn, and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. Field-Programmable Logic and Applications. Springer Berlin Heidelberg, 1997.

[3] Betz, Vaughn, Jonathan Rose, and Alexander Marquardt. Architecture and CAD for deep-submicron FPGAs. Vol. 497. Springer Science & Business Media, 2012.

[4] Rose, Jonathan, et al. The VTR project: architecture and CAD for FPGAs from verilog to routing. Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays. ACM, 2012.

[5] VPR User Manual 7.0 - https://github.com/verilog-to-routing/vtr-verilog-to-routing/blob/master/vpr/VPR_User_Manual_7.0.pdf

[6] Luu, Jason, et al. VTR 7.0: Next generation architecture and CAD system for FPGAs. ACM Transactions on Reconfigurable Technology and Systems (TRETS) 7.2 (2014): 6.

[7] Kernighan, Brian W., and Shen Lin. An efficient heuristic procedure for partitioning graphs. Bell system technical journal 49.2 (1970): 291-307.

[8] Fiduccia, Charles M., and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. Design Automation, 1982. 19th Conference on. IEEE, 1982.

[9] Yang, Saeyang. Logic synthesis and optimization benchmarks user guide: version 3.0. Microelectronics Center of North Carolina (MCNC), 1991.

[10] Hall, Kenneth M. An r-dimensional quadratic placement algorithm. Management science 17.3 (1970): 219-229.

[11] Farooq, Umer, Zied Marrakchi, and Habib Mehrez. Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization. Springer Science & Business Media, 2012.

[12] Jamieson, Peter, et al. Odin II-an open-source verilog HDL synthesis tool for CAD research. Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on. IEEE, 2010.

[13] Pistorius, Joachim, et al. Benchmarking method and designs targeting logic synthesis for FPGAs. Proc. IWLS. Vol. 7. 2007.

[14] Betz, Vaughn, and Jonathan Rose. Directional bias and non-uniformity in FPGA global routing architectures. Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design. IEEE Computer Society, 1997.

[15] Hitchcock, Robert, Gordon L. Smith, and David D. Cheng. Timing analysis of computer hardware. IBM journal of Research and Development 26.1 (1982): 100-105.

[16] Pan, David Z., Bill Halpin, and Haoxing Ren. Timing-driven placement. Handbook of Algorithms for VLSI Physical Automation (2007): 223-233.

[17] Chen, Gang, and Jason Cong. Simultaneous timing driven clustering and placement for FPGAs. Field Programmable Logic and Application. Springer Berlin Heidelberg, 2004. 158-167.

[18] Riess, Bernhard M., and Gisela G. Ettelt. Speed: Fast and efficient timing driven placement. Circuits and Systems, 1995. ISCAS'95., 1995 IEEE International Symposium on. Vol. 1. IEEE, 1995.

[19] Dunlop, Alfred E., et al. Chip layout optimization using critical path weighting. Proceedings of the 21st Design Automation Conference. IEEE Press, 1984.

[20] Burstein, Michael, and Mary N. Youssef. Timing influenced layout design. Proceedings of the 22nd ACM/IEEE Design Automation Conference. IEEE Press, 1985.

[21] Kong, Tim Tianming. A novel net weighting algorithm for timing-driven placement. Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design. ACM, 2002.

[22] Ren, Haoxing, David Z. Pan, and David S. Kung. Sensitivity guided net weighting for placement-driven synthesis. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 24.5 (2005): 711-721.

[23] Tsay, Ren-Song, and Juergen Koehl. An analytic net weighting approach for performance optimization in circuit placement. Proceedings of the 28th ACM/IEEE Design Automation Conference. ACM, 1991.

[24] Rajagopal, Karthik, et al. Timing driven force directed placement with physical net constraints. Proceedings of the 2003 international symposium on Physical design. ACM, 2003.

[25] Halpin, Bill, C. Y. Chen, and Naresh Sehgal. Timing driven placement using physical net constraints. Proceedings of the 38th annual Design Automation Conference. ACM, 2001.

[26] Choi, Wonjoon, and Kia Bazargan. Incremental placement for timing optimization. Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design. IEEE Computer Society, 2003.

[27] Srinivasan, Arvind, Kamal Chaudhary, and Ernest S. Kuh. RITUAL: A performance driven placement algorithm. Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on 39.11 (1992): 825-840.

[28] Maidee, Pongstorn, Cristinel Ababei, and Kia Bazargan. Timing-driven partitioning-based placement for island style FPGAs. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 24.3 (2005): 395-406.

[29] Kleinhans, Jürgen M., et al. GORDIAN: VLSI placement by quadratic programming and slicing optimization. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 10.3 (1991): 356-365.

[30] Sigl, Georg, Konrad Doll, and Frank M. Johannes. Analytical placement: A linear or a quadratic objective function?. Proceedings of the 28th ACM/IEEE Design Automation Conference. ACM, 1991.

[31] Spindler, Peter, Ulf Schlichtmann, and Frank M. Johannes. Kraftwerk2—a fast force-directed quadratic placement approach using an accurate net model. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 27.8 (2008): 1398-1411.

[32] Viswanathan, Natarajan, and Chris Chong-Nuen Chu. FastPlace: efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 24.5 (2005): 722-733.

[33] Hur, Sung-Woo, et al. Force directed mongrel with physical net constraints. Design Automation Conference, 2003. Proceedings. IEEE, 2003.

[34] Eisenmann, Hans, and Frank M. Johannes. Generic global placement and floorplanning. Proceedings of the 35th annual Design Automation Conference. ACM, 1998.

[35] Xu, Yonghong, and Mohammed AS Khalid. QPF: efficient quadratic placement for FPGAs. Field Programmable Logic and Applications, 2005. International Conference on. IEEE, 2005.

[36] Xu, M., Gary Gréwal, and Shawki Areibi. StarPlace: A new analytic method for FPGA placement. INTEGRATION, the VLSI journal 44.3 (2011): 192-204.

[37] Gort, Marcel, and Jason H. Anderson. Analytical placement for heterogeneous FPGAs. Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on. IEEE, 2012.

[38] Kim, Myung-Chul, Dong-Jin Lee, and Igor L. Markov. SimPL: An effective placement algorithm. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 31.1 (2012): 50-60.

[39] Lin, Tzu-Hen, Pritha Banerjee, and Yao-Wen Chang. An efficient and effective analytical placer for FPGAs. Proceedings of the 50th Annual Design Automation Conference. ACM, 2013.

[40] Funatsu, N., and Y. Takashima. Overlap-aware analytical placement based on Stable-LSE. Proc. of Synthesis and System Integration of Mixed Information Technologies (2009): 318-323.

[41] Chen, Tung-Chieh, et al. NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 27.7 (2008): 1228-1240.

[42] Ludwin, Adrian, and Vaughn Betz. Efficient and deterministic parallel placement for FPGAs. ACM Transactions on Design Automation of Electronic Systems (TODAES) 16.3 (2011): 22.

[43] Romeo, F., Vincentelli Ak Sangiovanni, and Md Huang. An Efficient General Cooling Schedule For Simulated Annealing. Proceeding of IEEE International Conference on Computer Aided Design, 1986.

[44] Kirkpatrick, Scott. Optimization by simulated annealing: Quantitative studies. Journal of statistical physics 34.5-6 (1984): 975-986.

[45] Swartz, William, and Carl Sechen. New algorithms for the placement and routing of macro cells. Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on. IEEE, 1990.

[46] Lam, Jimmy, and Jean-Marc Delosme. Performance of a new annealing schedule. Proceedings of the 25th ACM/IEEE Design Automation Conference. IEEE Computer Society Press, 1988.

[47] Chu, Chris. Placement. Electronic Design Automation: Synthesis, Verification, and Testing (2007): 635-684.

Appendix A

Acronyms

Care has been taken in this thesis to minimize the use of acronyms, but this cannot always be achieved. This appendix contains a table of acronyms and their meaning.

A.1 Acronyms

Table A.1: Acronyms

Acronym	Meaning
CLB	Configurable Logic Block
BLE	Basic Logic Element
LUT	Look-Up Table
CG	Center of Gravity
SA	Simulated Annealing
AP	Analytical Placement

Appendix B

Data Structures

This appendix contains the information about the data structure we generated in order to develop the connectivity matrix in VPR. The s_clb_info data structure shown below contains four member variables.

- **current_clb_name**
  Stores the name of the current block (CLB or I/O) from where the connectivity to its inputs (Fanin) will be traced.

- **index**
  This is the index of the entry in memory location pointed by s_clb_info. All I/Os occupy the initial locations. Once all the I/Os are parsed, the next entry starts from a CLB.

- **connections**
  This is an array which stores the index of the CLBs or I/Os connected to the current block. This information is used later while printing the connectivity matrix to a file.

- **input_elements**
  This variable stores the size of the connections array, i.e. it hold the total Fanin to the current block. If the same CLB or I/O makes multiple connections from its different output pins to the current block being parse, then same value is reflected in the input_elements value and those indices are repeated in the connections array.

The data structure is as follows:

```
struct s_clb_info {
        char *current_clb_name;
        int index;
        int *connections;
        int input_elements;
};
```