An Efficient Routability-Driven Analytic Placer for Ultrascale FPGA Architectures

by

Ziad Abuowaimer

A Thesis
presented to
the University of Guelph

In partial fulfillment of requirements
for the degree of
Doctor of Philosophy
in
Engineering

Guelph, Ontario, Canada

© Ziad Abuowaimer, April, 2018

ABSTRACT

An Efficient Routability-Driven Analytic Placer for Ultrascale FPGA Architectures

Ziad Abuowaimer
University of Guelph, 2018

Advisor:
Professor Shawki Areibi,
Professor Anthony Vannelli

Field Programmable Gate Arrays (FPGAs) continue to find increasingly wide use in commercial products as a good trade-off between CPU and ASIC, due to their flexibility and versatility. The increasing complexity and scale of modern FPGAs impose great challenges on the FPGA Computer-Aided Design (CAD) flow. However, within the FPGA CAD flow, placement remains one of the most important, time-consuming steps. In this thesis, we developed two novel routability-driven analytic placement tools for Xilinx UltraScale architectures, GPlace-pack and GPlace-flat. The former (GPlace-pack) placed third in the ISPD 2016 Routability-driven Placement Contest for FPGAs. The later (GPlace-flat) is a flat analytic placer which incorporates several unique features including a novel window-based procedure for satisfying legality constraints in lieu of packing, an accurate congestion estimation method based on modifications to the pathfinder global router, and a novel detailed placement algorithm that optimizes both wirelength and external pin count. Experimental results show that compared to the top three winners at the recent ISPD'16 FPGA placement contest, GPlace-flat is able to achieve (on average) a 7.53%, 15.15%, and 33.50% reduction in routed wirelength, respectively, while requiring less overall runtime. Despite the superiority results that are achieved by GPlace-flat compared to the state-of-the-art placers, there is no one best flow for all benchmarks. Therefore, we present a general machine-learning framework, that seeks to address the disconnect between different stages of the FPGA CAD flow that adversely affect the quality of results of the implemented designs.

Acknowledgements

I would like to express my deep gratitude and special appreciation to my supervisors Dr. Shawki Areibi and Dr. Anthony Vannelli for their help and support during my work and for the guidance they provided me while carrying out this research. I would like to thank Dr. Gary Grewal for his valuable discussions and feedback that greatly helped me enhance the quality of my work and my thesis. I would also like to thank my labmate Ryan, and the undergraduate students Dani, Timothy, Jeremy, Andrew, and Matthew for their collaboration with some of the work that is published in this thesis. Without them, my work would have been hardly possible to complete. I would like to express my thanks to the faculty and staff in the school of Engineering for their support and help. Special thanks to my colleagues in the ISLAB and my friends at the University of Guelph for their encouragement, support and the great time we spent together, special thanks to Omar Ahmed, Ahmed El-Wattar, Ahmed ElShamli, Mohamed ElMahgiubi, Dunia Jamma, and Abeer Al-Hyari. Finally, special thanks to my family. Words cannot express how grateful I am to my father, mother, wife, brothers and sister, for all of the sacrifices that you have made on my behalf.

To

my father, mother, wife, brothers, and sister

whose love and encouragement helped accomplish this thesis.

List of Publications

- **Journal Papers:**

    1. "GPlace3.0: Routability-Driven Analytic Placer for UltraScale FPGA Architectures"
       Z. Abouwaimer, D. Maarouf, T. Martin, J. Foxcroft, G. Grewal, S. Areibi, A. Vannelli
       ACM Transactions on Design Automation of Electronic Systems (TODAES), 2018.

- **Conference Papers:**

    1. "GPlace: a congestion-aware placement tool for ultrascale FPGAs"
       R. Pattison, **Z. Abouwaimer**, G. Grewal, S. Areibi, A. Vannelli
       IEEE/ACM International Conference On Computer Aided Design (ICCAD) Austin, Texas, USA, November 5, 2016.

    2. "Automatic Flow Selection and Quality-of-Result Estimation for FPGA Placement"
       G. Grewal, S. Areibi, **Z. Abouwaimer**, M. Westrik and B. Zhao
       (Best Paper Award) IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 29, 2017.

- **Poster:**

    1. "A Machine Learning Framework for FPGA Placement"
       G. Grewal, S. Areibi, **Z. Abouwaimer**, M. Westrik and B. Zhao
       International Symposium on Field Programmable Gate Arrays (FPGA 2017) Monterey, CA, USA.

Contents

1 Introduction **1**

1.1 Motivations and Challenges 2

1.2 Research Contributions 4

1.3 Thesis Organization 6

2 Background and Literature Review **7**

2.1 UltraScale FPGA Architecture 7

2.2 FPGA CAD Flow 9

2.3 FPGA Packing 10

2.3.1 Packing Problem 11

2.3.2 Related Work on FPGA Packing 11

2.4 FPGA Placement 12

2.4.1 Placement Problem 12

2.4.2 FPGA Placement Algorithms 12

2.4.3 Related Works on FPGA Placement 14

2.4.4 Wire-Length Net-Model 15

2.4.5 Optimization Techniques 17

2.4.6 Detailed Placement 18

2.4.7 Overview of the Independent Set Matching Algorithm 20

2.5 Original StarPlace (Homogeneous) 21

2.5.1 Star+ Model . . . 22
2.5.2 Star+ Jacobi iteration . . . 24
2.5.3 Legalization . . . 25
2.6 Machine Learning in EDA . . . 25
2.7 Summary . . . 27

3 Overall Methodology and Tools **28**
3.1 Routability-Driven Analytic Placers for Ultrascale FPGAs . . . 28
3.1.1 GPlace-pack . . . 30
3.1.2 GPlace-flat . . . 30
3.1.3 ML Automatic Flow Selection . . . 31
3.2 Experimental Design . . . 32
3.2.1 Benchmarks . . . 32
3.2.2 Experimental Setup . . . 33
3.3 Summary . . . 34

4 GPlace-pack **35**
4.1 GPlace-pack Flow . . . 35
4.1.1 Pin Propagation Pre-Placement . . . 37
4.1.2 Flat Initial Placement . . . 38
4.1.3 Placement-aware Packing . . . 40
4.1.4 Congestion Estimation . . . 41
4.1.5 Computing Switch Utilization . . . 41
4.1.6 Local Refinement . . . 43
4.2 ISPD 2016 Contest Winning Placers . . . 43
4.3 Preliminary Results of GPlace-pack . . . 46
4.3.1 Routing Results . . . 46
4.4 Summary . . . 48

5 GPlace-flat **50**

5.1 GPlace-flat Flow . . . 50

5.1.1 [Phase I] Pin Propagation Pre-Placement . . . 52

5.1.2 [Phase I] WL-driven Global Placement . . . 52

5.1.3 [Phase II] Congestion Estimation using a Global Router . . . 59

5.1.4 [Phase II] Cell (LUT) Inflation . . . 66

5.1.5 [Phase II] Congestion-driven Global Placement . . . 68

5.1.6 [Phase III] Dual-Objective Hierarchical Independent Set Matching . . . 70

5.2 Experimental Design . . . 74

5.2.1 Dual-Objective Interleaving Independent Set Matching (DOISM) Effectiveness Validation . . . 75

5.2.2 Congestion Estimation . . . 77

5.2.3 Window-based Bi-partition Legalization Effectiveness Validation . . . 77

5.2.4 Runtime Analysis . . . 78

5.2.5 Comparison with Previous Works . . . 79

5.2.6 Vivado Router Runtime Comparisons . . . 82

5.2.7 Timing Results Comparisons . . . 82

5.2.8 Extended Experimental Results . . . 83

5.3 Summary . . . 86

6 Automatic Flow Selection for FPGA Placement **87**

6.1 Introduction . . . 87

6.2 Methodology . . . 89

6.2.1 Overall Framework . . . 89

6.2.2 Benchmarks used for Training . . . 92

6.2.3 Feature Extraction . . . 93

6.2.4 Placement Flows . . . 93

6.2.5 Training the Classification System . . . 96

6.2.6 Evaluation Criteria 96

6.3 Results 97

6.3.1 Experimental Setup 97

6.3.2 Framework Outcomes 98

6.4 Summary 103

7 Conclusions and Future Work **105**

7.1 Conclusions 106

7.2 Future Work 107

Bibliography **109**

List of Tables

3.1 ISPD 2016 Placement Contest Benchmark Statistics 33
3.2 Range of Key Circuit Features for 372 Benchmarks [47] 33
3.3 List of Academic Placers for Ultrascale FPGAs 33

4.1 ISPD 2016 Placement Contest Winners Flows Comparisons 43
4.2 Comparison with ISPD'16 Contest Winners on ISPD 2016 Benchmark Suite 47

5.1 Comparison Between Different Congestion Estimation Methods 61
5.2 PathFinder Global Router Parameter Definitions 62
5.3 Comparison Between Dual-Objective ISM (DOISM) and ISM in [5] 75
5.4 Routed Wirelength Before and After Dual-Objective ISM (DOISM) 76
5.5 Comparison Between Different Congestion Estimation Methods 77
5.6 Routed Wirelength w. and w/o Window-Based Legalization 78
5.7 Runtime Breakdown of GPlace-flat in (seconds) 79
5.8 Comparison with ISPD'16 Contest Winners on ISPD 2016 Benchmark Suite 80
5.9 Comparison with State-of-the-Art Academic FPGA Placers 80
5.10 Vivado Router CPU Time in (seconds): GPlace-flat vs. UTPlaceF3.0 [5] 82
5.11 Timing results Comparison with State-of-the-Art Academic FPGA Placers 83
5.12 Performance Comparison with mPFGR vs. WLPA 84
5.13 Performance Comparison w. and w/o Window-Based Legalization 84
5.14 Performance Comparison of DOISM vs. Post Global Placement 85

5.15 Comparision with the state-of-the-art on 372 Xilinx Benchmarks 85

6.1 Range of Parameters for 372 Circuits 92

6.2 Circuit Features 93

6.3 Predicting Best Flow: WL Placement 98

6.4 Predicting Best Flow: WL Routing 99

6.5 Predicting Best Flow: Timing 100

6.6 Predicting Best Flow: Power 100

6.7 Regression Models: Linear vs. RF 101

6.8 Regression: Runtimes and Speedup 102

6.9 Performance of Other Regression Models 102

List of Figures

1.1 Thesis Contributions . . . 5

2.1 Illustration of Xilinx UltraScale Architecture . . . 8
2.2 Tile Architecture in Xilinx UltraScale . . . 9
2.3 A Typical FPGA CAD Flow . . . 10
2.4 Taxonomy of FPGA Placement Techniques . . . 13
2.5 Half-Perimeter Wirelength (HPWL), Clique, and Star Models . . . 16
2.6 B2B Net Model . . . 17
2.7 Independent Set Matching Steps . . . 20
2.8 StarPlace (Homogeneous) Flow . . . 22

3.1 Overall Methodology of Proposed Framework . . . 29

4.1 GPlace-pack Flow . . . 37
4.2 Computing WLPA with Prefix Sums (WLPA=1) . . . 42
4.3 Proposed Flows of ISPD 2016 FPGA Contest Winners . . . 44
4.4 Convergence of Star+ Cost during GPlace-pack Phases . . . 48

5.1 GPlace-flat Flow . . . 51
5.2 Window Based Legalization Flow . . . 53
5.3 Control Signal FF bi-partition Tree . . . 58

5.4 Congestion Map of FPGA05 in ISPD2016 contest: (a) WLPA; (b) PFGR (3 iters.)/VPR-PF costs; (c) PFGR (3 iters.)/NCTUgr2.0 cost; (d) mPFGR (3 iters.); (e) Vivado detailed router 60
5.5 mPFGR with Different Cost Functions: a) FPGA05, and b) FPGA01 61
5.6 Resolving Congestion via Cell Inflation 67
5.7 LUT bipartitioning with Cell-density 69
5.8 Dual-Objective Independent Set Matching 70

6.1 ML EDA Framework: Training 90
6.2 ML EDA Framework: Testing & Updating 91

Abbreviations

I/O : Input/Output

CAD : Computer Aided Design

SA : Simulated Annealing

CLB : Configurable Logic Blocks

VPR : Versatile Place-and-Route

CPU : Central Processing Unit

FPGA : Field Programmable Gate Array

ASICs : Application Specific Integrated Circuits

DSP : Digital Signal Processor

RAM : Random Access Memory

HPWL : Half Perimeter Wire-length

QoR : Quality of Result

NP : Non-deterministic Ploynomial-time

LUT : Lookup Table

FF : Flip-Flop

mPFGR : modified Path Finder Global Router

WLPA : Wire-length per Area

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) continue to find increasingly wide use in commercial products, due to their flexibility and versatility. However, within the FPGA *Computer-Aided Design (CAD)* flow, *placement* remains one of the most important, time-consuming steps. Given a netlist representation of a circuit, placement involves mapping functional blocks in the netlist to legal locations on the FPGA in a way that optimizes one (or more) objectives. The most basic objective involves minimizing the sum of the total (estimated) *wirelength* for each net. However, as commodity FPGAs grow in size and complexity, *routability* is also becoming an increasingly important placement objective. This is because the amount and type of interconnect available in different regions of the FPGA is fixed; hence, placements that fail to spread congestion throughout the FPGA often incur subsequent routing demands that exceed the prefabricated routing resources available in certain regions of the device. The end result is that the ensuing routing step either fails, or the router is forced to work unreasonably hard to obtain success. Either scenario is undesirable, as placement and routing runtimes are already excessive [1].

In this thesis, we aim to develop a routing-aware analytic placer for Xilinx UltraScale FPGAs that consistently generates high-quality placements while exhibiting good runtime performance. We have targeted UltraScale FPGAs because they are large, state-of-the-art heterogeneous based devices that make finding even a legal placement challenging. Moreover, due to the increasing

runtime for key optimization stages in the FPGA CAD flow and larger amounts of data generated and passed from one stage to the next, more pressure is imposed at each stage to generate high-quality solutions appropriate for later stages in the flow. This trend provides impetus to continue to explore other approaches that can be applied to problems throughout the CAD flow. Machine Learning (ML) can assist a designer with decision making and CAD tools with problem solving. For example, different placement tools may use different flows, resulting in placements with different wirelength, critical-path delay, power consumption, etc. Given a new circuit to place, an ML model can be trained and used to recommend the ”best” placement flow, where best may refer to runtime or Quality-of-Result (QoR). Current algorithms do not discriminate between benchmarks when performing the search for a near optimal solution. ML can act as a recommender and guide to perform specific optimization strategies either before a given stage or during a given stage in the CAD flow.

1.1 Motivations and Challenges

Packing and placement are two key steps to achieve high-quality physical implementation with good routability. Despite advancements in literature, current FPGA packing and placement algorithms still suffer from several challenges:

1. Quality of results and runtime trade-off: As the size of FPGA designs and the complexity of modern FPGA architectures increase, sophisticated rules/constraints pose more restrictions on FPGA placement. Compared to the traditional Simulated Annealing (SA) approach, analytical placement has shown great promises in solution quality and scalability. Unlike ASICs, FPGAs are discrete in nature and a continuous algorithm may not always achieve superior QoR. Thus, continuous global placement results need to be improved in order to successfully apply ASIC placement flow for FPGAs.

2. Simultaneous Packing and Placement: Traditionally, packing and placement are separated stages in an FPGA CAD flow to reduce the complexity of each subproblem. However,

packing algorithms that ignore physical information and only considers the affinity of a logic module may result in poor wirelength. Therefore, to achieve better placement results, stronger integration between packing and placement is desirable. Existing packing algorithms suffer from the following limitations:

- Logical packing algorithms, such as those proposed in [2, 3], perform packing based only on logical connectivity which could cluster cells that are physically far apart. This leads to wirelength-unfriendly netlists that may degrade routability.
- Placement-aware packing techniques, such as those proposed in [4, 5], consider physical locations of cells during packing, may still produce netlists that are wirelength-unfriendly. These placers perform packing based on physical locations generated by first performing a *Flat Initial Placement (FIP)*. The FIP does not consider control set constraints associated with the Flip-Flops (FFs). However, designs with large numbers of control sets (e.g., clocks, resets and control enables) are common in modern high-end FPGAs. This reduces the logic utilization and hence forces the packing method to pack FFs that are far apart to fit into the FPGA area, possibly degrading wirelength and routability.

3. Simultaneous Placement and Routing: Routing quality is highly dependent on the placement solution. Therefore, to better handle routability, the constraints induced by the routing architecture and resources should be addressed during global placement.

4. New approach to reduce the compilation runtime: Most of the key optimization problems encountered in the FPGA design process today are NP-complete [6]. As FPGAs continue to scale in accordance with Moore's law, so does the size of the applications targeted for them. This trend provides impetus to continue to explore other approaches that can be applied to problems throughout the CAD flow. Machine learning models can provide a useful metrics prediction, like congestion, that can guide the placement process to enable high-quality results.

1.2 Research Contributions

The theme and the major contributions of this thesis are shown in Figure 1.1. A brief description of the main contributions in this thesis can be categorized in three major paradigms:

1. **Congestion-aware FPGA Placer**: A novel congestion-aware placement tool for Xilinx's UltraScale architectures are presented in Chapter 4. The proposed placement algorithm, called GPlace-pack, has the following main features:

   - A placement-aware packing that satisfies all hard constraints such as; LUT-sharing, FF control set constraints, and incorporates physical information based on flat global placement.
   - A fast method for estimating congestion that is independent of the placement quality is implemented, making its use early in the placement process feasible.

2. **Routability-driven FPGA Placer**: The proposed placement algorithm in Chapter 5, called GPlace-flat, seeks to optimize both wirelength and routability. The main contributions of this work are listed below:

   - A novel window-based bi-partitioning legalization procedure for legalizing flat placements is proposed. The procedure satisfies all hard constraints imposed by the UltraScale architecture, while minimizing cell displacement. Within GPlace-flat it eliminates the need for an initial packing step, and allows the placer to optimize globally.
   - A faster, more accurate version of the pathfinder global router [7] is proposed (mPFGR) for performing congestion estimation. Within GPlace-flat, the mPFGR congestion estimation is used to guide cell inflation/deflation optimizations.
   - A dual objective independent set matching algorithm is proposed for improving both wirelength and external pin count. Within GPlace-flat it is used to perform detailed placement.

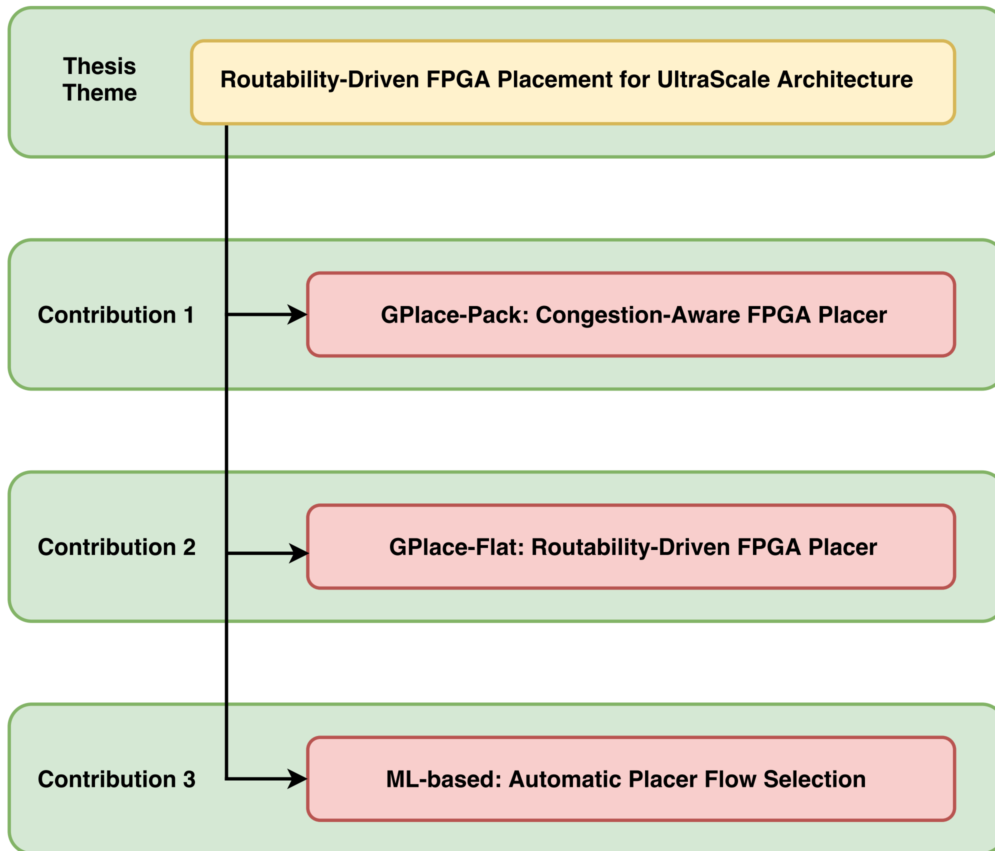


Figure 1.1: Thesis Contributions

- Experimental results based on the ISPD'16 Routability-Driven FPGA Placement Contest benchmark suite show that GPlace-flat outperforms the top three [8] contest winners with respect to runtime, routed wirelength, and number of routable placements.
- Experimental results based on 360 additional benchmarks received from Xilinx Inc. show that GPlace-flat also outperforms the latest (improved) placers from the top 3 contest winners [4, 5, 9, 10] across a wide range of metrics, including routed wirelength, number of best solutions found, fewest number of unroutable placements, and placement and routing runtimes.

3. **Machine Learning: An Automatic Flow Selection**: In Chapter 6, the potential capabilities of the framework are demonstrated by applying it to the *placement* stage of the FPGA flow. The main contributions of this work include the following:

- We propose a novel ML-based framework for recommending the most appropriate placement procedure to use for a new circuit. Seven different academic placement flows are considered, each based on the award-winning placer in [9], as well as the Vivado placement tool. Four different placement objectives are considered: minimizing estimated wirelength, routed wirelength, critical-path delay, and power.
- We demonstrate the flexibility of the proposed framework by also using it to efficiently and accurately predict various quality metrics without performing placement and routing, like estimated wirelength, routed wirelength, critical-path delay, power, post-routing column and row utilization, and a circuit's Rent exponent.

1.3 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 provides essential background information and discusses the main previous works in FPGA packing and placement, routability-driven FPGA placement, and applying machine learning in the Electronic Design Automation (EDA) field. The overall methodology that describes the different phases of the research along with the benchmarks and experimental setup used in this thesis are introduced in Chapter 3. The proposed congestion-aware FPGA placer, GPlace-pack, for Ultrascale FPGA architecture is introduced in Chapter 4. Chapter 5 introduces a novel routability-driven flat analytic FPGA placer for Ultrascale FPGAs. This work contains several unique features such as: integrate global routing with global placement to address the constraints induced by routing architecture and resources during the placement stage, and develop a novel detailed placement algorithm, based on enhancements to *Independent-Set Matching (ISM)* [5] to minimize wirelength and external pin count, with the goal of improving both wirelength and routability. A novel prediction/classification framework based on machine learning techniques to improve the quality of results in FPGA placement implementation is discussed in Chapter 6. Finally, the thesis conclusion is provided in Chapter 7 along with potential future work.

Chapter 2

Background and Literature Review

In this chapter, necessary background material that covers the Ultrascale FPGA architecture, FPGA CAD flow, FPGA packing, FPGA placement, and wire models will be presented. More emphasis will be on analytic placement and star+ net-length model since they are more relevant to this thesis. This chapter also discusses the prior works related to the key contributions in this thesis, which are the development of routability-driven FPGA placer for modern FPGAs, and an automatic placement flow selection based on Machine learning techniques to improve quality-of-results. State-of-the-art FPGA packing and placement algorithms, and the incorporation of packing and placement to improve routability in modern FPGAs are discussed in this chapter. Finally, we highlight the related prior works of applying Machine learning in EDA. The main objective of this chapter is to help the reader understand several topics related to this thesis.

2.1 UltraScale FPGA Architecture

Xilinx UltraScale FPGAs [11] consist of heterogeneous programmable logic blocks, such as general logic (i.e., LUTs), random access memory blocks (i.e., BRAMs), and digital-signal processing blocks (i.e., DSPs). The detailed architecture and layout is illustrated in Fig. 2.1. As well, there are prefabricated routing segments of different lengths in both horizontal and vertical directions. Slices contain both LUTs and FFs, which when grouped together share a single switch for rout-

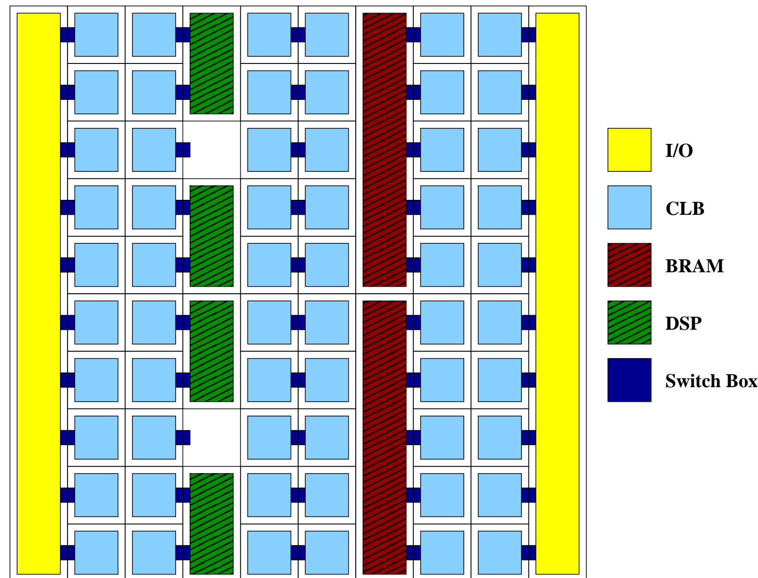


Figure 2.1: Illustration of Xilinx UltraScale Architecture

ing. Switch boxes provide both intra- and inter-slice connectivity. As shown in Fig. 2.2, slices are placed side-by-side such that each pair of slices share the same switch box in a back-to-back formation. Each slice has eight *Basic Logic Elements (BLEs)*, and each BLE contains a single 6-input LUT and two FFs. The 6-input LUTs can be configured to implement either one logic function of up to 6-inputs with one output, or two distinct logic functions of up to 5-inputs and two separate outputs. To implement two distinct functions in the same LUT, the sum of the distinct inputs of the two functions must not exceed five. There are only two clock (CLK) signals and two set/reset signals per slice. The bottom 4 BLEs share the same CLK and RESET signals, while the upper 4 BLEs use the second CLK and RESET signals. Every group of four BLEs is further divided into two subgroups, each of which must share the same clock-enable (CE) signal. These control-set constraints, coupled with the other placement constraints, make the placement problem extremely challenging [8]. Even if all of the architectural constraints are satisfied by the current placement, there is no guarantee that the placement will be routable. Moreover, different placement solutions may require the router to perform more or less work to find a feasible routing solution.

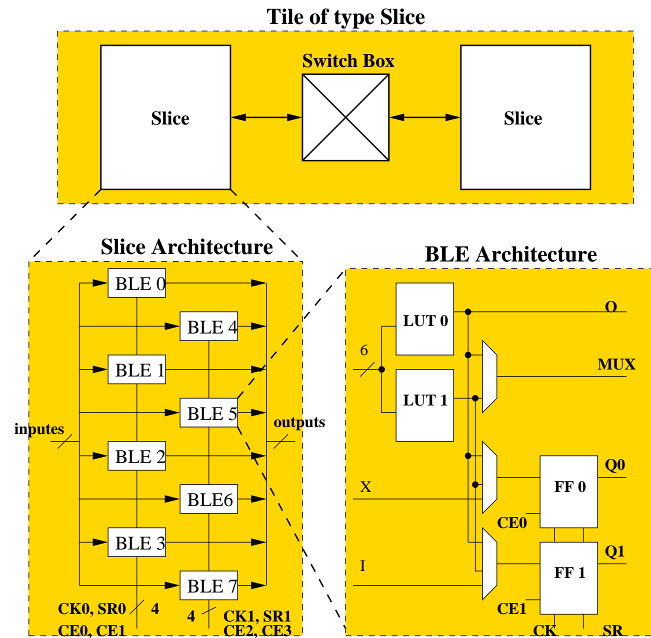


Figure 2.2: Tile Architecture in Xilinx UltraScale

2.2 FPGA CAD Flow

Figure 2.3 shows a typical FPGA CAD flow. The CAD flow takes the HDL design such as Verilog or VHDL as an input and converts the design into bit streams to configure the FPGA device through a sequence of translation and optimization steps. During the synthesis phase, the HDL design is translated to RTL logic netlist and optimized for some metrics such as; performance, area, and power. This optimization is independent of the target device and technology. Next, the RTL logic netlist is mapped into the available resources on the target FPGA architecture such as look-up tables (LUTs), and Flip-Flops (FFs) during technology mapping phase. The results of the technology mapping phase is called, technology mapped netlist. A netlist is a collection of nets, which are a set of logic elements that are connected together. In the packing phase, several LUTs and FFs grouped together into a basic logic elements (BLE), and then several BLEs packed into a complex logic block (CLB) subject to some constraints. Next, the placement phase assigns the logic elements in the netlist to physical locations of resources on the FPGA while optimizing one or more objectives such as: wirelength, routability, timing, and power. After placement, the routing

phase assigns nets to proper routing resources to connect the logic elements together. The routing resources are prefabricated in the FPGA, therefore, the router may not be able to route placements with severe congestion. Finally, if the router is successful in routing all the signals (nets), a bit stream is generated and used to configure the target FPGA device.

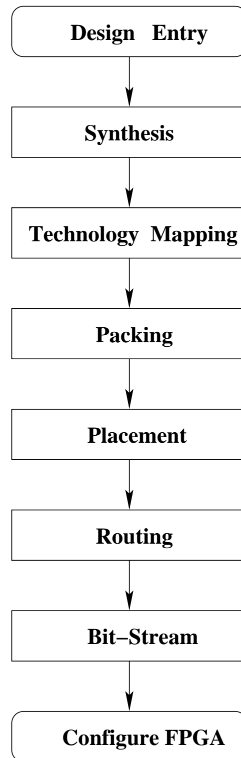


Figure 2.3: A Typical FPGA CAD Flow

2.3 FPGA Packing

In a traditional FPGA CAD flow, packing and placement stages are separated to reduce the complexity of each subproblem. Logical packing techniques that ignores the physical information of the cells may cluster cells that are physically far apart [12, 13]. Therefore, those logical packing algorithms may produce wirelength unfriendly netlists that restrict the placement optimization and hence leads to a suboptimal placement solution.

2.3.1 Packing Problem

Traditionally, packing falls between technology mapping and placement steps in the FPGA CAD flow, but recently it becomes more tightly integrated with the placement process. Packing typically forms complex logic elements from a technology mapped netlist in two levels of hierarchy. First, a basic logic block (BLE) is formed by grouping several look-up tables (LUTs) and Flip-Flops (FFs) together. Second, several BLEs are clustered together into a complex logic block (CLB) while satisfying all architecture (hard) constraints. The packing step in the FPGA CAD flow is the process wherein technology mapped elements, such as LUTs and FFs, are packed into the available FPGA hardware resources such as a complex logic block (CLB).

2.3.2 Related Work on FPGA Packing

FPGA packing can be classified into three categories: (i) seed-based packing, (ii) cluster-merging packing, and (iii) partitioning-based packing. Seed-based packing randomly selects a cell as a seed and then keeps merging unpacked cells into the seed based on an attraction function until the capacity limit is reached. This packing approach is able to pack too densely, but lacks the global view of connectivity. Several seed-based packing algorithms are proposed in the literature based on different objectives and attraction functions such as: VPack [14], T-VPack [15], RPack [16], MO-Pack [17], iRAC [18], and AAPack [19]. Cluster-based packing chooses the best pair with highest attraction for merging. An example of cluster-merge FPGA packing is HDPack [20], which adopts the best choice (BC) clustering. Usually BC cluster-based packing produces clusters with half utilization (i.e. loose packing), therefore, it combines with seed-based packing to avoid loose packing. Partitioning-based packing, like PPack [21], applies recursive partitioning followed by moving illegal CLBs to get a legalized packing solution.

2.4 FPGA Placement

2.4.1 Placement Problem

FPGA placement is the process of mapping the cells in a circuit netlist to sites on an FPGA. A netlist is represented by a hyper-graph $G_h(V, E_h)$, where vertices V is a set of cells (i.e., LUTs, FFs, RAMs, DSPs, and IOs) to be placed and the hyper-edges E_h is a set of nets. Each net is a subset of V that specifies the connections to be made between cells. The architecture defines a rectangular region with width W and height H divided into columns each dedicated to one cell type (i.e., slice, RAMs, DSPs, and IOs). The placement must satisfy all architectural constraints associated with FF control sets, LUT-sharings, block and column type matching. In routability-driven placement, the goal is to successfully route the design by placing cells on the chip such that the nets have minimal overlap with one another.

2.4.2 FPGA Placement Algorithms

Unlike an ASIC, FPGA placement area has predefined sites, where modules can only be placed at, that are arranged in a regular array, which makes it more constrained and challenging to solve. Modern FPGA placement algorithms can be categorized into three major approaches: partitioning, simulated annealing, and analytical based placement.

2.4.2.1 Simulated Annealing-Based Placement

In this type of placers, the module positions are perturbed based on simulated annealing [22] to optimize the placement solution. Simulated annealing is an important method for FPGA placement due to its flexibility in handling complex objective functions. The well-known VPR [23] is the best academic SA-based FPGA placer. Through VPR, the authors introduced several key improvements to FPGA placement, including fast incremental Half Perimeter Wire-Length (HPWL) computations, a well-tuned annealing schedule, timing-driven clustering, and the concept of path-based weighting for timing-driven placement. Simulated annealing can often achieve good quality

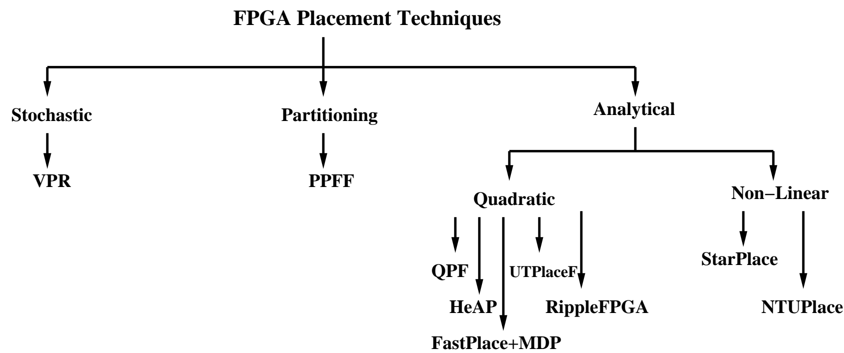


Figure 2.4: Taxonomy of FPGA Placement Techniques

placement on small designs, however, with the existence of big macros the module perturbation may not be trivial for large-scale designs. The lack of scalability makes simulated annealing based placement inappropriate for large-scale circuits.

2.4.2.2 Partitioning-Based Placement

Min-cut or partitioning-based placement [24] applies a recursive top-down partitioning to divide the circuit and FPGA region into sub-circuits and subregions, and then assign sub-circuits into subregions. The main drawback of min-cut based placement is that the global information from previous partitioning steps can be lost once the partition step is performed, thus deteriorating the solution quality.

2.4.2.3 Analytical Placement

The analytical placement problem is composed of an objective function and a set of placement constraints that is formulated as a mathematical programming problem to optimize the objectives through analytical optimization techniques. In large-scale ASIC designs, analytical placement can achieve better placement quality as shown in ISPD placement contests. However, recently more works were carried out on analytical placement for FPGAs. Modern analytic placers are typically performed in three steps: global placement, legalization, and detailed placement which are considered to be the main ingredients of the analytical placement flow.

2.4.3 Related Works on FPGA Placement

FPGA placement algorithms can be classified into three main categories based on the optimization approach employed: simulated-annealing, partitioning, and analytic, as shown in Figure 2.4 . Simulated-annealing based placers, like the most famous academic FPGA CAD tool VPR/VTR [23, 25], are able to achieve high-quality placement solutions, but at the expense of long runtimes. Placement algorithms based on partitioning, like PPFF [7], employ recursive top-down partitioning to divide the circuit and FPGA into sub-circuits and sub-regions by minimizing the number of cuts between sub-regions. However, placement quality often suffers because wirelength is not directly minimized. Analytic approaches are used extensively today in many academic and industrial frameworks since they produce high quality results compared to partitioning based approaches and in much less running time than simulated annealing. Analytic placement algorithms, like QPF [26], StarPlace [27], HeAP [28], and LLP [29] use smooth functions to approximate a non-smooth wirelength cost function, and solve a system of equations using efficient numerical methods. However, analytic placers often have difficulty dealing with hard constraints imposed by the FPGA architecture. Moreover, most analytic FPGA placers [26, 27, 30] consider homogeneous, island-style FPGA architectures. More recently, academic analytic placers for heterogeneous FPGA architectures have started to appear in [28, 29, 31, 32].

Many of the analytic FPGA placers (e.g., [28, 29, 31]) employ wirelength models that do not consider the type of segmented-routing architectures used in today's modern FPGA architectures. Two exceptions are [30, 32], both of which seek to minimize segmented-routing wirelength. None of the FPGA analytic placers mentioned above directly model congestion.

Packing has been previously used in some FPGA flows to address congestion. The approach in [3] employs loose packing to avoid over utilization of *Configurable Logic Blocks (CLBs)*, thus reducing congestion. However, this strategy can rapidly lead to the under utilization of CLBs resulting in the circuit failing to fit on the FPGA. Moreover, the uniform under utilization of CLBs can also lead to an increase in total wirelength and hence congestion. To compensate, Un/DoPack [2] first identifies congested regions, unpacks CLBs in those regions, and then re-packs those CLBs

with a reduced cluster size. However, the entire process is extremely expensive, because it requires using the actual router to identify the congested regions of CLBs. Also, Un/DoPack does not handle the congestion that is introduced by "flyover" nets (i.e., nets which do not have any pins in that congested region).

Unlike academic FPGA architectures, commercial FPGAs have more hard constraints that limit and complicate traditional packing algorithms that are primarily based on circuit connectivity. Most analytical placers developed for FPGAs in the past did not consider specific routing congestion issues in their flows, which causes some complication in routing these netlists. However, in the spirit of the ISPD 2016 Routability-Driven FPGA Placement contest, a few routability-driven analytic placers [4,5,9,10] targeting a modern Xilinx UltraScale FPGA architecture are published.

2.4.4 Wire-Length Net-Model

The total wirelength of the routed wires is not known during the placement stage. Therefore, a fast and accurate wirelength estimation model should be used to predict final routing wires. There are many wirelength estimation models such as half-perimeter wirelength (HPWL) and Star model based on the center of gravity of a net. The most widely used model is HPWL since it provides exact optimal wirelength for two and three pins as well as it can be computed and updated efficiently. However, HPWL is not differentiable and smooth, hence it is hard to find its minimum value. Many smooth and differentiable approximations of HPWL are used to model the wirelength such as: quadratic model, bound to bound model (B2B), and logarithm-sum-exponential (LSE) model. Star+ model is a modified version of the star model that is near linear while still differentiable.

2.4.4.1 Quadratic Model

The Quadratic model is suitable only for two-pin connections therefore multi-pin connections must be broken down into two-pin connections using either a clique model, star model, or hybrid of the two models. The clique and star models of the multi-pin net are shown in Figure 2.5. The cost function is calculated as the sum of the quadratic length of all connections and the net weight

is used to adjust the approximation of the linear HPWL as shown by the following equation for x-direction:

$$L_x = \frac{1}{2}\sum_{i=1}^{P}\sum_{j=1}^{P}\gamma_{x,ij}(x_i - x_j)^2 \tag{2.1}$$

Where P is the number of pins in a net and $\gamma_{x,ij} = \frac{1}{P}$ for clique model, and x_i, x_j are the x-positions of blocks connected to a net.

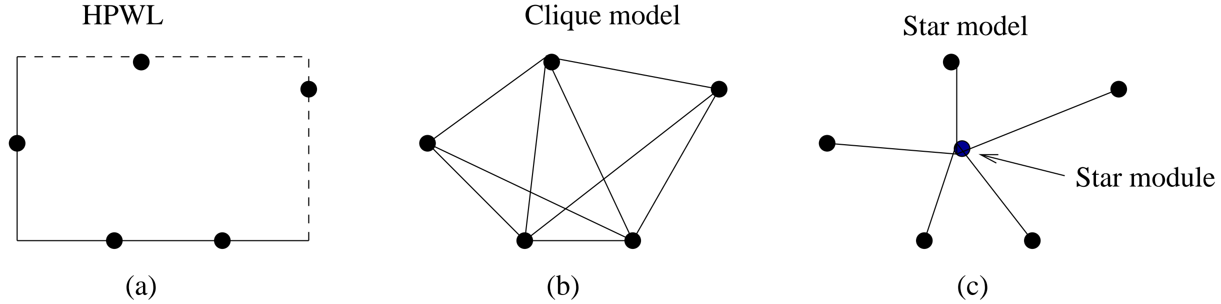


Figure 2.5: Half-Perimeter Wirelength (HPWL), Clique, and Star Models

2.4.4.2 B2B Model

The HPWL model tends to ignore the inner connections while the inner connections of the clique net model contribute to the clique length which leads to a high approximation error of the total wirelength model. The B2B model removes all inner connections and hence all connections are joined to the two boundary modules as shown in Figure 2.6. According to the B2B model the net weight of P-pin net is defined by Equation 2.2.

$$\omega_{x,ij} = \frac{2}{P-1} \times \frac{1}{|x_i - x_j|} \tag{2.2}$$

By applying the above connection weight to the B2B model the differentiable quadratic objective function exactly matches the HPWL as shown by Equation 2.3.

$$L_x^{BB} = \frac{1}{2}\sum_{i,j\in N}\omega_{x,ij}(x_i - x_j)^2 = max_{v_i\in e}x_i - min_{v_i\in e}x_i \tag{2.3}$$

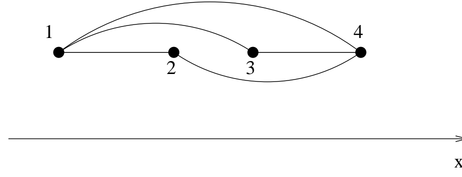


Figure 2.6: B2B Net Model

2.4.4.3 LSE Model

A logarithm-sum-exponent (LSE) can be used to accurately approximate and smooth the HPWL as shown by Equation 2.4.

$$LSE_x = \gamma \left(log \sum_{v_k \in e} e^{\frac{x_k}{\gamma}} + log \sum_{v_k \in e} e^{\frac{-x_k}{\gamma}} \right) \tag{2.4}$$

When γ is equal to zero, the LSE model is reduced to the exact HPWL. However, in practical implementation, a small reasonable γ value is chosen to avoid arithmetic overflow.

2.4.5 Optimization Techniques

Based on the literature covering mathematical optimization methods, most analytic placers fall into two categories: quadratic programming, and non-linear programming.

2.4.5.1 Quadratic Programming

The objective function of a quadratic wirelength model is optimized as follows:

$$min\ Q_x = \frac{1}{2} x^T Q_x x + C_x^T x + d_x \tag{2.5}$$

The hyper-edge connectivity is represented by the Hessian matrix Q_x. The Hessian matrix is symmetric and positive definite if a sufficient number of blocks (usually I/O blocks) are fixed [33]. The wirelength along the x and y directions are separable and hence can be solved separately.

This optimization problem has a unique minimizer since it is strictly convex. Using the conjugate gradient method (CG) the positive definite system of linear equations can be solved efficiently.

2.4.5.2 Non-linear Programming

Usually a penalty method is used to optimize the non-linear placement problem due to the use of analytical wirelength and overlap reduction model. Many non-linear placers for ASIC design use a LSE wirelength model and some smooth density control functions as in Equation 2.6. Solving non-linear problems is very time consuming [34] and therefore the multilevel approach is often used.

$$min\ LSE(x, y) + \lambda \sum_{b} (D_b(x, y) - M_b)^2 \tag{2.6}$$

2.4.6 Detailed Placement

Detailed placement is the final step that is performed after global placement and legalization. The goal of this step is to further improve the final quality of the placement solution. A large body of detailed placement techniques introduced and proposed by several researchers can be classified into the following classes. The first class of the detailed placement techniques uses branch-and-bound to compute an optimal cell reordering using a sliding window. This technique is limited by a very small window size, which constraints the optimization benefit, due to the computational complexity. The second class is based on stochastic search techniques. Historically, simulated annealing has been viewed as a poorly-scalable, time-consuming strategy for detailed placement. Therefore, a greedy (zero-temperature) or low-temperature simulated annealing is used for detailed placement. Another method of detailed placement, like RippleFPGA [10], computes the optimal interval for each given cell that results in the minimum bounding box for all nets connected to the cell and then attempts to place cells within their optimal intervals. The fourth common detailed placement method applies the augmenting path algorithm to solve a min-cost bipartite matching. The idea of bipartite matching can be applied to optimize wirelength. In order to find min-cost matching, the changes in the total wire length that would result from moving each element to

each location are used as the edge weights of a bipartite graph. However, the optimal HPWL improvement cannot be guaranteed by solving this matching problem, since the edge weight of a cell depends on the positions of other connected cells in the same matching set. To overcome this drawback, authors in NTUPlace3 [34], and UTPlaceF [4, 5] restrict the bipartite matching to independent set matching (ISM), where the cells are chosen such that none of them are directly connected to any other cell in the set, meaning they do not share any nets. As a result, the change in wire length from moving one of the cells is independent of the location of all the other cells in the set. Performing a min-cost bipartite matching can then guarantee the minimum possible wire length that is possible from moving the elements in the set. In order to increase the solution space for ISM, empty locations that do not currently contain an element are also considered. To add one of these empty locations to a set for ISM, a virtual "white space" element is added to the set of elements. This white space is matched to a location just like any other element, and represents leaving that location empty. Several white spaces can be included in a single independent set. Since a white space is a virtual element and so is not connected to any wires, the wire length cost of moving it to any location is zero. To further improve the results, UTPlaceF [4, 5] used a congestion-aware restriction on ISM to help prevent the creation of congested regions where there are not sufficient routing resources. To make ISM congested aware, three restrictions are used which can disallow some moves. First, an element cannot be moved to a location that is congested, though it is always allowed to remain in its initial location. Secondly, a white space that is in a congested region cannot be moved. Finally, unlike other elements, a white space element that is not in a congested location can be moved into a congested region. In UTPlaceF [5], ISM is hierarchically applied to CLBs, BLEs to help obtaining better improvements by escaping local minimum. In this thesis, we adopt ISM in UTPlaceF [5] to optimize for dual objectives to further improve wirelength and routability. Therefore, a detailed description of the ISM will be presented in the following section.

2.4.7 Overview of the Independent Set Matching Algorithm

The independent set matching algorithm can be broken down into 4 main steps. First an independent subset of elements must be chosen from the set of elements in the circuit. Next, a bipartite graph is built from the chosen set of elements and the set of their locations. The bipartite graph is then solved for the cost-minimizing matching. Finally each element is moved to the new location determined by the bipartite matching.

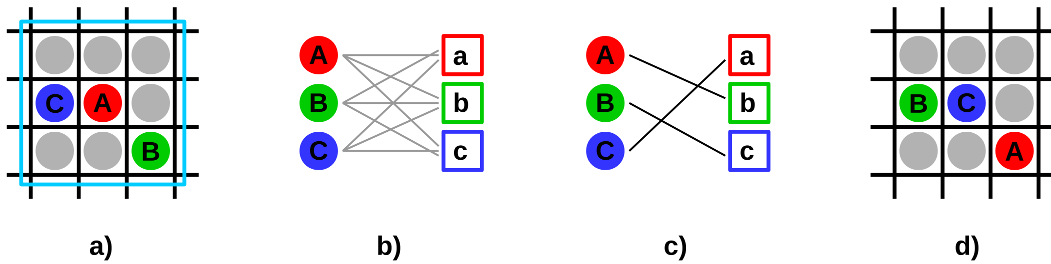


Figure 2.7: Independent Set Matching Steps

Generating an independent set: To begin generating a set, a window is chosen that encompasses a number of physical locations on the FPGA, as shown in Figure 2.7a. One at a time, the algorithm attempts to add each element and white space in the window to the set. If a proposed element shares a net with one of those already in the set it is rejected, otherwise the element is added to the set. This process continues until either the maximum set size is reached, or the algorithm finishes attempting to add every element and white space in the window.

Building a bipartite graph: A bipartite graph is constructed using a set U containing the elements found in the first step, and a set V that contains the current locations occupied by each of those elements. Edges are added connecting each element in U to each location in V. Figure 2.7b shows an example of such a graph. Each edge (u, v) with $u \in U$ and $v \in V$ is given a weight that corresponds to the increase in the total wire length that would result from moving element u to location v. However, when using congestion-aware ISM not every edge is added in order to adhere to the congestion restrictions. While every element maintains a connection to its current location,

a non white space element does not form an edge connecting it to any other location with routing congestion. A white space element that is currently in a congested location does not form any additional edges. In addition to congestion, there may be further restrictions on element movement for some element types, such as FFs, that are necessary to ensure the final placement is legal. If placing the element in a location would not be legal, that edge is not added to the graph.

Solving the bipartite matching: Once the bipartite graph has been constructed, there are several existing algorithms that can be used to find the cost-minimizing matching. Because $|U| = |V|$ since elements and locations were added to the graph as pairs, the result matches each element u to a distinct location v, as shown in Figure 2.7c. The resulting matching provides the configuration of the chosen elements in the chosen locations with the lowest wire length.

Moving the nodes: In this final step, each element is moved to the location found in the bipartite matching. As long as the elements were initially at legal locations, the legality is preserved and the new placement is valid. Figure 2.7d shows the resulting placement from the example.

In order to improve the entire global placement, ISM must be run for windows covering the entire circuit, and every element in the circuit must be included in ISM. Continued iterations can provide further improvement to the placement. When considering congestion, it is also necessary to periodically update the congestion estimate to take into account the elements that have been moved.

2.5 Original StarPlace (Homogeneous)

Both GPlace-pack and GPlace-flat that are proposed in this thesis employ the same wirelength model used in StarPlace [27], as well as similar legalization procedures. However, original StarPlace does not directly support heterogeneous architectures and does not optimize for congestion. The StarPlace flow is illustrated in Figure 2.8. StarPlace uses VPR's TV-Pack to first pack LUTs and FFs into CLBs to make the placement homogeneous. Then, in order to avoid trivial solutions, I/O blocks are initially fixed; logic blocks are assigned random locations. StarPlace then performs a

maximum number of iterations where wirelength is optimized based on the Star+ [27] model. The Star+ model along with the derivation of its Jacobi iteration, and bi-partition legalization procedure are described in the subsections that follow.

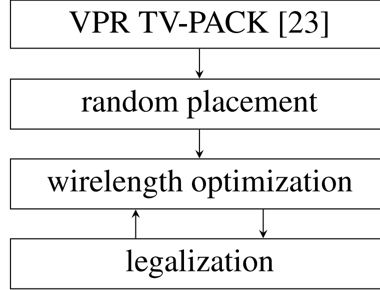


Figure 2.8: StarPlace (Homogeneous) Flow

2.5.1 Star+ Model

Both the clique and star models are inaccurate models of linear wirelength, because they estimate the linear distance between blocks with squared distances. The Star+ model is a modified version of the star model that is near linear while still differentiable. Star+ was developed as a part of the StarPlace FPGA placer [27] and applied a square-root to star cost to linearize the star model. Similar to the quadratic wire-length objective, the star+ model can handle each dimension separately. Therefore, only the x-dimension is discussed here. The center-of-gravity of a net is defined by equation 2.7.

$$x_{cl} = \frac{1}{k_l} \sum_{\forall i \in Net_l} x_i \tag{2.7}$$

where x_i represents the x-coordinate of $block_i$, the cardinality of the Net_l is represented by k_l, and the expression $i \in Net_l$ means that $block_i$ connects to Net_l. The Star+ model for a Net_l is defined by Equation 2.8:

$$S_{xl} = \gamma \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \phi} \tag{2.8}$$

The γ parameter is set to a value of 1.59 [27] to scale the estimated wirelength closer to the routed wirelength and ϕ is set to one to balance quality with routability [27]. To keep the cost

differentiable a positive constant ϕ is added as shown in Equation 2.8. The Star+ model provided in Equation 2.8 does not allow for efficient updating after a block has moved nor calculating the cost in a single pass through x. Due to these limitations of Equation 2.8, an alternative formulation that allows for constant time updates and single-pass computation is derived [27]. First, we denote the sum by $Sum_{xl} = \sum_{\forall i \in Net_l} x_i$ and the sum of squares by $SOS_{xl} = \sum_{\forall i \in Net_l} {x_i}^2$.

$$
\begin{aligned}
S_{xl} &= \gamma \sqrt{\sum_{\forall i \in Net_l} (x_i - Sum_{xl}/|Net_l|)^2 + \phi} \\
&= \gamma \sqrt{\sum_{\forall i \in Net_l} ({x_i}^2 - 2(Sum_{xl}/|Net_l|)x_i + (Sum_{xl}/|Net_l|)^2 + \phi} \\
&= \gamma \sqrt{\sum_{\forall i \in Net_l} {x_i}^2 - 2Sum_{xl}/|Net_l| \sum_{\forall i \in Net_l} x_i + \sum_{\forall i \in Net_l} (Sum_{xl}/|Net_l|)^2 + \phi} \\
&= \gamma \sqrt{SOS_{xl} - 2{Sum_{xl}}^2/|Net_l| + |Net_l|(Sum_{xl}/|Net_l|)^2 + \phi} \\
&= \gamma \sqrt{SOS_{xl} - {Sum_{xl}}^2/|Net_l| + \phi}
\end{aligned}
\tag{2.9}
$$

Sum_{xl} and SOS_{xl} can be computed in a single pass for each net. Moreover, Sum_{xl} and SOS_{xl} can be updated in constant time when a block is moved, therefore, Star+ cost can be computed in $\Theta(|Net_l|)$.

2.5.2 Star+ Jacobi iteration

The Star+ cost is differentiable when ϕ is greater than zero, hence it is suitable for analytic placement methods. The partial derivative of Star+ with respect to a block's x_j position is:

$$\begin{aligned}
\frac{\partial S_{xl}}{\partial x_j} &= \frac{\partial}{\partial x_j}\gamma\sqrt{SOS_{xl} - {Sum_{xl}}^2/|Net_l| + \phi} \\
&= \frac{\gamma}{2\gamma\sqrt{SOS_{xl} - {Sum_{xl}}^2/|Net_l| + \phi}}\frac{\partial}{\partial x_j}(SOS_{xl} - {Sum_{xl}}^2/|Net_l| + \phi) \\
&= \frac{\gamma}{2S_{xl}}\left(\frac{\partial SOS_{xl}}{\partial x_j} - \frac{\partial}{\partial x_j}{Sum_{xl}}^2/|Net_l| + \frac{\partial \phi}{\partial x_j}\right) \\
&= \frac{\gamma}{2S_{xl}}(0 + 2x_j - (2Sum_{xl})(1)/|Net_l|) \\
&= \frac{\gamma(2x_j - 2Sum_{xl}/|Net_l|)}{2S_{xl}} \\
&= \frac{\gamma(x_j - x_{cl})}{S_{xl}}
\end{aligned} \tag{2.10}$$

To find the minimum cost placement we solve for where the derivative is zero.

$$\begin{aligned}
\sum_{\forall j \in Net_l} \frac{\partial S_{xl}}{\partial x_j} &= 0 \\
\sum_{\forall j \in Net_l} \frac{\gamma(x_j - x_{cl})}{S_{xl}} &= 0 \\
\gamma \sum_{\forall j \in Net_l} \frac{x_j - x_{cl}}{S_{xl}} &= 0 \\
\sum_{\forall j \in Net_l} \frac{x_j}{S_{xl}} - \sum_{\forall j \in Net_l} \frac{x_{cl}}{S_{xl}} &= 0 \\
\sum_{\forall j \in Net_l} \frac{x_j}{S_{xl}} &= \sum_{\forall j \in Net_l} \frac{x_{cl}}{S_{xl}} \\
x_j \sum_{\forall j \in Net_l} \frac{1}{S_{xl}} &= \sum_{\forall j \in Net_l} \frac{x_{cl}}{S_{xl}} \\
x_j &= \frac{\sum_{\forall j \in Net_l} \frac{x_{cl}}{S_{xl}}}{\sum_{\forall j \in Net_l} \frac{1}{S_{xl}}}
\end{aligned} \tag{2.11}$$

To obtain the Jacobi-iteration we add iteration indices as in Equation 2.12.

$$x_j{}^{(k+1)} = \frac{\sum_{\forall j \in Net_l} \frac{x_{cl}{}^{(k)}}{S_{xl}{}^{(k)}}}{\sum_{\forall j \in Net_l} \frac{1}{S_{xl}{}^{(k)}}} \tag{2.12}$$

2.5.3 Legalization

After solving for wirelength, the placement has real-valued coordinates that (likely) violate the constraints imposed by the architecture, thus requiring legalization. StarPlace uses recursive bi-partitioning to legalize the placement. In bi-partitioning, the FPGA is recursively bisected into smaller regions and arranged in a 2-dimensional tree. Each node in the tree stores the location and size of the region, as well as a capacity for each site type. Initially, all cells belong to the root of the tree in a region that spans the entire FPGA. Cells are stored in two arrays and each array is sorted by the cell's position with one array for each dimension. Next, the array corresponding to the dimension being cut is partitioned into two. The split point for the partitions in the array must not be higher than the left-child's capacity and not lower than the total number of cells minus the right child's capacity to avoid overflow. The point in this legal range that is closest to the dividing line of the left and right child regions is selected as the split point and the cells below the split point go to the left child and the rest go to the right child. This process repeats until the leaf nodes in the tree are reached and each cell is assigned its own site.

2.6 Machine Learning in EDA

The authors in [35] discuss the basic principles of applying machine learning in *Electronic Design Automation (EDA)*. They also introduce several concepts in statistical learning and highlight that the challenges in practical implementation are often associated with feature development. In [36] authors present In-time, a machine learning compilation infrastructure approach supported by a cloud for automating the selection of parameters controlling synthesis optimizations. They show 70% reduction in final timing results across industrial benchmark problems for the Altera CAD

flow. The work in [37] presents a framework to generate a Power Distribution Network (PDN) based on machine learning. The design flow quickly predicts the total wirelength of an ASIC global route associated with a given PDN configuration to speed the search process. In [38], the authors propose a machine learning based framework for fast Sub-Resolution Assist Feature generation with high quality of results. Their experimental results show that their approach is 10x faster than a commercial Calibre tool. The authors in [39] propose a machine learning-based approach to tune FPGA design parameters. Their proposed approach finds parameter configurations that meet the timing constraints and are within 0.2% of the optimal power consumption. The work in [40] introduces an efficient placement and routing algorithm for 3D-FPGAs which employ Reinforcement Learning (RL) and Support Vector Machines (SVM) to guide local search to achieve better solutions by identifying good starting points. The approach yields better results in terms of total interconnect length and channel-width over previous techniques. In [41] the authors propose a machine learning approach based on Bayesian learning and a classification framework for generating multiple CAD parameter combinations to help attain timing closure. The authors in [42] proposed a neural network framework to estimate the total wirelength of a circuit that is mapped onto an FPGA prior to the placement and routing phase. Results obtained indicate the neural network estimation has an average error below 6% compared to VPR. The work in [43] presents a novel approach for automatic optimization of reconfigurable design parameters based on knowledge transfer techniques. This work builds on earlier work published in [44]. The proposed approach is capable of achieving up to a 35% reduction in optimization time in producing a design with similar performance compared to other optimization methods. However, the paper is not clear upon the methodology used, nor the objectives achieved. The authors in [45] build on their previous work in [42] and propose a Knowledge Based Artificial Neural Network (KBANN) that models an FPGA's architecture. The KBANN framework predicts the routing channel width required for a given circuit on various FPGA architectures. To the best of our knowledge, there is no prior work that applies machine-learning techniques to the back-end tool of FPGA CAD (Placement) for algorithm selection, nor a similar framework with the same objectives to the one

proposed in Chapter 6.

2.7 Summary

In this chapter, we introduced the architecture of modern heterogeneous FPGAs such as Xilinx's UltraScale FPGAs. The FPGA placement problem along with its objectives and constraints were stated. Different FPGA packing and placement algorithms were discussed with more emphasis on analytic placement since it is at the core of this thesis. The placement problem is computationally intractable and very difficult to solve and a means to overcome the complexity issue is to perform it in several stages including 1) global placement 2) legalization and 3) detailed placement. Several wirelength net models, along with different optimization methods, were also introduced. The long compilation time associated with simulated annealing justifies a revisit of other FPGA placement approaches. Recent literature and the ISPD contests shows that analytic placers in the ASIC domain can achieve better quality of results and runtime over other placement approaches. However, unlike ASICs, FPGAs are discrete in nature and a continuous algorithm cannot always achieve superior QoR. Thus, continuous global placement results need to be improved in order to successfully apply ASIC placement flow to FPGAs. Modern FPGA CAD flow requires a simultaneous packing and placement to achieve high quality results and avoid packing cells that are physically far apart. An early feedback from a route stage to the placement is important to address the congestion and the constraints induced from the routing architecture. Moreover, machine learning could provide some insightful prediction and evaluation of routed wirelength, and congestion to guide the FPGA placement process. In the next chapter, an overall methodology and framework is explained briefly along with the benchmarks and experimental setup used in this thesis.

Chapter 3

Overall Methodology and Tools

The main objective of this chapter is to provide the reader with an overview of the overall methodology used in this thesis. The intent is to help the reader understand the overall framework being proposed, and to identify the key components of the framework, which are presented in Chapters 4, 5 and 6 respectively. We first explain two developed routability-driven analytic placers for Ultrascale FPGAs, followed by a brief explanation of an automatic placer flow selection to improve the QoRs. The second part of this chapter will describe the experimental setup along with the benchmarks that are used to perform these experiments.

3.1 Routability-Driven Analytic Placers for Ultrascale FPGAs

Figure 3.1 shows the overall methodology framework used in this thesis. The first proposed placement tool, called GPlace-pack, is based on the original StarPlace [27]. However, the original StarPlace is a homogeneous FPGA placer that accepts packed-netlist as an input, and does not consider congestion. GPlace-pack works on the 12 ISPD'16 Contest benchmarks, which are flat technology-mapped netlists, that target a heterogeneous Xilinx's UltraScale FPGA architecture. Therefore, GPlace-pack performs both packing and placement processes and modifies the bi-partitioning legalization used in StarPlace to consider heterogeneous FPGAs. The second placement tool "GPlace-flat" is also based on the StarPlace, but with large modification of the bi-

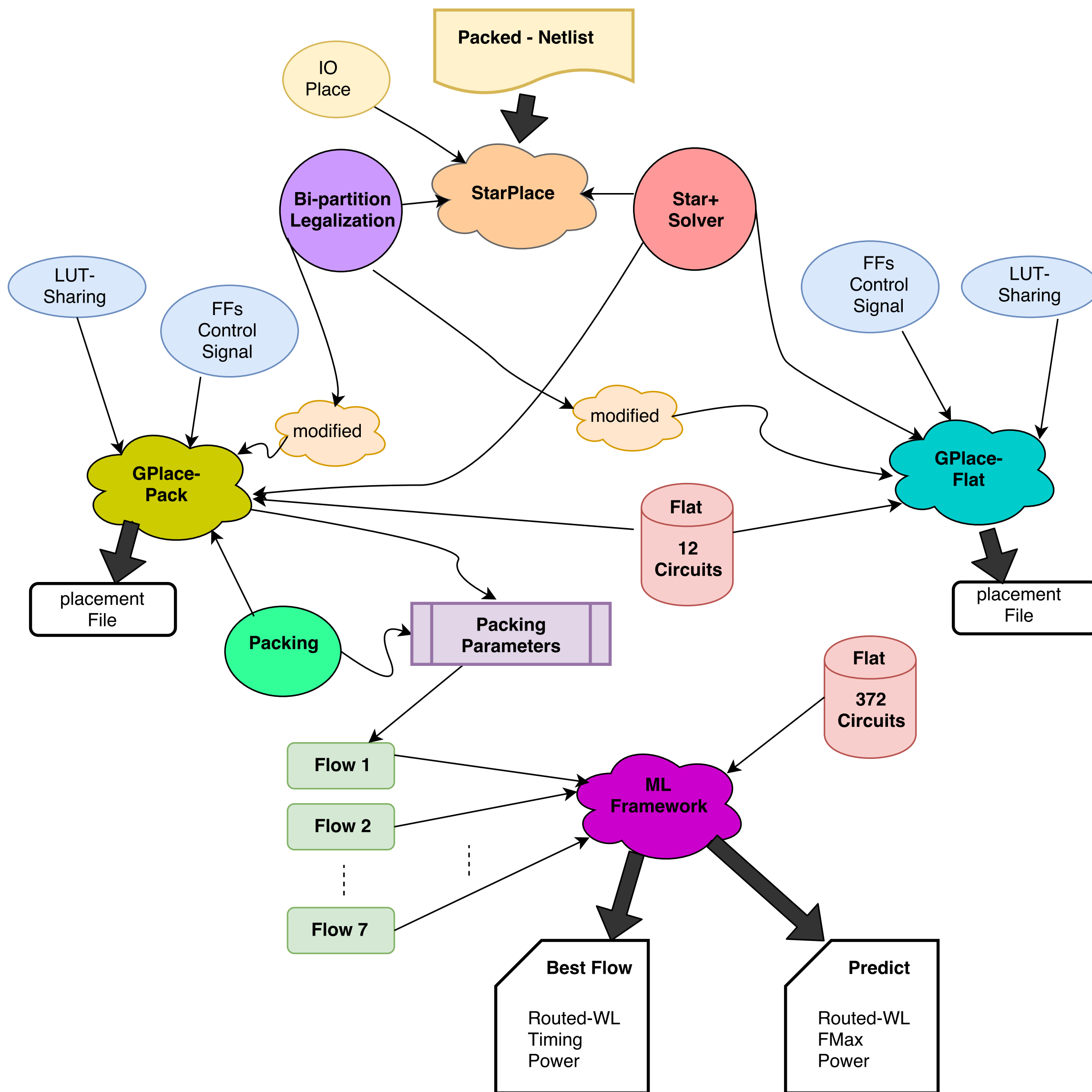


Figure 3.1: Overall Methodology of Proposed Framework

partitioning legalization to consider different capacities for different types of cells, and to handle all the hard constraints such as: LUT-sharing, and FFs control set constraints. The final module built in this thesis is a ML-based automatic flow selection that accepts 7 different flows generated from the former “GPlace-pack” tool and 372 benchmarks as inputs, and select the best flow among the 7 different flows for a given circuit based on some metrics such as: routed wirelength, timing, and power. This machine learning framework is also able to predict some useful metrics such as: routed wirelength, maximum frequency, and power for a given flow.

3.1.1 GPlace-pack

Traditionally, packing is used to alleviate congestion in FPGA flows. Traditional packing is performed as a separate step in the FPGA CAD flow before the placement stage. The packing algorithms that perform packing based on only logical connectivity could cluster cells that are physically far apart, and therefore, lead to poor wirelength and routability solution. The proposed GPlace-pack flow performs packing after flat initial placement to consider physical locations of LUTs and FFs during the packing stage to avoid packing components that are located far apart. GPlace-pack modified StarPlace [27] to consider heterogeneous FPGA architectures, like Ultrascale FPGAs, and also to avoid congestion. GPlace-pack uses the same wirelength solver as StarPlace. However, the legalization procedure uses different capacities for each cell type to roughly legalize the placement. The placement is only roughly legalized because control-set constraints and LUT sharing constraints are not considered during bi-partitioning in GPlace-pack. Instead, these constraints are solved in the packing stage. After packing, however, bi-partitioning is sufficient to legalize the packed slices. To improve congestion, we added a congestion model to estimate the routability. Packing is then adjusted to repack the whole circuit with a reduced limit on the external pin counts for each slice to reduce local congestion around the slices. The GPlace-pack placer is executed on the 12 ISPD'16 Contest benchmarks, and then the placement results produced by GPlace-pack are routed by the Xilinx Vivado detailed router. GPlace-pack is compared to the ISPD'16 Contest winners based on the following metrics: routability, routed wirelength, and placer runtime respectively.

3.1.2 GPlace-flat

To avoid issues associated with packing that can sometimes restrict the placement solution space, an analytic flat placer called GPlace-flat is next proposed. GPlace-flat handles FF control-set constraints, as well as LUT sharing during global placement. This eliminates the need for a packing step in the FPGA flow, and allows the placer to optimize globally while enforcing legalization constraints. The GPlace-flat flow consists of three main phases. Phase I seeks to produce a high-

quality wirelength placement. This is achieved by performing a pin-propagation pre-placement to place cells close to their IOs. Then, several iterations of analytic flat placement are performed to globally optimize for wirelength while enforcing all legalization constraints. WL-driven global placement in GPlace-flat is based on the Star+ [27] wirelength model, with extended window-based bi-partitioning legalization to support heterogeneous architectures. The generated placement satisfies all hard constraints and is optimized for wirelength, but still does not, yet, consider congestion. In Phase II, efficient global routing is performed to produce an accurate congestion map, and LUTs are inflated proportional to their number of pins and the congestion values of their tiles. Since GPlace-flat estimates congestion and inflates cells only once, having an accurate congestion map is crucial to proposed flow. The congestion-driven global placement shares the same framework with the WL-driven global placement except that inflated LUTs are bi-partitioned based on their densities. In Phase III, detailed placement is performed using Dual Objective Independent Set Matching (DOISM) to further optimize for both wirelength and external pin count (to improve routability). GPlace-flat is evaluated on the 12 ISPD'16 Contest benchmarks based on the following metrics as: routability, routed wirelength, and placer runtime respectively. As well, an additional 360 benchmarks were provided directly from Xilinx Inc. These benchmarks were used to compare GPlace-flat to the most recently improved versions of the first and second place ISPD'16 contest winners. Subsequent experimental results show that GPlace-flat is able to outperform the improved placers in a variety of areas including the number of best solutions found, fewest number of benchmarks that cannot be routed, runtime required to perform placement, and runtime required to perform routing.

3.1.3 ML Automatic Flow Selection

While the literature has produced many different placement flows, there is no single flow that exhibits superior performance on all possible circuits. Due to variations in circuit characteristics and FPGA architectures, as well as the different optimization strategies employed by different placers, flows that perform well on some circuits may perform poorly on others. We present a general

machine-learning framework that seeks to address the disconnect between different stages of the FPGA CAD flow that adversely affect the quality of results of the implemented designs. The framework consists of a suite of techniques (Artificial Neural Network, Decision Tree, Support Vector Machine, K Nearest Neighbor, and Random Forest) to model the underlying relationship between the characteristics of circuits and the best CAD algorithm (and parameters) to use for obtaining an optimized implementation on an FPGA. The efficacy of the framework is demonstrated by training the framework to choose between different FPGA placement flows, and also by training the framework to predict various quality metrics related to FPGA placement. The proposed ML framework is evaluated on 372 benchmarks provided directly from Xilinx Inc., and seven academic placement flows based on the award-winning placer in [9]. First, the ML framework predicts the best placer of the seven placers flows based on four different objectives: minimizing estimated wirelength, routed wirelength, critical-path delay, and power. Second, we demonstrate the flexibility of the proposed framework by also using it to efficiently and accurately predict various quality metrics without performing placement and routing, like estimated wirelength, routed wirelength, critical-path delay, power, post-routing column and row utilization, and a circuit's Rent exponent.

3.2 Experimental Design

3.2.1 Benchmarks

All frameworks were tested using all 12 ISPD 2016 contest benchmarks [8] shown in Table 3.1. Frameworks were also tested using an additional 360 benchmarks provided directly by Xilinx Inc. These benchmarks were created using a netlist-generation tool, Gnl [46]. During synthesis, key circuit features were varied among the benchmarks including, the numbers of LUTs, FFs, DSPs, BRAMS, and IOs. Also, different Rent exponents were used to vary interconnection complexity, and the number of resets were varied to pose significant challenges during placement. Table 3.2 shows the range of the previous circuit parameters. Each benchmark is described using the Bookshelf format [8].

3.2.2 Experimental Setup

The main advantage of using the benchmarks listed in Table 3.1 is that they facilitate a direct comparison with other state-of-the-art placement flows that use the Bookshelf format listed in Table 3.3.

Table 3.1: ISPD 2016 Placement Contest Benchmark Statistics

Benchmark	#LUTs	#FF	#BRAM	#DSP	#CSet	#IO	R.E
FPGA-1	49K	55K	0	0	12	150	0.4
FPGA-2	98K	74K	100	100	121	150	0.4
FPGA-3	245K	170K	600	500	1281	400	0.6
FPGA-4	245K	172K	600	500	1281	400	0.7
FPGA-5	246K	174K	600	500	1281	400	0.8
FPGA-6	345K	352K	1000	600	2541	600	0.6
FPGA-7	344K	357K	1000	600	2541	600	0.7
FPGA-8	485K	216K	600	500	1281	400	0.7
FPGA-9	486K	366K	1000	600	2541	600	0.7
FPGA-10	346K	600K	1000	600	2541	600	0.6
FPGA-11	467K	363K	1000	400	2091	600	0.7
FPGA-12	488K	602K	600	500	1281	400	0.6

Table 3.2: Range of Key Circuit Features for 372 Benchmarks [47]

#LUTs	#FF	#BRAM	#DSP	#CSet	#IO	R.E
44K-518K	52K-630K	0-1035	0-620	11-2684	150-600	0.4-0.8

Table 3.3: List of Academic Placers for Ultrascale FPGAs

Placer	ISPD 2016 Contest	ICCAD 2016	Current Revision
uoguelph	GPlace-Pack	GPlace-Pack [9]	GPlace-Flat
utexas	UTPlaceF 1.0	UTPlaceF 2.0 [4]	UTPlaceF 3.0 [5]
CUHK	RippleF 1.0	RippleF 2.0 [10]	

All of the placers are compared based on two metrics: solution quality in terms of routed wirelength, and absolute runtime (in seconds). All placement algorithms ran on an Intel (Xeon CPU E3-1270 v5 @ 3.60GHz) processor equipped with 16GB memory. The algorithms were developed using C and compiled using gcc 4.4.7 (Red Hat 4.4.7-18) compiler. The routing is

performed by Xilinx Vivado v2015.4 with a patch applied to make it compatible with the modified bookshelf benchmarks format of the ISPD 2016 placement contest. The target device is xcvu95, part of the Virtex Ultrascale family. The device's aspect ratio and the average spacing between blocks were determined based on the Xilinx Ultrascale VU095 architecture, the latest 20nm FPGA chip. All placers were executed with a single thread and Vivado Xilinx was set to be in the same configurations as in the ISPD 2016 Placement Contest, which limits the running time to 12 hours.

3.3 Summary

In this chapter we introduced the overall methodology flow of the proposed work along with the benchmarks and experimental setup used throughout this thesis. The methodology flow consists of three phases: Phase I deals with developing a new congestion-aware FPGA placer tool for Xilinx's Ultrascale FPGA architectures. In Phase II a novel flat analytic placer, that seeks to optimize both wirelength and routability simultaneously, is proposed for Xilinx's Ultrascale FPGA architectures. Finally, Phase III proposes a novel technique based on machine learning to predict metrics that can be useful to facilitate the placement process. This chapter also described the benchmarks and the experimental setup that are used throughout this thesis.

Chapter 4

GPlace-pack

In this Chapter, we describe the design and the development of GPlace-pack. This placer placed third in the ISPD 2016 FPGA Design contest held in California USA. GPlace-pack employs the same wirelength model used in StarPlace [27]. However, the legalization procedure was modified to use different capacities for each cell type. The proposed placer is evaluated using the ISPD'16 Contest benchmarks and compared with the first and second winners in routed wirelength and runtime. Finally, we discuss the differences and highlight the limitations of the top three ISPD'16 Contest winner placers.

4.1 GPlace-pack Flow

Targeted at homogeneous, island-style FPGAs, StarPlace [27] is not directly applicable to the UltraScale architecture. Therefore, it had to be modified and adapted for the heterogeneous UltraScale architecture presented in Chapter 2. The resulting flow, called GPlace-pack, is summarized in Figure 4.1. Algorithm 1 shows the pseudo code of the GPlace-pack placer. GPlace-pack uses the same wirelength solver as StarPlace, but the legalization procedure was modified to use different capacities for each cell type to roughly legalize the placement. The placement is only roughly legalized because control-set constraints and LUT sharing constraints are not considered during bi-partitioning in GPlace-pack. Instead, these constraints are solved in the packing stage.

ALGORITHM 1: GPlace-pack algorithm

```
1: Convert net-list to graph using Star+ net model
2: Pin Propogation Pre-placement
3: repeat
4:    K ← 2√|Cells|          //No. of iter K is set to 2 × √number of movable cells
5:    while K ≥ 1 do
6:       Perform ⌊K⌋ Wirelength Optimization Iterations
7:       Rough Bi-Partition Legalization
8:       K ← 0.8K
9:    end while
10:   Set Congestion_th           //Set the congestion threshold to some value
11:   repeat
12:      Placement-aware CLB Packing   //Pack based on affinity and placement info.
13:      K ← 2√|Sites|
14:      while K ≥ 1 do
15:         Perform ⌊K⌋ Wirelength Optimization Iterations
16:         Bi-Partition Legalization
17:         K ← 0.8K
18:      end while
19:      Measure Congestion            //apply WLPA congestion estimator
20:      congestion ← max Congestion
21:      loosen packing parameters   //change the packing parameters (i.e., no. of externals)
22:   until congestion < Congestion_th
23: until congestion < Congestion_th
24: Local Refinement
```

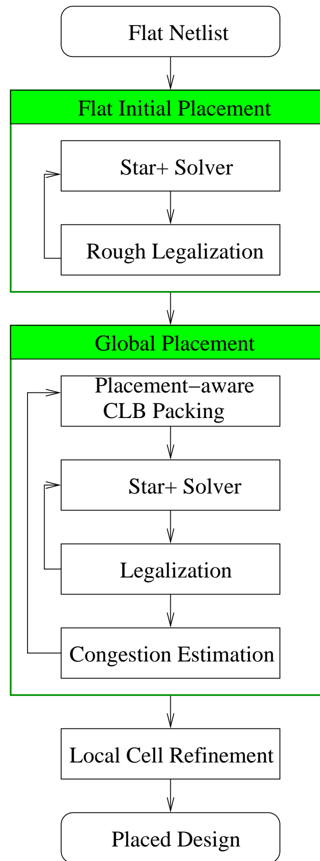


Figure 4.1: GPlace-pack Flow

After packing, however, bi-partitioning is sufficient to legalize the packed slices. The GPlace-pack packing algorithm was inspired by the flow of [48]. To improve congestion, we added a congestion model to estimate the routability, then adjusted packing to repack the whole circuit with a reduced limit on the external pin counts for each slice to reduce local congestion around the slices.

4.1.1 Pin Propagation Pre-Placement

Initially, a placement is generated by propagating the positions of fixed I/O cells through the net-list at line 2 of Algorithm 1. First, the net-list is converted to a graph by creating a directed edge from the driver of the net to each sink in the net. The cells are iterated in topological order and moved to the average position of their inputs, placing cells near the input pads they are most connected to. The entire process is repeated, but with the direction of the edges reversed, thereby placing

cells near their ultimate outputs. Finally, the results of propagating from inputs and outputs are averaged.

4.1.2 Flat Initial Placement

In flat placement step from line 4 to line 9 of Algorithm 1, GPlace-pack uses the same wirelength solver as StarPlace [27], but the legalization procedure is modified to use different capacities for each cell type to roughly legalize the placement. The placement is only roughly legalized because control-set constraints and LUT sharing constraints are not considered during bi-partitioning in GPlace-pack. Both the Star+ model and the legalization procedure are described in the subsections that follow.

4.1.2.1 Wirelength Optimization

In the Star+ model, a net is converted to a star by adding a centre-of-gravity node at the mean position (c_{yl}, c_{xl}) of the cells in the net by Equation 4.1. The Star+ cost is computed for each net and dimension in Equation 4.2. Equation 4.3 updates the x-position of cell i at iteration k using the Jacobi method at line 6 of Algorithm 1.

$$c_{xl} = \frac{1}{|\mathrm{Net}_l|} \sum_{i \in \mathrm{Net}_l} x_i \tag{4.1}$$

$$S_{xl} = \sqrt{1 + \sum_{i \in \mathrm{Net}_l} (x_i - c_{xl})^2} \tag{4.2}$$

$$x_i^{(k+1)} = \sum_{l: i \in \mathrm{Net}_l} \frac{c_{xl}^{(k)}}{S_{xl}^{(k)}} \Bigg/ \sum_{l: i \in \mathrm{Net}_l} \frac{1}{S_{xl}^{(k)}} \tag{4.3}$$

4.1.2.2 Legalization

After solving for wirelength, the placement has real-valued coordinates that (likely) violate the constraints imposed by the architecture, thus requiring legalization. StarPlace uses recursive bi-

partitioning to legalize the placement. In bi-partitioning at line 7 of Algorithm 1, the FPGA is recursively bisected into smaller regions and arranged in a 2-dimensional tree. Each node in the tree stores the location and size of the region, as well as a capacity for each site type. Initially, all cells belong to the root of the tree in a region that spans the entire FPGA. Cells are stored in two arrays and each array is sorted by the cell's position with one array for each dimension. Next, the array corresponding to the dimension being cut is partitioned into two. The split point for the partitions in the array must not be higher than the left-child's capacity and not lower than the total number of cells minus the right child's capacity to avoid overflow. The point in this legal range that is closest to the dividing line of the left and right child regions is selected as the split point and the cells below the split point go to the left child and the rest go to the right child. This process repeats until the leaf nodes in the tree are reached and each cell is assigned its own site.

4.1.2.3 Solver Schedule

GPlace-pack does not measure convergence, instead the number of solver iterations is computed a priori using the number of placeable elements in the circuit. Algorithm 2 shows the schedule used in GPlace-pack where the k value controls the number of solver iterations. Initially K is equal to two times the square root of the number of blocks in the netlist and is repeatedly decreased to 80% of its previous value when the placement is legalized, thereby limiting the amount of change the solver can introduce as the algorithm progresses. GPlace-pack solves and legalizes each cell type

ALGORITHM 2: Jacobi Solver Schedule algorithm

1: $K \leftarrow 2\sqrt{|\text{Cells}|}$ *//No. of iter. K is set based on the no. of movalbe cells*
2: **while** $K \geq 1$ **do**
3: Perform K Round Robin Iterations *//explained in Algorithm 3*
4: $K \leftarrow 0.8K$
5: **end while**

in a round robin fashion as shown in Algorithm 3. First, k Star+ jacobi iterations are performed for all types, followed by rough legalization of the placement with different capacities for each cell type as in lines [1, 2] of Algorithm 3. The placement is only roughly legalized because control-set

constraints and LUT sharing constraints are not considered during bi-partitioning in GPlace-pack. In lines [3, 4] of Algorithm 3, only cells of slice type (LUTs, FFs) are solved and then legalized. Finally, a non-slice type cells (BRAMs, DSPs) are solved and legalized.

ALGORITHM 3: Round Robin Shcedule algorithm

1: Perform K Wirelength Jacobi Iterations for all Cells;
2: Rough Bi-Partition Legalization;
3: Perform K Wirelength Jacobi Iterations for Slice types (LUTs, FFs);
4: Rough Bi-Partition Legalization;
5: Perform K Wirelength Jacobi Iterations for non-Slice types (BRAMs, DSPs);
6: Rough Bi-Partition Legalization;

4.1.3 Placement-aware Packing

The output of the previous flat initial placement step is used to inform packing with the physical information of the cells to guide packing. At line 12 of Algorithm 1 packing begins by dividing the FPGA into a regular grid. The size of each bin in the grid is initially (1×1) but can grow up to (12×12) if the placement remains congested. Packing proceeds in each bin by selecting an arbitrary cell as a seed to start the cluster. Only cells, that are located in the same bin of the seed cell, will be considered for clustering. A cell becomes a candidate if adding it to the cluster does not violate any capacity, control-set, or LUT-sharing constraints, and does not result in too many external pins. The maximum number of allowable external pins is initially high (i.e., 104) and is decreased later if the congestion is too high, before repacking. Next, the candidates are ranked by their affinity to the current cluster and the candidate with the highest affinity is added to the cluster. The affinity is computed as the number of edges between the cluster and the candidate as shown in Algorithm 4, treating a net as a clique. Candidates are added in this way until there are no more suitable candidates. From the remaining unclustered cells, a new seed is selected arbitrarily and the clustering repeats until there are no more seeds. Next, the wirelength optimization and legalization are applied to the packed net-list.

ALGORITHM 4: Affinity Calculation algorithm

```
1: affinity ← 0
2: for all net that is common to both clusters do
3:     if |net| = 2 then
4:         affinity ← affinity + 3            //add weight of 3 if it is 2-pin net
5:     else if |net| < 10 then
6:         ipin ← |net ∩ C_i|                 //number of pins intersect with Cluster C_i
7:         jpin ← |net ∩ C_j|                 //number of pins intersect with Cluster C_j
8:         affinity ← affinity + ipin · jpin
9:     end if
10: end for
11: return affinity
```

4.1.4 Congestion Estimation

In line 19 of Algorithm 1, GPlace-pack estimates congestion using the Wire-Length-Per Area (WLPA) model with the Weighted Half-Perimeter Wire-Length (WHPWL) estimate from VPR [23]. WLPA is used to estimate congestion because it can be efficiently computed and can accurately identify areas of high congestion (i.e., hot spots).

To further improve the accuracy, the bounding box for each net is converted from slice coordinates to switch coordinates. The vertical and horizontal WLPA ($\text{WLPA}_x, \text{WLPA}_y$) are computed by Equation 4.4–4.5.

$$\text{WLPA}_{xl} = q(|\text{Net}|)/h_l \quad (4.4)$$

$$\text{WLPA}_{yl} = q(|\text{Net}|)/w_l \quad (4.5)$$

where q is the compensation function from VPR and w_l, h_l are the width and height of the bounding box, respectively.

4.1.5 Computing Switch Utilization

A novel and efficient computation of WLPA that is independent of the sizes of the nets bounding boxes is proposed and explained in this section. GPlace-pack computes the switch utilization

using prefix sums. The benefit to this approach is that the computation time is independent of the quality of placement and can, therefore, be applied early in the placement process. Normally the computation of WLPA involves iterating over the bounding boxes of all nets in the netlist. Using this naïve method for a netlist with a total of P pins and a $W \times H$ sized FPGA results in a complexity of $\mathcal{O}(PWH)$, while the prefix-sum method outlined here has the complexity $\Theta(P + WH)$.

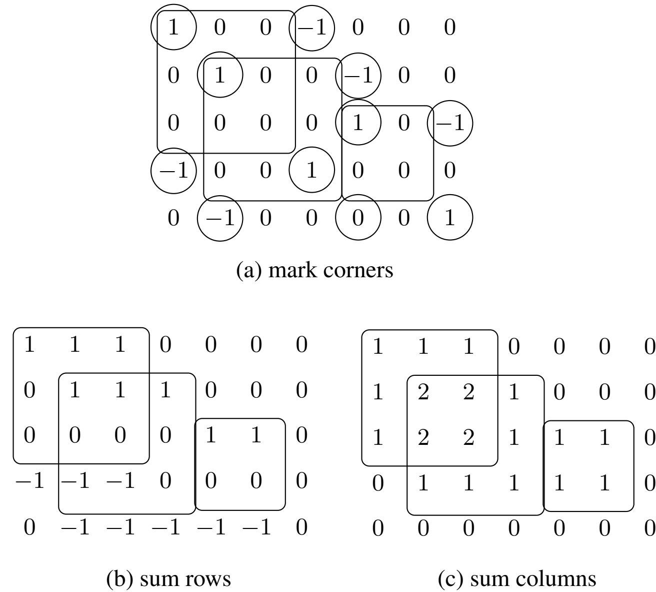


Figure 4.2: Computing WLPA with Prefix Sums (WLPA=1)

The steps to computing the switch utilization are outlined and demonstrated in Figure 4.2. First, the bounding box for each net is computed, converted to switch coordinates, and then the WLPA is estimated. For simplicity, Figure 4.2 assumes the WLPA is one for all nets. Next, a matrix of zeros is allocated with the same size as the FPGA. For each net, the WLPA value is added to the matrix near the corners of the bounding box. More specifically, for a bounding box, (x_1, y_1, x_2, y_2), the WLPA value is added at (x_1, y_1) and $(x_2 + 1, y_2 + 1)$, while the WLPA is subtracted at $(x_2 + 1, y_1)$ and $(x_1, y_2 + 1)$. Next, an inclusive prefix-sum is performed on each row of the matrix, and then for each column. The resulting matrix contains the total wirelength estimate of each switch, and is converted to a utilization estimate by dividing by the channel width. Both the horizontal and vertical switch utilization are computed using prefix sums.

4.1.6 Local Refinement

In line 24 of Algorithm 1, Local refinement performs a single iteration of wirelength optimization to find a more ideal, but likely illegal, position for each cell. Next, local refinement attempts to move each cell from its current legal position to the slice closest to its ideal position, provided the move will not violate any slice-constraints. This process is repeated for several iterations.

4.2 ISPD 2016 Contest Winning Placers

Since 2005, ISPD has held many different contests on topics covering placement, global routing, gate sizing and clock tree synthesis on ASIC design tools. The ISPD 2016 contest was the first contest on FPGA CAD tools, covering challenging topics related to FPGA CAD ”Routability-driven FPGA Placement”. The contest benchmarks are based on an industry leading 20nm technology Vertex UltraScale device, and reflect typical modern, high-end FPGA designs [8]. The contest evaluation metrics include wirelength, routability, and runtime. The flows of the top three winning placers of the ISPD FPGA Placement Contest are shown in Fig. 4.3. Figures 4.3 (a), (b) show the flows for the first and second place contest winners, respectively: UTPlaceF ICCAD16 [4] and RippleFPGA ICCAD16 [10]. Figure 4.3 (c) shows our previous flow, GPlace-pack, which placed third in the contest. Table 4.1 summarizes the main differences between the contest win-

Table 4.1: ISPD 2016 Placement Contest Winners Flows Comparisons

	FPGA Placer		
Placement Module	UTPlaceF ICCAD16 [4]	RippleFPGA ICCAD16 [10]	GPlace-pack
AP Net-length Model	B2B	B2B	Star+
Congestion Estimation	Adaptive ASIC Global Router	WLPA	WLPA
Congestion Removal	Congestion-aware Packing	Cell Inflation	Re-Packing
Global Placement target	CLB Packed Netlist	BLE Packed Netlist	CLB Packed Netlist
Detailed Placement	Congestion-aware ISM	Optimal region Greedy Moves	Local Greedy Moves

ners. UTPlaceF ICCAD16 and RippleFPGA ICCAD16 both use an *Analytical Placement (AP) Bound-to-Bound (B2B)* [33] net length model, while GPlace-pack employs the same wirelength model used in StarPlace [27], a near-linear model, called Star+. To estimate congestion during the global placement, RippleFPGA ICCAD16 and GPlace-pack use a probabilistic method based

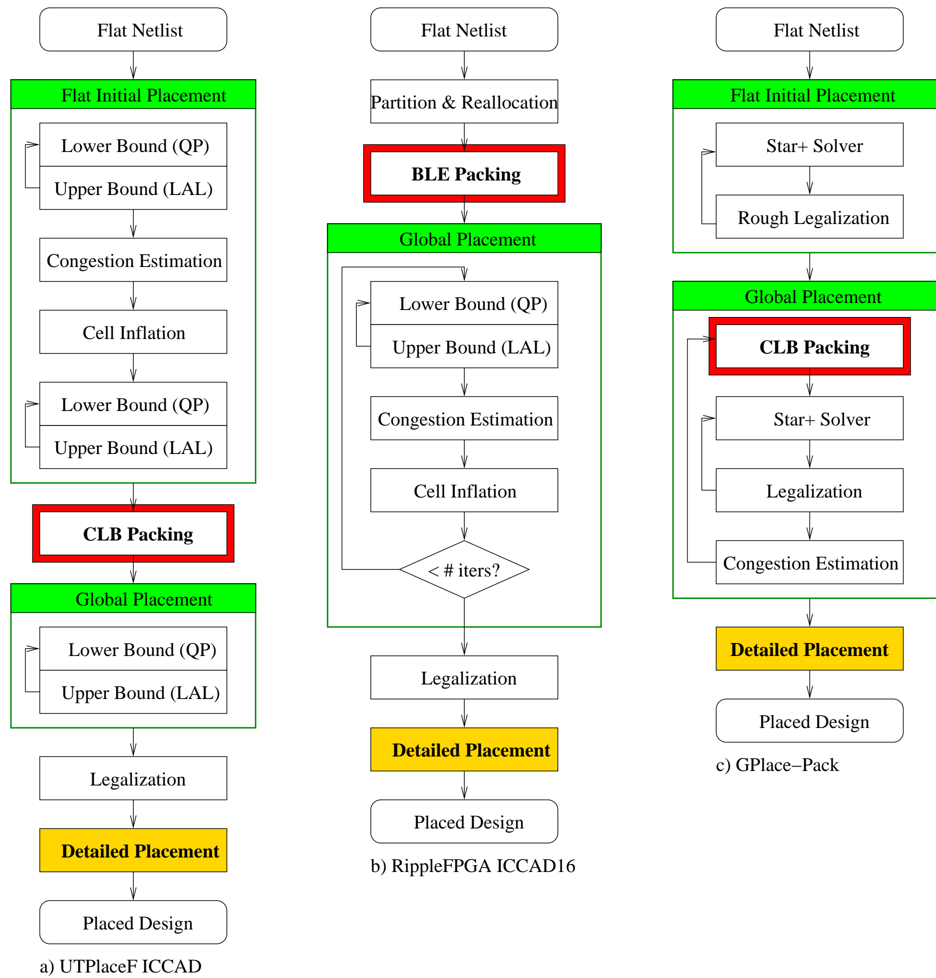


Figure 4.3: Proposed Flows of ISPD 2016 FPGA Contest Winners

on *Wirelength Per Area (WLPA)* [9, 10], whereas UTPlaceF ICCAD16 uses an ASIC global router (NCTUgr2.0) [49]. Since the NCTUgr2.0 global router works on a grid graph and does not handle the different segment lengths present in modern FPGA architectures, UTPlaceF ICCAD16 experimentally adjusts channel widths in the grid graph to produce a more accurate congestion map.

In the global placement stage, UTPlaceF ICCAD16, RippleFPGA ICCAD16, and GPlace-pack optimize wirelength for packed netlists. Both UTPlaceF ICCAD16 and GPlace-pack rely on packing to reduce the routing demand and hence congestion, while RippleFPGA ICCAD16 uses BLE bloating to reduce the congestion. Traditionally, alleviating congestion in FPGAs is addressed by packing to produce a placement-friendly netlist. However, packing approaches may unnecessarily restrict the placement solution space leading to degraded wirelength and possibly

unroutable placements.

During the final detailed placement stage, UTPlaceF ICCAD16 further minimizes *Half-Perimeter Wirelength (HPWL)* using an *Independent Set Matching (ISM)* approach [4], while RippleFPGA ICCAD16 uses the *Optimal Region Greedy Moves (ORGM)* to optimize HPWL. GPlace-pack uses local greedy refinements to further optimize the Star+ cost. Below, we summarize some key issues in existing placers that motivate our work in the next Chapter.

- Logical packing algorithms, such as those proposed in [2, 3], perform packing based only on logical connectivity which could cluster cells that are physically far apart. This leads to wirelength-unfriendly netlists that may degrade routability.

- Placement-aware packing techniques (e.g., UTPlaceF ICCAD16, and GPlace-pack), which consider physical locations of cells during packing, may still produce netlists that are wirelength-unfriendly. These placers perform packing based on physical locations generated by first performing a *Flat Initial Placement (FIP)*. The FIP does not consider control set constraints associated with the FFs. However, designs with large numbers of control sets (e.g., clocks, resets and control enables) are common in modern high-end FPGAs. This reduces the logic utilization and hence forces the packing method to pack FFs that are far apart to fit into the FPGA area, possibly degrading wirelength and routability. This issue will be discussed in the experimental results section, where we compare UTPlaceF ICCAD16 and GPlace-flat.

- Even though RippleFPGA ICCAD16 does not fully pack the netlist, its global placement phase does not handle control set constraints appropriately. This leads to a mismatch between the results obtained from global placement and the final legalized results when designs have large numbers of control sets and FFs. Therefore, it is of particular importance to handle these control set constraints during global placement.

Based on the limitations of the current flows discussed above, and the results that will be presented in Chapter 5, there seems to be no clear view to either pack or avoid packing for the ISPD2016 benchmarks that will be placed on the UltraScale Xilinx architecture. Furthermore,

experiments conducted on an additional 360 benchmarks also show that no one flow is capable of producing routable results for all benchmarks. Therefore, our objective in the next chapter is not only to develop a placer that is capable of producing better *Quality-of-Result (QoR)* in terms of routed wirelength for most of the benchmarks compared to current state-of-the-art placers, but one that is also capable of increasing the chance of routability and reducing the effort of the router to produce solutions in a reasonable time.

4.3 Preliminary Results of GPlace-pack

Based on the results presented later in Section 4.3.1, it was observed that GPlace-pack may unpack and repack the entire net-list if the placement is congested. This global repacking leads to a rapid increase in wirelength for highly-congested designs, and an increase in the number of slices used in the FPGA. Consequently, a circuit may not fit on the FPGA if packed too loosely. Furthermore, any rapid increase in wirelength may increase congestion further, thus reducing routability.

4.3.1 Routing Results

To test GPlace-pack, the 12 benchmarks from the 2016 ISPD placement contest [8] described in Table 3.1 are placed and routed. The routed wirelength for each benchmark is provided in Table 4.2. Table 4.2 shows the results of GPlace-pack compared to the ISPD'16 Contest winners on routed wirelength and runtime, where on average GPlace-pack has 33% wirelength increase over the first placer, and GPlace-pack is 51% slower than the first placer.

Figure 4.4 shows the convergence of the Star+ wirelength model during different phases of the GPlace-pack flow. Flat initial placement does not consider hard constraints such as FFs control set constraints, and LUT-sharing constraints. Therefore, a big jump of the Star+ wires occurs at iteration 30, when CLB packing is performed to legalize the placement solution. Although, global placement iterations optimizes for Star+ wirelength in a packed netlist, the Star+ cost is still high. From the results presented in Table 4.3.1 and the Star+ convergence shown in Figure 4.4,

Table 4.2: Comparison with ISPD'16 Contest Winners on ISPD 2016 Benchmark Suite

Benchmark	1st Place (UTPlaceF1.0)		2nd Place (RippleF1.0)		3rd Place (GPlace-pack)	
	Routed WL	Runtime(s)	Routed WL	Runtime(s)	Routed WL	Runtime(s)
FPGA-1	PE[a]	-	379932	118	581975	97
FPGA-2	677877	435	679878	208	1046859	191
FPGA-3	3223042	1527	3660659	1159	5029157	862
FPGA-4	5628519	1257	6497023	1149	7247233	889
FPGA-5	10264769	1266	UR	-	UR	-
FPGA-6	6630179	2920	7008525	4166	6822707	8613
FPGA-7	10236827	2703	10415871	4572	10973376	9169
FPGA-8	8384338	2645	8986361	2942	12299898	2741
FPGA-9	UR[b]	-	13908997	5833	UR	-
FPGA-10	PE	-	PE	-	UR	-
FPGA-11	11091383	3227	11713479	7331	UR	-
FPGA-12	9021769	4539	PE	-	UR	-
Norm.	**+0%**	**1.00X**	**+7%**	**1.27X**	**+33%**	**1.51X**

[a]PE: Placement error

[b]UR: Unroutable placement

it seems that GPlace-pack suffers from several issues that lead to poor wirelength and congestion optimizations. These issues can be summarized as the following:

- Seed-based clustering used in the packing stage is not efficient since it is heavily dependent on the order of the cells to pick as a seed.
- When congestion is detected, global repacking of the entire netlist is performed with different packing parameters. This global repacking increases the wirelength where it is not necessary and hence worsens the congestion.
- The maximum congestion value is not a descriptive metric for congestion.
- The local refinement stage is un-aware of congestion issues, and performs local greedy moves that can become trapped quickly in the local minimum.

To overcome those issues in the GPlace-pack flow a new placer called GPlace-flat is proposed in the next chapter. The proposed flat analytic placer, GPlace-flat, handles FF control-set constraints, as well as LUT sharing during global placement. This eliminates the need for a packing step in the FPGA flow, and allows the placer to optimize globally while enforcing legalization constraints. A global router was developed for use in GPlace-flat that accurately estimates congestion for UltraScale FPGA devices. An enhanced cell inflation technique is also proposed to spread

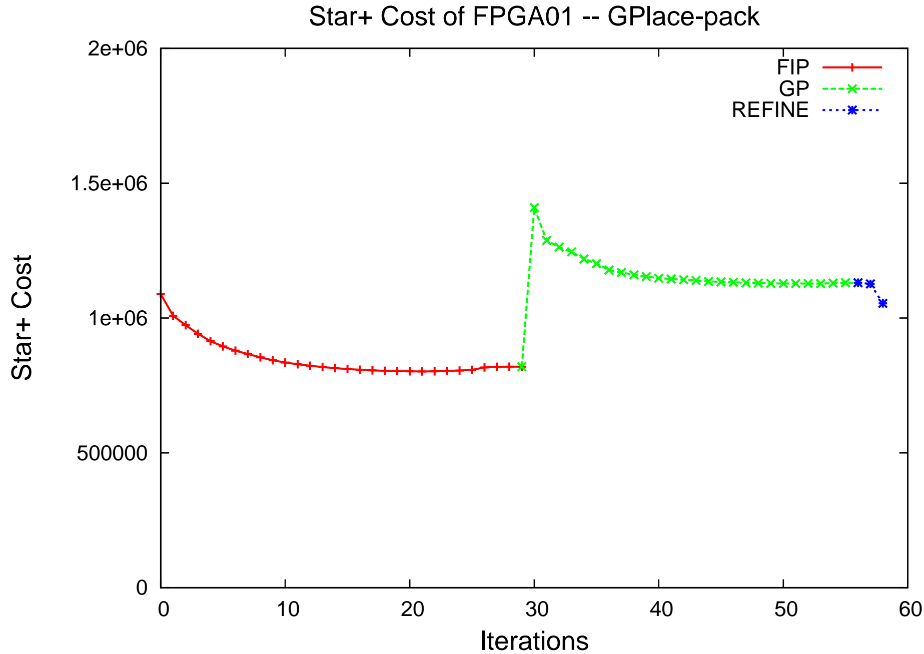


Figure 4.4: Convergence of Star+ Cost during GPlace-pack Phases

cells in congested regions to reduce congestion. Finally, a novel detailed placement stage is added to further improve wirelength. Since GPlace-flat targets flat netlists (i.e., it does not perform any kind of packing), a Dual Objective Independent Set Matching (DOISM) detailed placement is proposed to further improve both wirelength and routability. DOISM alternates between minimizing wirelength and minimizing external pin count. In Chapter 5, we introduce our methodology for developing GPlace-flat, and highlight the effectiveness of the flow.

4.4 Summary

In this chapter, we presented a new congestion-aware placement tool for Xilinx's UltraScale architectures called GPlace-pack. GPlace-pack participated in the ISPD 2016 Routability-driven Placement Contest for FPGAs, where it placed third. A discussion of the top three ISPD'16 Contest winning placers along with their limitations is introduced. In the next chapter, we will first attempt to address these limitations by proposing several novel techniques that overcome the draw-

backs. The proposed techniques implemented in Chapter 5 (GPlace-flat) are capable of efficiently resolving congestion, thus improving routability and reducing wirelength. Moreover, another important objective in this work is to develop a placer that is also capable of increasing the chance of routability and reducing the effort of the router to produce solutions in reasonable time.

Chapter 5

GPlace-flat

In this chapter, a flat analytic placer called GPlace-flat is developed to avoid issues associated with packing that can sometimes restrict the placement solution space. The proposed GPlace-flat, handles FF control-set constraints, as well as LUT sharing during global placement. This eliminates the need for a packing step in the FPGA flow, and allows the placer to optimize globally while enforcing legalization constraints. A global router was developed for use in GPlace-flat that accurately estimates congestion for Xilinx UltraScale FPGA devices. An enhanced cell inflation technique is also proposed to spread cells in congested regions to reduce congestion. Finally, a novel detailed placement stage is added to further improve wirelength. Since GPlace-flat targets flat netlists (i.e., it does not perform any kind of packing), a *Dual Objective Independent Set Matching (DOISM)* detailed placement is proposed to further improve both wirelength and routability. DOISM alternates between minimizing wirelength and minimizing external pin count (note: external pins are exposed pins that connect different Slices, hence, they do not include pins within the same Slice).

5.1 GPlace-flat Flow

Figure 5.1 shows the flow of GPlace-flat. The overall flow consists of three main phases. Phase I seeks to produce a high-quality wirelength placement. This is achieved by performing a pin-

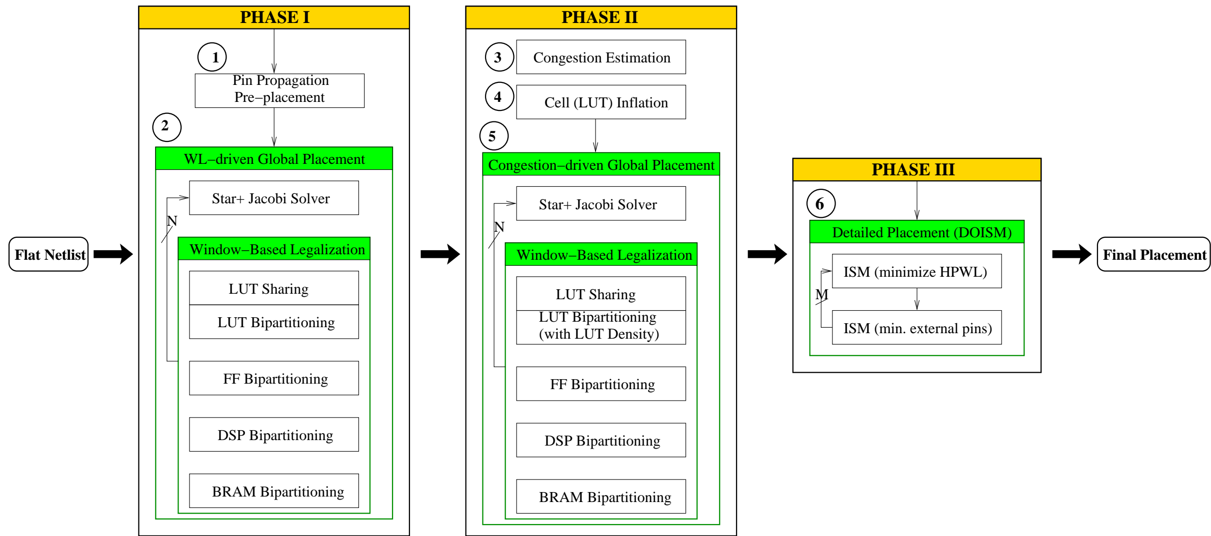


Figure 5.1: GPlace-flat Flow

propagation pre-placement (Fig. 5.1: step (1)) to place cells close to their IOs. Then, several iterations of analytic flat placement are performed to globally optimize for wirelength while enforcing all legalization constraints. WL-driven global placement (Fig. 5.1: step (2)) in GPlace-flat is based on the Star+ [27] wirelength model, with extended window-based bi-partitioning legalization to support heterogeneous architectures. The generated placement satisfies all hard constraints and is optimized for wirelength, but still does not, yet, consider congestion. In Phase II, efficient global routing is performed to produce an accurate congestion map (Fig. 5.1: step (3)) and LUTs are inflated (Fig. 5.1: step (4)) proportional to their number of pins and the congestion values of their tiles. Since GPlace-flat estimates congestion and inflates cells only once, having an accurate congestion map is crucial to our flow. The congestion-driven global placement (Fig. 5.1: step (5)) shares the same framework with the WL-driven global placement except that inflated LUTs are bi-partitioned based on their densities. In Phase III, detailed placement is performed using DOISM (Fig. 5.1: step (6)) to further optimize for both wirelength and external pin count (to improve routability). In the following subsections each component of the GPlace-flat framework is presented in detail, and the main merits of adding each component to the flow is discussed.

5.1.1 [Phase I] Pin Propagation Pre-Placement

An initial placement is generated by propagating the positions of fixed I/O cells through the netlist. First, the netlist is converted to a graph by creating a directed edge from the driver of the net to each sink in the net. The cells are iterated in topological order and moved to the average position of their inputs, placing cells near the input pads they are most connected to. The entire process is repeated, but with the direction of the edges reversed, thereby placing cells near their ultimate outputs. Finally, the results of propagating from inputs and outputs are averaged and cells placed accordingly. Based on our previous work [27], Star+ analytic technique can benefit from pre-placement; thus producing better results than random initial pre-placement.

5.1.2 [Phase I] WL-driven Global Placement

WL-driven global placement in GPlace-flat employs the Star+ [27] wirelength model. However, after solving the resulting equation system for wirelength, the placement has real-valued coordinates that (likely) violate the constraints imposed by the architecture, thus requiring legalization. Rather than performing a simple recursive bi-partitioning on the entire FPGA, like in [27], multiple smaller windows are used for bi-partitioning to generate a more uniform spreading of cells throughout the placement area. Figure 5.2 shows the main steps. Adjacent cells (which could be in the form of LUTs, FFs, DSPs, and BRAMs) that are over utilized (i.e., illegal) are grouped together to form clusters (Fig. 5.2 (a)). Clusters are formed using a recursive search that seeks to combine cells in the same site or adjacent sites. For each new cluster, a minimum area window, called a *Legal-window (L-window)*, is determined. The L-window for a cluster must contain sufficient sites to place all of the FFs and LUTs contained in the cluster. Moreover, it must also contain sufficient sites to place any BRAMs or DSPs in the cluster. This ensures that all of the cells within a cluster can be legalized. If multiple L-windows for different clusters overlap, as shown in Fig. 5.2 (b), cells belonging to the overlapping clusters are merged into a new cluster, and a new L-window is determined (Fig. 5.2 (c)). The cells in each cluster are then legalized within their L-windows using a recursive bi-partitioning procedure that handles FF and control-set constraints as well as LUT

sharing (Fig. 5.2). This process removes the need to perform packing.

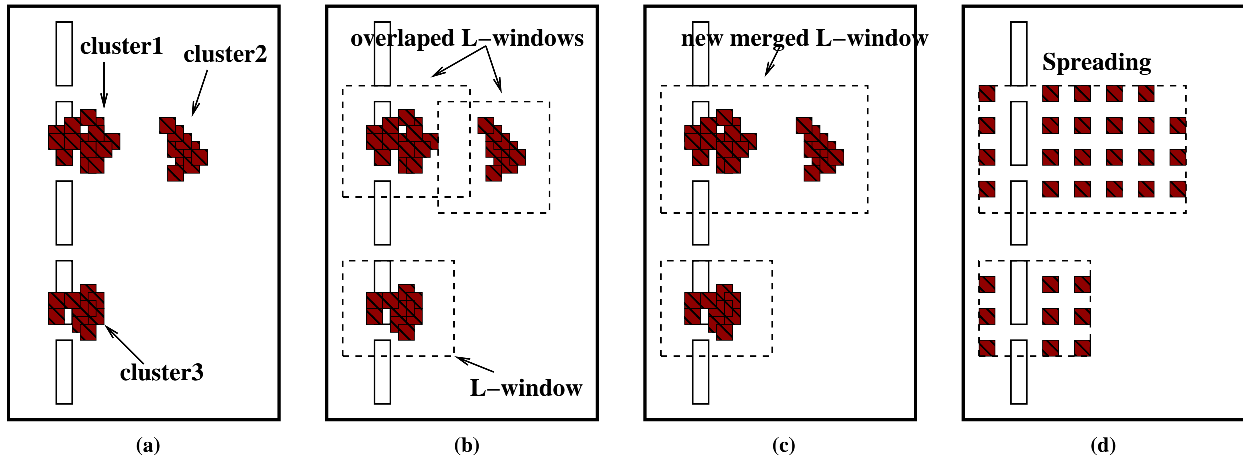


Figure 5.2: Window Based Legalization Flow

5.1.2.1 Find Minimum L-Windows

The L-window corresponding to each cluster must contain sufficient sites to legalize all the FFs, LUTs, DSPs, and BRAMs contained by the cluster. The number of sites required to completely legalize shared LUTs, BRAMs, and DSPs can be computed in O(1) based on their numbers. However, calculating the number of sites required to legalize FFs with control-set constraints takes linear time complexity as explained later in Section 5.1.2.3. The pseudo-code for finding a minimum L-window for each cluster is summarized in Algorithm 5. The process of finding a minimum L-window occurs in three stages. Initially, the L-window is set to the bounding box of the cluster (line 1). The first stage (lines 2-4) then expands the bounding box by doubling the smaller dimension of the L-window until the L-window is large enough to contain all of the cells in the cluster. When doubling a dimension, the L-window is automatically cropped, as necessary, so as to not exceed the dimensions of the FPGA. If the smaller dimension is already as large as it can be (e.g., the L-window already spans the full width of the FPGA but not the height), the larger dimension is doubled instead. The second stage (lines 5-20) performs a binary search to narrow down the approximate minimum dimensions of the L-window. The L-window is only modified on the larger of the two axes. The initial search space (lines 6-7) ranges from 0 to the current dimension of the

ALGORITHM 5: Find Minimum Window

Require: Cluster $cluster$: a group of overutilized adjacent cells.

Ensure: L-window: a minimum rectangular window centered in the middle of the cluster, and sufficient to contain all cluster's cells.

1: $cluster.window \leftarrow cluster.boundingBox$;
2: **while** $cluster.window.resources < cluster.resources$ **do**
3: $cluster.window.smallerDimension \leftarrow cluster.window.smallerDimension * 2$;
4: **end while**

5: $axis \leftarrow cluster.window.largerDimension.axis$;
6: $upperBound \leftarrow cluster.window[axis].max - cluster.window[axis].min$;
7: $lowerBound \leftarrow 0$;
8: $clusterCenter \leftarrow cluster.window[axis].min + (upperBound + lowerBound)/2$;
9: **while** $lowerBound <= upperBound$ **do**
10: $testDim \leftarrow (upperBound + lowerBound)/2$;
11: $cluster.window[axis].min \leftarrow clusterCenter - (testDim/2)$;
12: $cluster.window[axis].max \leftarrow clusterCenter + (testDim/2) + (testDim\%2)$;
13: **if** $cluster.window.resources <= cluster.resources$ **then**
14: $upperBound \leftarrow testDim - 1$;
15: **else**
16: $lowerBound \leftarrow testDim + 1$;
17: **end if**
18: **end while**
19: $cluster.window[axis].min \leftarrow clusterCenter - (lowerBound/2)$;
20: $cluster.window[axis].max \leftarrow clusterCenter + (lowerBound/2) + (lowerBound\%2)$;

21: $shrinking \leftarrow true$;
22: **while** $shrinking$ **do**
23: $shrinking \leftarrow false$;
24: **for** $axis$ in X, Y **do**
25: $cluster.window[axis].min \leftarrow cluster.window[axis].min + 1$;
26: **if** $cluster.window.resources >= cluster.resources$ **then**
27: $shrinking \leftarrow true$;
28: **else**
29: $cluster.window[axis].min \leftarrow cluster.window[axis].min - 1$;
30: **end if**
31: $cluster.window[axis].max \leftarrow cluster.window[axis].max - 1$;
32: **if** $cluster.window.resources >= cluster.resources$ **then**
33: $shrinking \leftarrow true$;
34: **else**
35: $cluster.window[axis].max \leftarrow cluster.window[axis].max + 1$;
36: **end if**
37: **end for**
38: **end while**

L-window on the larger axis. The final stage (lines 21-38) crops the edges of the L-window by a single row/column at a time. The complexity of finding an exact minimum L-window is $O(log(n))$, where n is the size of the larger dimension of the final L-window.

After each cluster's L-window has been determined, it is possible that some of the L-windows overlap. Since each cluster assumes that it has access to all of the resources in its L-window, this can result in insufficient resources to legalize a second cluster in the case of two overlapping clusters. Therefore, before legalizing within an L-window, any clusters whose L-windows are overlapping are merged, and a new L-window is calculated for the merged cluster. This is repeated until no L-windows overlap. In large benchmarks, the number of clusters after all of the minimum L-windows are calculated and clusters merged is sometimes end up with one cluster with a window that is the full FPGA. Every time two clusters are merged, the required sites must be recalculated before an exact L-window can be computed, which can be fairly costly if many clusters need merging. To minimize the number of clusters requiring merging, an estimation of L-window size is computed in $O(1)$ time using only the number of LUTs. This estimation is used to preemptively merge nearby clusters before any exact L-windows are calculated.

Pseudo-code for performing preemptive merging is shown in Algorithm 6. During preemptive merging, L-window sizes and locations are estimated in $O(1)$ time. This estimation is used to preemptively merge nearby clusters before any exact windows are calculated. The complexity of preemptive merging is $O(n^2)$, where n is the number of clusters. Windows are assumed to be square. Windows are centered on the same point as the bounding box for the cluster. When no two clusters have estimated L-windows that overlap, preemptive merging terminates.

5.1.2.2 LUT Sharing in Bi-partitioning

The goal of LUT-sharing is to reduce LUT utilization (i.e., # of 6-input LUTs) throughout the FPGA by seeking opportunities to combine LUTs. A slice in the Xilinx UltraScale architecture contains 8, 6-input LUTs. However, a single 6-input LUT can implement 2 LUTs if together the number of unique inputs is less than 6. GPlace-flat pairs LUTs while aiming to maximize the num-

ALGORITHM 6: Preemptive Merging

Require: $clusters[1..n]$;

Ensure: $clusters[1..m], m \leq n$ list of clusters after merging overlapped clusters by fast guessing of their L-window sizes;

```
1: while true do
2:    merges ← 0;
3:    for i ← 0 to i ← n do
4:       for j ← i + 1 to j ← n do
5:          for cluster in {clusters[i],clusters[j]} do
6:             center ← center(cluster.boundingBox);
7:             sidelength ← sqrt(cluster.numLUTs/8);
8:             window[i] ← square(center, sidelength);
9:          end for
10:         if overlap(window[i], window[j]) then
11:            merge(cluster[i], cluster[j]);
12:            merges ← merges + 1;
13:         end if
14:      end for
15:   end for
16:   if merges == 0 then
17:      break;
18:   end if
19: end while
```

ber of shared inputs and minimize the distance between the paired LUTs in the placement. First, GPlace-flat searches for groups of LUTs that have n common inputs. LUTs with common inputs are found by hashing all possible subsets of their inputs of size n; that is, for sharing n inputs, a k-input LUT hashes all (k choose n) subsets of its inputs and is added to the corresponding buckets. After all LUTs have been hashed, each bucket is partitioned into groups that share the same n inputs. Next, each LUT is paired to the nearest unpaired LUT in the same partition, provided the distance between them is less than 1 in the placement. The pairing process is repeated for all values of n, starting at the highest number of shared inputs (5) and ending with the fewest (0).
The dependence between the sharing of LUTs and the LUT cell capacity of a slice makes identifying overflowing regions difficult. A slice may contain 8, 6-input LUTs, however, if the LUTs are smaller and can thus share, a slice can contain up to 16 LUTs. To resolve this dependence, GPlace-flat pairs LUTs prior to legalization and removes one LUT of each pair from the placement. Removing one LUT from each pair allows bi-partitioning to use a capacity of 8 LUTs per slice. After legalization, the removed LUTs are added back to the placement in the same position as their paired LUT.

5.1.2.3 Flip Flop Control Sets in Bi-partitioning

A slice may contain 16 FFs provided that they satisfy several constraints on their clock, clock-enable, and reset signals. The 16 FFs form a hierarchy based on their control signals. At the top level, the 16 FFs can be subdivided into two groups of 8, with each group sharing the same clock and reset signal. Next, each group of 8 must be subdivided into 2 groups of 4, each group sharing the same clock-enable signal. The bi-partitioning algorithm requires that the number of slices needed for all FFs in a region be computed and updated as FFs move between regions. To compute the slice count, GPlace-flat maintains a tree structure organizing the FFs in the circuit during partitioning. An example of tree structure to compute the required number of slices for FFs in a region is shown in Fig. 5.3. The root of the tree has a child for each region in the tree used for bi-partitioning. Figure 5.3 shows only Region 0 (i.e., RG0). Each region node has a child for each

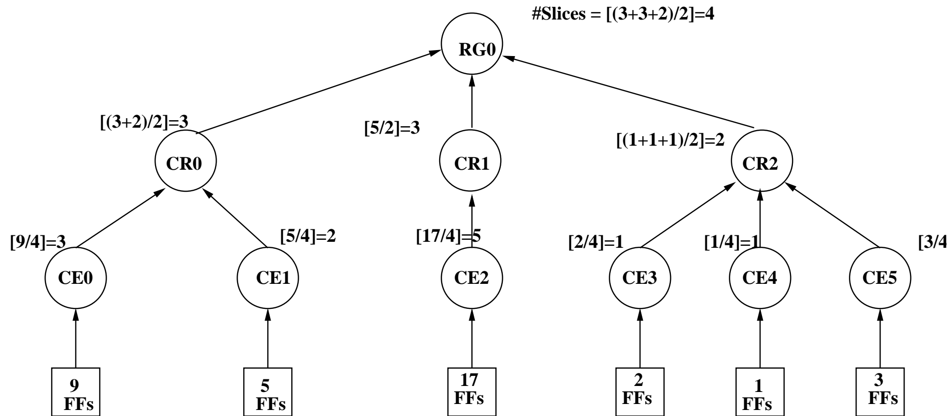


Figure 5.3: Control Signal FF bi-partition Tree

clock and reset combination (i.e., CR# in the region). In the next level, the clock and reset nodes have a child for each clock-enable (e.g., CE#). The next level contains a count of the number of FFs that match the clock-enable, reset, clock, and region as its nodes branch in the tree. Computing the number of slices required in each region is carried out in a bottom-up pass through the tree. At the bottom, the clock-enable node returns the number of groups of 4 it will require ($\lceil \text{FF}/4 \rceil$). The clock and reset node sums the value from each child and returns $\lceil \text{sum}/2 \rceil$, thereby computing the number of half slices. Finally, the region node sums the half slice values from each child and returns $\lceil \text{sum}/2 \rceil$, thereby computing the number of slices required in the region. This tree is used to identify overflow in a region by comparing the number of slices required to legalize FFs in the region with the available slices. Tree branches are updated when moving a FF between regions in $O(1)$ time (the height of the tree is always 3). Figure 5.3 shows that region RG0 contains 37 FFs with different clock, reset, and clock enable signals. If partitioning FFs considers only the capacity of a slice (i.e., 16 FFs per slice), the minimum number of slices required to legalize FFs is 3. However, to enforce all control-set constraints, the exact number of slices required to legalize FFs in RG0 is 4, as shown in Figure 5.3.

5.1.3 [Phase II] Congestion Estimation using a Global Router

The main goal of Phase I is to produce an initial placement that minimizes wirelength and routing demand. However, there is no guarantee that the resulting placement is free of congestion or that it can be even routed. Therefore, the objective of Phase II is to further reduce congestion to increase the likelihood of producing a routable placement without significantly degrading wirelength. The basic idea is to first use a global router once to estimate and identify regions of congestion (as explained in Section 5.1.3.2), and then perform cell (i.e., LUT) inflation to resolve congestion which is presented in Section 5.1.4. Since the global router is only run once, it is imperative that the router produces a very accurate congestion estimate. Therefore, we have made modifications to the PathFinder Global Router [7] to obtain a more accurate congestion estimate. Further motivation and details are given in the following subsections.

5.1.3.1 The Need for Accurate Congestion Estimation

Failure to accurately identify and estimate congested regions may lead to two equally undesirable situations: (1) unnecessary inflation of uncongested cells resulting in excess wirelength and worse routability, or (2) less application of inflation to congested cells, thus keeping the problem of congestion unresolved. Since congestion estimation is only performed once in Phase II, having an accurate congestion estimation technique is crucial. To address this, a modified *PathFinder Global Router (mPFGR)* was developed to accurately estimate congestion. The proposed modifications encompass three important features. The first involves adding an adaptive overflow penalty based on the router's progress to the original, negotiation-based cost function presented in NCTUgr2.0 [49]. This adaptive penalty enables the global router to converge much faster and with better wirelength compared to the original cost function. The second modification introduces two simple heuristic methods to capture the effects of local nets based on the number of pins in the switch box. The goal here is to close the performance gap between the global router and detailed router by taking into account the contribution of local wirelength and routing, respectively, when estimating global congestion. The third modification involves using an A* search to speedup the maze router

employed in mPFGR.

Before presenting the algorithmic details of the previous modifications, we provide the reader with some preliminary visual evidence in the form of congestion heat maps for the effectiveness of mPFGR versus other, well-known congestion-estimation methods. Figure 5.4 shows the congestion heat maps obtained using different techniques including wirelength per area (WLPA), Versatile-Place-and-Route's cost function augmented with a PathFinder congestion estimate (VPR-PF), PathFinder congestion based on NCTUgr2.0 cost (PFGR), mPFGR proposed in this work, and the Vivado detailed router. It is important to emphasize that although the congestion heat maps are for a single ISPD 2016 contest benchmark (i.e., FPGA05), the pictures they give are very typical for congested placements.

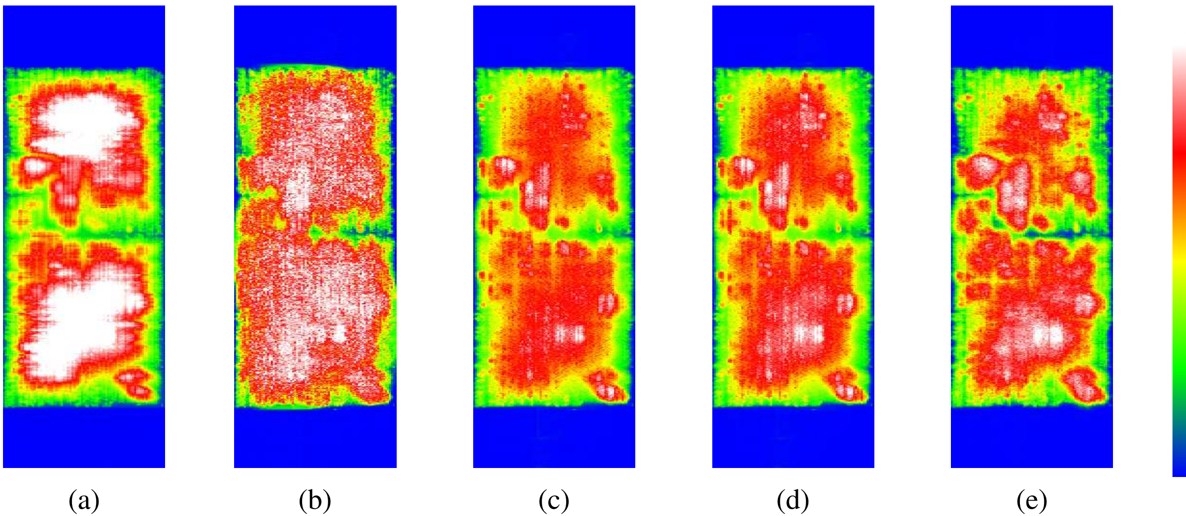


Figure 5.4: Congestion Map of FPGA05 in ISPD2016 contest: (a) WLPA; (b) PFGR (3 iters.)/VPR-PF costs; (c) PFGR (3 iters.)/NCTUgr2.0 cost; (d) mPFGR (3 iters.); (e) Vivado detailed router

In addition to a visual comparison, the *Sum of Absolute Difference (SAD)* and *Average Absolute Normalized Error (AANE)* measures from [50] can be used to numerically measure the difference between congestion heat maps. Table 5.1 shows the SAD and AANE between the different estimation methods relative to the actual congestion after the Vivado detailed router is applied (for two ISPD 2016 contest benchmarks[1]). Based on lower values for SAD and AANE, the proposed

[1]Note: that is typical for the most of ISPD 2016 benchmarks

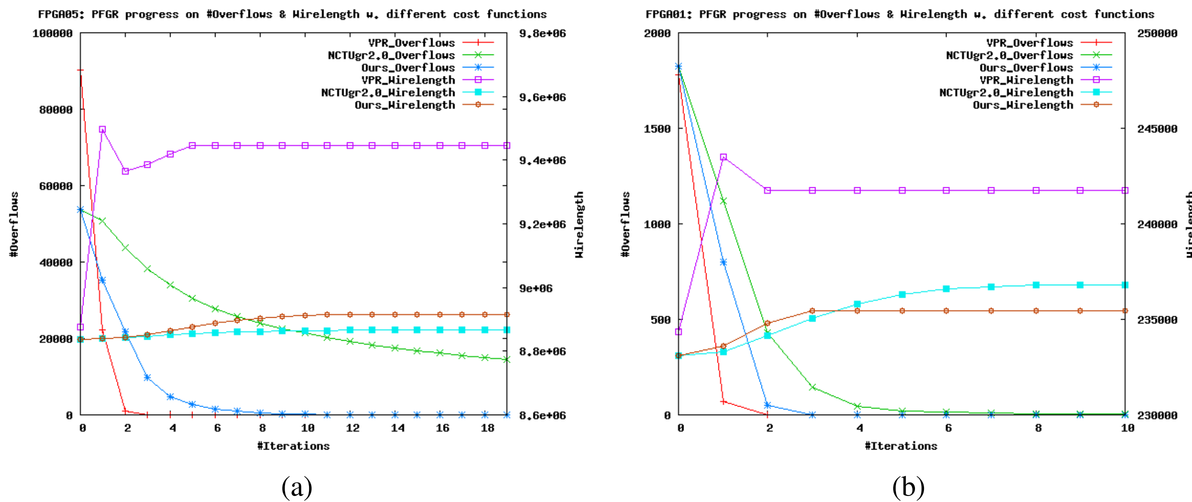
Table 5.1: Comparison Between Different Congestion Estimation Methods

Benchmark	Metric	WLPA	VPR-PF cost	NCTUgr2.0 cost	mPFGR cost
FPGA05	SAD[a]	+8778.4	+9705.8	+3683.4	**+3591.4**
	AANE[b]	0.0976	0.1079	0.041	**0.039**
FPGA03	SAD	+5014.7	+2756.6	+2258.6	**+2204.3**
	AANE	0.0625	0.0343	0.0281	**0.0275**

[a]SAD: Sum of Absolute Difference

[b]AANE: Average Absolute Normalized Error

mPFGR implementation shows superiority over the other methods.



(a) (b)

Figure 5.5: mPFGR with Different Cost Functions: a) FPGA05, and b) FPGA01

5.1.3.2 Modifications to PathFinder

Negotiation-based routers require signals to negotiate with each other to obtain routing resources. ASIC global routers, such as NCTUgr2.0 [49], BFR [51], and FastRoute [52], use different negotiation-based cost functions to overcome the drawbacks that are present in the original PathFinder global router. For example, NCTUgr2.0, a state-of-the-art ASIC global router, employs an adaptive *historical congestion* cost that can decay over time as edges in the (grid) graph become less congested. However, NCTUgr2.0's *overflow* penalty function is not adaptive. Consequently, the router can take a long time to resolve overflows and can converge to a suboptimal solution. This situation is

magnified in highly-congested designs. For example, in Fig. 5.5a the number of overflows for NCTUgr2.0 decreases as the number of iterations increases, but NCTUgr2.0 prematurely converges without resolving a large number of overflows. BFR [51] seeks to avoid the previous problem by adaptively increasing the overflow penalty as the router progresses. However, the overflow penalty increases by a predetermined amount over time. Thus, two very different circuits can be assigned the same penalty at the same relative point in the routing process. Unlike BFR [51], mPFGR seeks to adaptively increase the magnitude of the overflow penalty based on the on-going progress of the router. Specifically, the penalty assigned is based on the ratio of the initial and current overflows and, therefore, takes into account the progress of the router for a particular design.

Table 5.2: PathFinder Global Router Parameter Definitions

Parameter	Definition
p_r	present congestion penalty
h_r	historical congestion penalty
$base_r$	base cost
b_r	manhattan distance between the origin and the destination switches
cap_r	capacity of a given routing resource
dem_r	demand on a given routing resource

The PathFinder cost functions used by mPFGR were adapted from NCTUgr2.0 [49]. Equation 5.1 is used to determine the cost of using a global routing resource [7]. Table 5.2 shows the definition of the parameters presented in PathFinder's cost functions. Equation 5.2 is the sum of the overflows of a global routing resource, and is updated at the end of each PathFinder iteration. Equation 5.3 represents the historical congestion penalty for a global routing resource. Its value depends on the current PathFinder iteration and the value of Equation 5.2. C_1 is a constant which scales the historical congestion penalty and C_2 is a constant which causes the historical congestion penalty to decay based on the number of PathFinder iterations. In our implementation C_1 is set to 7.0 and C_2 is set to 4.0 similar to [49].

$$cost_r(i) = (1 + dah_r(i)) \times p_r(i) + base_r(i) \tag{5.1}$$

$$h_r(i) = \begin{cases} h_r(i-1) & \text{if } cap(r) \geq dem(r) \\ h_r(i-1) + (dem(r) - cap(r)) & \text{otherwise} \end{cases} \tag{5.2}$$

$$dah_r(i) = \frac{h_r(i)}{C_1 + C_2 \times \sqrt{i}} \tag{5.3}$$

Equation 5.4 represents the present congestion penalty for a global routing resource. Its value depends on the demand and capacity of a global routing resource, the ratio of the number of overlaps in the first PathFinder iteration to the most recent PathFinder iteration, and the current PathFinder iteration.

$$p_r(i) = \begin{cases} 1 + \frac{C_3}{1+e^{C_4(cap(r)-dem(r))}} & \text{if } cap(r) > dem(r) \\ 1 + \frac{C_3}{1+e^{C_4(cap(r)-dem(r))}} + (1 + log(\frac{\text{initial overlaps}}{\text{current overlaps}})) \times 20 \times i & \text{otherwise} \end{cases} \tag{5.4}$$

$$base_r(i) = (C_5 + C_6/2^i) \times b_r \tag{5.5}$$

In our implementation, C_3 is set to 150 and C_4 is set to 0.3 similar to [49]. However, equation 5.4 differs from the cost functions used by NCTU-GR 2.0 as a discrete jump in cost is added when the demand of a global routing resource is greater than its capacity. The magnitude of this penalty is determined based on the current PathFinder iteration and the ratio of the number of overlaps in the initial PathFinder iteration to the number of overlaps in the most recent PathFinder iteration. This penalizes global routing resources more heavily if they remain congested during later stages of the route. Equation 5.5 represents the base cost of using a global routing resource. This term decreases for each PathFinder iteration, which decreases the wirelength portion of the cost for each routing resource relative to the overflow/congestion portion of the cost. This contributes to the global router prioritizing wirelength more early in the route and overflow reduction later in the route. In our implementation C_5 is set to 30 and C_6 is set to 200 similar to [49]. This function differs from the cost function used by NCTUgr2.0 in that the term $C_5 + C_6/2^i$ is multiplied by b_r, which is the Manhattan distance between the origin switch and the destination switch of the

global routing resource. This is because NCTUgr2.0 is an ASIC global router which operates on a grid graph, and is not segmented like the global routing architecture of some FPGAs. Figures 5.5a and 5.5b show that our new cost resolves overflows much sooner than the cost functions used in NCTUgr2.0 [49]. To capture the effects of local nets on the global congestion map, two simple heuristic methods are used in mPFGR. The first represents local wirelength based on the number of pins located in a site connected to a switch, and involves adding demands to edges that are connected to the switch box based on Equation 5.6. K_1 is equal to 0.0008 in our experiments.

$$dem(e) = dem(e) + (K_1 \times \#pins \times cap(e)) \tag{5.6}$$

The second heuristic is used to reduce the capacity of the switch boxes by a value proportional to the number of pins located in a site connected to that switch. Since local connections within a tile are not a full cross bar, some of these local nets should be routed globally, outside of the tile. This correction method is inexpensive to calculate, and captures the effect of the local routing by adding blockages to the edges connecting to the switch box based on Equation 5.7. K_2 is 0.0028 in our experiments.

$$cap(e) = cap(e) - (K_2 \times \#pins \times cap(e)) \tag{5.7}$$

5.1.3.3 mPFGR Algorithm: Discussion and Pseudo-code Explanation

The Pseudo-code of the mPFGR proposed in this work is shown in Algorithm 7. mPFGR uses the PathFinder negotiated-congestion algorithm, with the modifications explained earlier, to route all nets and then iteratively resolve overlaps. On each PathFinder iteration (lines 4-40), the routing tree of each net is ripped up and then re-routed by invoking a maze router multiple times (lines 5-39). An A* search is used to speedup the maze router. The goal of the maze router is to find the path from a source node to a destination node with minimum cost. The cost of a path is simply the cost of all the routing resources which make up the path. The cost of using a routing resource is defined by the PathFinder cost functions (line 28).

ALGORITHM 7: PathFinder Global Router

Require: Netlist, placement and global routing resources graph for device

1: $RT[n] \leftarrow \varnothing \quad \forall n in$netlist
2: $h_c[r] \leftarrow 1 \quad \forall r in$global routing resources
3: **do**
4: **for** each net $i \in$ netlist **do**
5: $RT[net_i] \leftarrow \varnothing$
6: **for** $\{1, \ldots, |net_sink_switches_i|\}$ **do**
7: $prev_rr[s] \leftarrow NULL,\ dist[s] \leftarrow \infty,\ visited[s] \leftarrow false \quad \forall s \in$ switches
8: $PQ \leftarrow \varnothing$
9: $dist[origin_switch$ of $net_driver_i] \leftarrow 0$
10: Push $origin_switch$ of net_driver_i into PQ with value 0
11: $dist[destination_switch$ of $r] \leftarrow 0 \quad \forall r \in RT[net_i]$
12: Push $destination_switch$ of r into PQ with value 0 $\quad \forall r \in RT[net_i]$
13:
14: **while** $PQ \neq \varnothing$ **do**
15: $current \leftarrow$ pop top of PQ
16: $visited[current] \leftarrow true$
17: **if** $current \in net_sink_switches_i$ and sink not found **then**
18: Mark sink as found
19: $traceback_switch \leftarrow current$
20: **while** $prev_rr[traceback_switch] \neq NULL$ **do**
21: $RT[net_i] \leftarrow RT[net_i] \cup prev_rr[traceback_switch]$
22: $traceback_switch \leftarrow origin_switch$ of $prev_rr[traceback_switch]$
23: **end while**
24: **break**
25: **else**
26: **for** each outgoing global routing resource r of $current$ **do**
27: $dest_sw \leftarrow destination_switch$ of r
28: $grr_cost_r \leftarrow (1 + dah_r(i)) \times p_r(i) + base_r(i)$
29: $new_cost \leftarrow dist[current] + grr_cost_r$
30: **if** $visited[dest_sw] = false$ and $new_cost < dist[dest_sw]$ **then**
31: $dist[dest_sw] \leftarrow new_cost$
32: $prev_rr[dest_sw] \leftarrow r$
33: Push $dest_sw$ into PQ with value new_cost
34: **end if**
35: **end for**
36: **end if**
37: **end while**
38: **end for**
39: **end for**
40: $h_c[r] \leftarrow h_c[r] + min(0, dem[r] - cap[r]) \quad \forall r \in$ global routing resources
41: **while** overlaps exist

The PathFinder cost functions are then updated based on the current usage of each routing resource, the historical usage of each routing resource, and the current PathFinder iteration. The current usage of each routing resource is updated dynamically based on the route trees of each net, while the historical usage of each routing resource is updated at the end of each PathFinder iteration (line 40). Initially the global router prioritizes minimizing wirelength. The dynamic PathFinder cost functions result in overlap reduction being prioritized more in later iterations.

Each time a net is routed the maze router is invoked $|S_{net}|$ times, where S_{net} is the set of sinks for a net. Each time the maze router is invoked (lines 8 to 39), the search is initialized with the switches in the current routing tree for the net (lines 9 to 12). Initially the route tree for the net is empty, and the search is initialized with just the switch of the net driver. When a sink is located the current route tree for the net is expanded to include the path to the sink (lines 18-23), and the maze router is restarted. This step is repeated until all sinks are located and the route tree for the net is connected to all of the nets pins.

5.1.3.4 Algorithm Complexity

The maze router is invoked $i_p \times n \times \bar{s}$ times, where i_p is the number of PathFinder iterations required to remove all overlaps, n is the number of nets in the netlist and $\bar{s}$ is the average number of pins per net. The maze router uses breadth first search which has a worst case time complexity of $\mathcal{O}(|V| + |E|)$, where V is the set of switches and E is the set of global routing resources. Thus, the overall time complexity for the global router is $\mathcal{O}(i_p \times n \times \bar{s} \times (|V| + |E|))$.

5.1.4 [Phase II] Cell (LUT) Inflation

The first step to relieve and mitigate congestion is addressed by estimating congestion via the proposed Global Router (mPFGR) introduced in the previous section. Resolving congestion involves a second important step in the form of cell/LUT inflation. The basic idea behind cell inflation is to artificially increase the size (i.e., area) of cells found in congested regions, so that fewer of those cells occupy the congested region (i.e., reduce the cell density), thus leading to less congestion.

Consequently this should minimize the local pin density in the identified congested area and ultimately improve routability. Figure 5.6 shows the affect of inflating cells/LUTs to further resolve congestion within GPlace-flat.

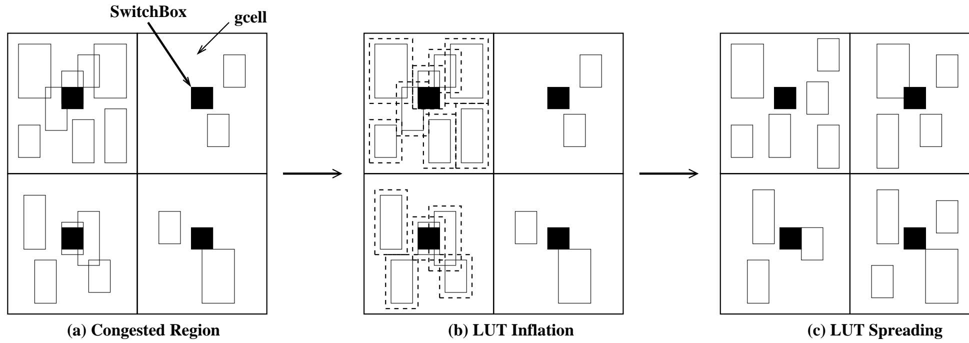


Figure 5.6: Resolving Congestion via Cell Inflation

The cell-inflation method employed by GPlace-flat is based on work published in [53] and involves performing the following steps. The cell density is computed for LUTs based on its number of pins, congestion level of the gcell it is located in (as shown in Figure 5.6), and the congestion level of the circuit/benchmark in total. The density of FFs, RAM, and DSP are not varied. Equations 5.8-5.9 show how LUTs are inflated based on congestion.

$$\mathrm{S} = \mathrm{C_1} + \mathrm{Cong_B}.\mathrm{C_2} \tag{5.8}$$

The factor S indicates the amount of congestion of the benchmark, where $C_1 = 0.36, C_2 = 0.28$ are constants empirically found, and $Cong_B$ is the average congestion of the top 10% congested switches (e.g. congestion level $> .65$). Originally, LUT density is set to 1. If the LUT is located withing a congested switch box that has a congestion ≥ 0.5, then the LUT density is computed as follows:

$$\mathrm{density(LUT)} = 1 + \mathrm{S}.(\frac{\mathrm{inputs(LUT)} + \mathrm{outputs(LUT)}}{\mathrm{ratio}.\mu_{LUT}} - 1) \tag{5.9}$$

where μ_{LUT} is the average number of pins of LUTs in the net-list, and $ratio$ is computed based on

the congestion value of the switch that LUT is connected to.

$$ratio = \begin{cases} 0.8 & \text{if } 0.5 \leq cong < 0.675 \\ 0.7 & \text{if } 0.675 \leq cong < 0.85 \\ 0.6 & \text{if } 0.85 \leq cong < 1.025 \\ 0.5 & \text{if } 1.025 \leq cong < 1.2 \\ 0.4 & \text{if } cong \geq 1.2 \end{cases}$$

Bi-partitioning uses the calculated density of cells to decide if a region is overflowing when moving cells between regions. If both regions are overflowing, the legalization will attempt to evenly distribute the density between regions.

5.1.5 [Phase II] Congestion-driven Global Placement

Following congestion estimation and cell inflation, a second round of flat placement is performed with the aim of improving wirelength. This optimization is necessary, as wirelength tends to deteriorate after performing cell inflation and (local) LUT movement. Overall, the congestion-driven global placement flow in Phase II shares the same framework with the WL-driven global placement flow in Phase I, but with the exception that LUT bi-partitioning handles different cell (i.e., LUT) densities during legalization.

5.1.5.1 Cell Density in Bi-partitioning

In the case of cell-density, there are two types of overflow that can occur: *hard* and *soft*. Hard overflows occur when a placement region does not have enough sites to accommodate all of the cells (i.e., LUTs and flip-flops) assigned to that region. In GPlace-flat, hard constraints are strictly avoided by using bi-partitioning to ensure legality. However, soft overflows are permitted. Soft overflows occur when the total cell density exceeds the capacity of the placement region. In the

latter case, cell density is kept roughly balanced between the two partitions. Figure 5.7 provides an illustration of cell-density bipartitioning. For the sake of simplicity, each site in Fig. 5.7 is assumed to have a unit area. Initially, all LUTs are assumed to have an area of 1 (Fig. 5.7(a)). Following LUT inflation, LUTs c and d have their area increased to 2 (Fig. 5.7(b)). During LUT bi-partitioning, two constraints must be satisfied. First, each partition must have enough sites to accommodate the total number of LUTs assigned to the partition. (This is a hard constraint.) Second, the total area of the LUTs assigned to a partition must not exceed the number of sites in that partition. (This is a soft constraint.) In Fig. 5.7(b), the left partition satisfies the hard constraint, since there are three available sites and three LUTs (i.e., a, b, and c). However, the soft constraint for the (left) partition is not satisfied. This is because the total area of the three LUTs (i.e., a, b, and c) in the (left) partition is 4 (i.e., 1+1+2), and, therefore, exceeds the number of available sites (i.e., 3) in the partition. In this case, the left partition has a soft overflow of 1. Both hard and soft constraints are met for the right partition, since both the number of LUTs (i.e., 3) and the total area of the LUTs (i.e., 4) is less than the number of sites (i.e., 6) in the partition. When performing cell-density bi-partitioning, the goal is to minimize the displacement of the cells from their original positions (calculated by the analytic solver) such that both hard and soft constraints are met in each partition. As shown in Fig. 5.7(c), this can be achieved by selecting LUT c, since it is closer to the cut, and moving it to the right partition. For some partitions, the soft constraints cannot be satisfied. Therefore, the goal is to try and keep the density of the LUTs in each partition (roughly) the same.

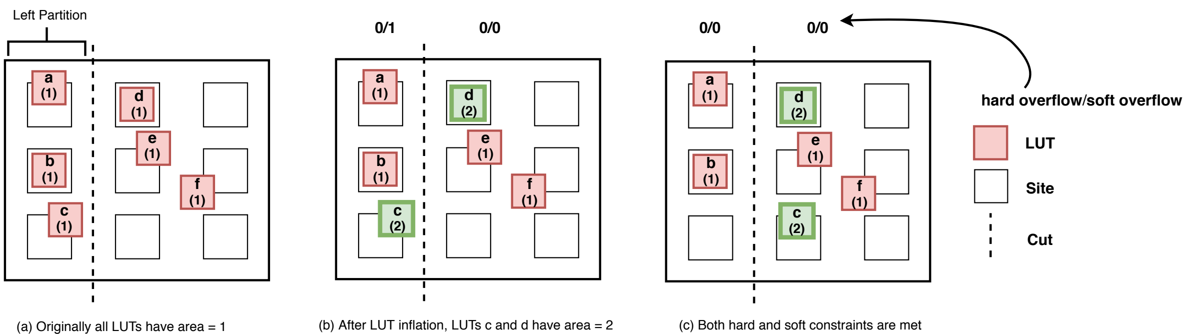
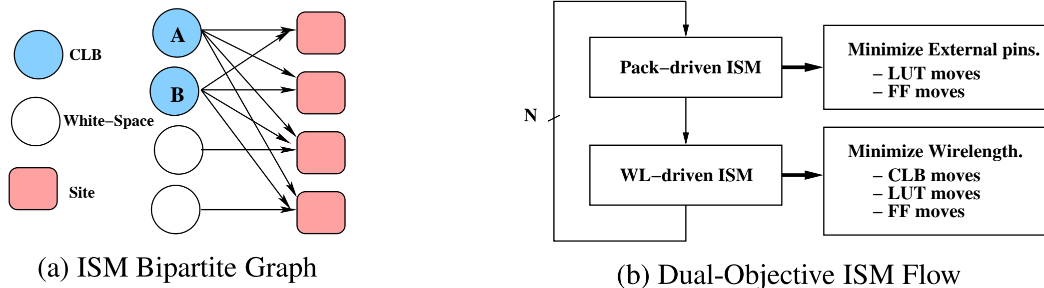


Figure 5.7: LUT bipartitioning with Cell-density

5.1.6 [Phase III] Dual-Objective Hierarchical Independent Set Matching

After Phase II, the current placement has much less congestion compared to the original placement produced by Phase I. However, there is still room for further improvement. In Phase III, we employ a detailed placement technique, called Dual-Objective Independent Set Matching (DOISM), to reduce both wirelength and external pin count. (Reduction of the latter effectively improves routability, as shown in [54].) DOISM is based on the *Independent Set Matching (ISM)* technique used successfully in UTPlace3.0 [5]. The basic idea behind ISM is to use bipartite matching to improve wirelength. The bipartite graph consists of two disjoint sets: the first contains cells, and the second contains the current locations of the cells as well as a few locations representing white-space, as shown in Fig. 5.8a. A weighted edge connects each cell in the first set to each location in the second set, where the weight represents the HPWL wirelength between the cell and the location. By finding a minimum weight matching, cells can be moved (locally) to minimize wirelength. The key concept in ISM involves ensuring that all of the cells in the first set do not share any nets; hence, all of the cells can be either swapped or moved (to white-space) without affecting the wirelength of any of the other cells in the set.



(a) ISM Bipartite Graph

(b) Dual-Objective ISM Flow

Figure 5.8: Dual-Objective Independent Set Matching

DOISM shares some similarity to the ISM implementation in [5], but differs in its application. Not only is the conventional ISM used to optimize wirelength, it is also used to reduce external pin count to improve routability. As shown in Fig. 5.8b, DOISM alternates between optimizing wirelength and reducing external pin count. DOISM seeks to reduce the number of external pins by moving FFs and shared LUTs. The ability to work directly with FFs at this stage of the flow

is possible because, unlike [5], FFs have not already been permanently assigned to BLEs. This decision improves global placement earlier in the flow by allowing LUTs and FFs to move freely during optimization while avoiding being locked into a slice. DOISM minimizes wirelength by moving CLBs, FFs, and shared LUTs in a hierarchical fashion. Details are given in the following subsection.

5.1.6.1 The DOISM Flow

Algorithm 8 describes DOISM in detail. Note that DOISM can either minimize wirelength or minimize pin count simply by changing the weights in the independent set-matching graph. The main loop (lines 1-14) iterates over every node (e.g., FF, LUT, or CLB) present in the placement, until the cost improvement drops below a required improvement threshold I_{th}. For each round of DOISM, a window (local neighborhood) is generated (line 2) by first choosing the node that has participated in DOISM the fewest number of times to act as the center of the window. This window contains all other nodes and white-spaces that are within a distance D_{is} of the chosen (center) node. An independent set is then generated using only the nodes contained within the window (line 3). Each of these nodes, and their locations on the FPGA, are then used to create a bipartite graph (line 4). Edges are added to the bipartite graph connecting each node to each location, and the cost of the edge is calculated (lines 5-8). A min-cost bipartite matching is performed on the graph using the Hungarian method (line 9). The location of each node in the graph is then updated (line 10), and the number of moves are recorded. Once the total number of moves reaches the threshold F_{cu}, a congestion estimation step is performed (line 12), if the congestion-aware feature is enabled.

The independent sets used by DOISM are generated in lines 16 to 33. Initially, every node is considered to be independent. The first loop (lines 18 to 31) searches each location within the window. The primary search method used in this implementation begins with the node at the center of the window that was chosen earlier (line 2). The algorithm then searches every other location in sequence, starting with those nearest to the center and searching outwards towards the window boundary. This ensures that many small movements, which are more likely to be favorable, are

ALGORITHM 8: Congestion-Aware Independent Set Matching

Require: Maximum independent set size N_{is}, maximum independent set radius D_{is}, congestion threshold U_{th}, required improvement threshold I_{th}, and congestion update frequency F_{cu}

1: **while** $improvement > I_{th}$ **do**
2:     Select a window W with a radius of D_{is} nodes
3:     $S \leftarrow$ **GenerateIndependentSet**(W)
4:     Add the nodes in S and their locations to bipartite graph g
5:     **for** each pair (n, l) with $n \in$ node set of g, $l \in$ location set of g **do**
6:         $cost \leftarrow$ **GetMovingCost**(n, l)
7:         Add the edge $(n, l, cost)$ to g
8:     **end for**
9:     Perform min-cost bipartite matching on g
10:     Update the location of each node in g
11:     **if** nodes moved since last congestion update $> F_{cu}$ **then**
12:         Update congestion estimates
13:     **end if**
14: **end while**
15:
16: **function GenerateIndependentSet**(W)
17: $indep[n] \leftarrow$ true for all nodes n
18: **for** each location l in W **do**
19:     **for** each node and white space n at location l **do**
20:         **if** $indep[n]$ **then**
21:             $S \leftarrow S \cup \{n\}$
22:             **if** $|S| \geq N_{is}$ **then**
23:                 **return** S
24:             **end if**
25:             **for** each node m connected to n **do**
26:                 $indep[m] \leftarrow false$
27:             **end for**
28:             Continue to next location l
29:         **end if**
30:     **end for**
31: **end for**
32: **return** S
33: **end function**
34:
35: **function GetMovingCost**(n, l)
36: **if** n is a white space **then**
37:     **if** congestion at the location of $n > U_{th}$ **then**
38:         **return** ∞
39:     **else**
40:         **return** 0
41:     **end if**
42: **else if** congestion at $l > U_{th}$ **then**
43:     **return** ∞
44: **else if** the move would result in an illegal placement **then**
45:     **return** ∞
46: **end if**
47: **return** change in HPWL of moving n to location l
48: **end function**

considered. An alternative search method is also included in the implementation that chooses the locations within the window in a random order. At each location there is only one CLB, but there may be multiple nodes when performing DOISM on LUTs or FFs. In the latter cases, a single node is chosen at random from among those present at the location. By limiting DOISM to one node per location at a time, the legality of the placement is guaranteed to be independent of the movement of any other node. This also has the benefit of increasing the solution space by ensuring the maximum number of distinct locations can be matched. The second loop (lines 19 to 30) attempts to add each node from the location until one is successfully added. If the chosen node is independent of the current set of nodes, it is added to the set at line 21, and then from each node that shares a net with it is marked as no longer being independent (lines 25-27). The set generation terminates once every location in the window has been searched, or the set reaches its maximum size of N_{is} (lines 22-24). The moving costs (which correspond to the edge weights in the bipartite graph) are then determined (lines 35-48). For white-spaces, if the node is currently in a region with congestion greater than U_{th} the cost is set to infinity, effectively ensuring that the location is not chosen in the final matching. Otherwise, the cost is set to 0 (line 37 to 41). For any other node, if the congestion at the destination exceeds the threshold value, the cost is set to infinity (line 43). The cost is also set to infinity (line 45) if a move would result in an illegal placement. This is mainly a concern with FFs, where there are restrictions on the number of different clock, clock enable, and reset signals available within a CLB. Finally, if none of the previous restrictions prevent a move, the movement cost is calculated (line 47). This cost corresponds to the estimated change in wirelength that would result from the move, and is found by calculating the HPWL of each net connected to the node before and after the move.

5.1.6.2 Time Complexity

The time complexity of DOISM is $O(u(lD_{is}^2 + k^2\frac{u}{v}N_{is}^2 + N_{is}^3))$, where u is the number of nodes (i.e., LUTs, FFs, and CLBs) in the circuit, v is the number of nets, k is the average number of nets connected to each node, and l is the number of nodes (i.e., LUTs, FFs, and CLBs) at each

location within D_{is}. Each time function $GenerateIndependentSet$ is called, the outer loop (lines 18 to 31) iterates up to $O(D_{is}^2)$ times. The next (inner) loop (lines 19 to 30) iterates $O(lD_{is}^2)$ times. However, the interior loop (lines 25 to 27) is only reached up to N_{is} times, so line 26 is executed $O(k^2\frac{u}{v}N_{is})$ times since each node is connected to k nets, and each net has $k\frac{u}{v}$ pins. This gives $GenerateIndependentSet$ a time complexity of $O(lD_{is}^2 + k^2\frac{u}{v}N_{is})$.

Each call to function $GetMovingCost$ has a worst case time complexity of $O(k^2\frac{u}{v})$, which occurs when the HPWL update on line 47 requires a full recalculation. When a full recalculation is not required, the time complexity is $O(1)$. The time complexity of constructing the bipartite graph (lines 3 to 8) is $O(lD_{is}^2 + k^2\frac{u}{v}N_{is}^2)$. $GenerateIndependentSet$ is called once (line 3), and then $GetMovingCost$ is performed N_{is}^2 times (line 6).

The Hungarian algorithm used for bipartite matching (line 9) requires $O(N_{is}^3)$ time. Updating the location of each node (line 10) requires $O(k^2\frac{u}{v}N_{is})$ time, since the HPWL for each net connected to the N_{is} nodes must be updated. Finally, the independent set matching is performed $O(u)$ times, resulting in the overall time complexity given above.

5.2 Experimental Design

In this section we present results obtained by GPlace-flat in terms of solution quality and CPU time. Benchmarks and experimental setup, which previously presented in Chapter 3 are used in this work. First, we justify the different modules proposed in our framework and highlight the contributions of each. In Section 5.2.1, the effectiveness of the DOISM detailed placer is presented. This is followed by an explanation of the benefits of the window-based legalization approach proposed in Section 5.2.3. The breakdown analysis of the runtime of GPlace-flat is then introduced in Section 5.2.4. Next, a detailed comparison with state of the art placers proposed in the literature is discussed in Section 5.2.5. In Section 5.2.6, the Vivado router runtime on placements produced by GPlace-flat and other placers is compared. The timing results on placements produced by GPlace-flat and other placers are compared in Section 5.2.7. Finally, in Section 5.2.8, additional

experiments are conducted on an extended set of benchmarks (372 total benchmarks) to highlight the robustness and efficiency of GPlace-flat placer and compare it again with other state-of-the-art FPGA placers.

5.2.1 Dual-Objective Interleaving Independent Set Matching (DOISM) Effectiveness Validation

To demonstrate the effectiveness of the proposed dual-objective hierarchical ISM (i.e., DOISM), two experiments are conducted. First, DOISM is compared with the ISM proposed in [5] in terms

Table 5.3: Comparison Between Dual-Objective ISM (DOISM) and ISM in [5]

Benchmark	ISM [5]				DOISM			
	Ext. pins[a]	Ext. nets[b]	HPWL	Routed WL	Ext. pins	Ext. nets	HPWL	Routed WL
FPGA-1	198083	64698	294219	379016	**172728**	**55301**	**291087**	**355720**
FPGA-2	345592	104505	666554	661668	**297309**	**87404**	**660105**	**644438**
FPGA-3	1045500	276455	2628590	3181163	**924959**	**236509**	**2604281**	**3101107**
FPGA-4	1198410	300478	5203390	5528940	**1055940**	**258181**	**5148517**	**5402715**
FPGA-5	1393180	328140	10635200	10861993	**1219330**	**282078**	**10294775**	**10507039**
FPGA-6	1724060	483992	4467140	6007099	**1543480**	**425564**	**4411627**	**5820321**
FPGA-7	1948990	522744	8297760	9755853	**1727360**	**453892**	**8209415**	**9508871**
FPGA-8	1805230	459024	7704620	8287680	**1581590**	**387828**	**7642291**	**8126475**
FPGA-9	2203190	568697	10023400	11988132	**1965080**	**493194**	**9940218**	**11710738**
FPGA-10	1895810	618329	4867570	7194330	**1684840**	**541593**	**4807377**	**6836300**
FPGA-11	2030210	532315	9540550	10520913	**1778600**	**456910**	**9403768**	**10260312**
FPGA-12	1929700	599468	5561020	7514971	**1662010**	**502354**	**5479660**	**7224371**
Norm.	**+0.00%**	**+0.00%**	**+0.00%**	**+0.00%**	**-11.88%**	**-13.95%**	**-1.43%**	**-2.91%**

[a]Ext. pins: refer to exposed pins of Slices (CLBs) and ignores absorbed pins within a Slice (CLB).

[b]Ext. nets: refer to exposed nets of Slices (CLBs) and ignores absorbed nets within a Slice (CLB).

of external pins, external nets, HPWL, and routed wirelength. Table 5.3 clearly shows that the proposed DOISM results in reductions of -11.88%, -13.95%, -1.43%, and -2.91% [5] in terms of number of external pins, number of external nets, HPWL, and routed wirelength, respectively[2]. Reductions in the number of external pins (and external nets) compared with ISM [5] can be directly attributed to the fact that DOISM seeks to minimize external pin count. However, DOSIM is also able to obtain a reduction in HPWL.

In GPlace-flat, DOISM is applied hierarchically to CLBs, LUT pairs, and FFs. Because

[2]DOISM achieves better routed wirelength in all cases compared to ISM

Table 5.4: Routed Wirelength Before and After Dual-Objective ISM (DOISM)

Benchmark	Post Global Placement	Post DOISM	Improvement
	Routed WL[a]	Routed WL	(%)
FPGA-1	463895	355720	-23.32%
FPGA-2	753476	644438	-14.47%
FPGA-3	3491431	3101107	-11.18%
FPGA-4	5949053	5402715	-9.18 %
FPGA-5	11570305	10507039	-9.19 %
FPGA-6	6800940	5820321	-14.42%
FPGA-7	10604927	9508871	-10.34%
FPGA-8	8813467	8126475	-7.79 %
FPGA-9	12982007	11710738	-9.79 %
FPGA-10	9382827	6836300	-27.14%
FPGA-11	11126867	10260312	-7.79 %
FPGA-12	8406145	7224371	-14.06%
Total	**90345340**	**79498407**	**-12.01%**

[a]Routed WL is based on Vivado detailed router

GPlace-flat does not have an initial, explicit packing stage, we do not necessarily start with strong connectivity between LUTs and FFs within the CLBs. However, by using DOISM to minimize HPWL and external pin-count in alternating fashion, the connectivity within the CLBs can be gradually improved under the influence of both objectives. Moving CLBs not only improves HPWL, but also helps to escape local minima. In addition, minimizing the external pins and nets reduces the workload of the router, as the router has fewer numbers of nets, and nets with fewer pins, to route. (Intuitively, routing an n-pin net is typically easier than routing an m-pin net, where $m > n$.) Therefore, DOISM achieves better routed wirelength than ISM in GPlace-flat flow.

In the second experiment, routed wirelength is compared before and after applying DOISM. Table 5.4 clearly shows that DOISM produces placements with (on average) -12.01% less routed wirelength.

Table 5.5: Comparison Between Different Congestion Estimation Methods

Benchmark	WLPA		VPR cost		NCTUgr2.0 cost		mPFGR cost	
	SAD[a]	AANE	SAD	AANE	SAD	AANE	SAD	AANE
FPGA-1	438.13	0.0101	533.04	0.0123	382.30	0.0088	385.06	0.0089
FPGA-2	1054.19	0.0234	956.02	0.0216	721.06	0.0160	721.86	0.0160
FPGA-3	3315.14	0.0506	2997.82	0.0458	1947.83	0.0297	1948.33	0.0297
FPGA-4	4816.34	0.0664	5490.95	0.0757	2441.59	0.0337	2440.95	0.0336
FPGA-5	8861.61	0.1003	9437.32	0.1068	3492.89	0.0395	3407.85	0.0385
FPGA-6	4455.26	0.0680	4848.85	0.0740	3150.70	0.0481	3146.93	0.0480
FPGA-7	5656.85	0.0666	8743.73	0.1030	3899.12	0.0459	3887.16	0.0458
FPGA-8	6804.91	0.0924	9039.32	0.1227	4322.54	0.0587	4329.88	0.0588
FPGA-9	7697.48	0.0915	10791.36	0.1283	4652.12	0.0553	4622.25	0.0549
FPGA-10	7397.77	0.0857	7263.71	0.0841	6590.56	0.0763	6541.23	0.0758
FPGA-11	8405.49	0.0972	11014.24	0.1274	4813.60	0.0557	4811.26	0.0557
FPGA-12	6796.92	0.0872	6810.71	0.0874	5294.54	0.0679	5255.54	0.0674

[a] All metrics are relative to the baseline congestion obtained after Vivado detailed router

5.2.2 Congestion Estimation

In this section, we compare the performance of mPFGR to other, well-known congestion estimation methods, including Wire Length Per Area (WLPA), VPR's PathFinder cost (PFGR), and the negotiation based cost function embedded in the ASIC NCTU Global Router (NCTUgr2.0). Table 5.5 shows the SAD and ANNE values for each of the previous estimation methods for the 12 ISPD benchmarks. The results in Table 5.5 show that both mFPGR and NCTUgr2.0 always achieve lower SAD and ANNE values compared with WLPA and VPR's PathFinder. When compared head-to-head, mFPGR obtains lower SAD and ANNE values compared with NCTUgr2.0 for 8 of the 12 benchmarks. In general, mFPGR outperforms NCTUgr2.0 on the larger, more congested benchmarks. The impact of congestion estimation on routability, routed wirelength, and the runtime of the Vivado router will be discussed in Section 5.2.8.

5.2.3 Window-based Bi-partition Legalization Effectiveness Validation

Table 5.6 shows the effectiveness of the window-based legalization (i.e., WB-Legalization) approach compared to the non-window-based legalization that uses the entire FPGA. Results ob-

Table 5.6: Routed Wirelength w. and w/o Window-Based Legalization

Benchmark	w/o window-based legalization Routed WL	w. window-based Legalization Routed WL	Improvement (%)
FPGA-1	388206	**355720**	**-8.37%**
FPGA-2	695076	**644438**	**-7.29%**
FPGA-3	3939870	**3101107**	**-21.29%**
FPGA-4	5627399	**5402715**	**-3.99%**
FPGA-5	10674889	**10507039**	**-1.57%**
FPGA-6	5934822	**5820321**	**-1.93%**
FPGA-7	9561706	**9508871**	**-0.55%**
FPGA-8	**8062642**	8126475	0.79%
FPGA-9	**11583546**	11710738	1.10%
FPGA-10	6944954	**6836300**	**-1.56%**
FPGA-11	10337550	**10260312**	**-0.75%**
FPGA-12	7261613	**7224371**	**-0.51%**
Total	**81012273**	**79498407**	**-1.87%**

tained in Table 5.6 clearly indicate that the WB-Legalization outperforms the non-window-based legalization on average by 1.87% in routed wirelength. The most improvement occurs for the smaller benchmarks (i.e., FPGA01- FPGA04). This is because the area that the inflated cells cover for these smaller benchmarks tends to be much less than the total area available on the FPGA. Working within this smaller area, WB-legalization is able to spread cells out uniformly from the center of each congestion hotspot, thus avoiding potentially large increases in wirelength when seeking to reduce congestion. The same is less true for larger benchmarks. The area occupied by inflated cells tends to be much larger, possibly occupying the entire FPGA. Consequently, cells may have to move much farther from their original positions to avoid congestion, thus adversely affecting wirelength.

5.2.4 Runtime Analysis

The runtime breakdown of GPlace-flat is shown in Table 5.7. On average, 68.84% of the total runtime is consumed by flat global placement (34.43% by WL-driven Global Placement, and 34.41% by Congestion-driven Global Placement), while Congestion estimation based on mPFGR and the dual-objective ISM detailed placement (DOISM) consume 13.71%, and 17.06% of the total run-

Table 5.7: Runtime Breakdown of GPlace-flat in (seconds)

Benchmark	GP1(s) WL-Driven	GR(s) Global Router	GP2(s) Cong-Driven	DP (s) DOISM	Others (s)	Total (s)
FPGA-1	18	11	18	23	1	71
FPGA-2	34	21	34	45	2	136
FPGA-3	163	87	162	111	2	525
FPGA-4	170	114	171	112	2	569
FPGA-5	181	170	182	118	3	654
FPGA-6	369	137	368	182	4	1060
FPGA-7	398	174	399	192	4	1167
FPGA-8	400	203	399	203	5	1210
FPGA-9	551	240	547	240	5	1583
FPGA-10	588	134	595	264	6	1587
FPGA-11	523	194	518	243	5	1483
FPGA-12	732	159	732	312	7	1942
Norm.	**34.43%**	**13.71%**	**34.41%**	**17.06%**	**0.38%**	**100%**

time, respectively.

5.2.5 Comparison with Previous Works

In this section, we compare the results achieved by GPlace-flat with the top 3 winners of the ISPD'16 placement contest as well as other state-of-the-art FPGA placers. The comparison is based on both routed wirelength and runtime. All routed wirelength are reported by Xilinx Vivado v2015.4. Normalized results in the last row of each table (Table 5.8, and Table 5.9) are based on the comparisons with GPlace-flat, and only benchmarks that state-of-the-art placers completed successfully are considered. Table 5.8 compares GPlace-flat to the three ISPD contest winners using the 12 contest benchmarks in Table 3.1. Unlike the contest winners, GPlace-flat finds a routable solution for all 12 benchmarks. Moreover, it obtains the best solution for each benchmark. GPlace-flat outperforms the top three placers in routed wirelength by 7.53%, 15.15%, and 33.50%, respectively. Even though the top three winners at the recent ISPD'16 FPGA placement contest along with our proposed GPlace-flat run on different machines, we can still see that the runtime of GPlace-flat is faster by 2.35X, 3.52X, and 4.76X than these placers, respectively.

Table 5.9 compares GPlace-flat to the most recently improved versions of the first and second

Table 5.8: Comparison with ISPD'16 Contest Winners on ISPD 2016 Benchmark Suite

Benchmark	1st Place (UTPlaceF1.0)		2nd Place (RippleF1.0)		3rd Place (GPlace1.0)		GPlace-flat	
	Routed WL	Runtime(s)	Routed WL	Runtime(s)	Routed WL	Runtime(s)	Routed WL	Runtime(s)
FPGA-1	PE[a]	-	379932	118	581975	97	**355720**	**71**
FPGA-2	677877	435	679878	208	1046859	191	**644438**	**136**
FPGA-3	3223042	1527	3660659	1159	5029157	862	**3101107**	**525**
FPGA-4	5628519	1257	6497023	1149	7247233	889	**5402715**	**569**
FPGA-5	10264769	1266	UR	-	UR	-	**10507039**	**654**
FPGA-6	6630179	2920	7008525	4166	6822707	8613	**5820321**	**1060**
FPGA-7	10236827	2703	10415871	4572	10973376	9169	**9508871**	**1167**
FPGA-8	8384338	2645	8986361	2942	12299898	2741	**8126475**	**1210**
FPGA-9	UR[b]	-	13908997	5833	UR	-	**11710738**	**1583**
FPGA-10	PE	-	PE	-	UR	-	**6836300**	**1587**
FPGA-11	11091383	3227	11713479	7331	UR	-	**10260312**	**1483**
FPGA-12	9021769	4539	PE	-	UR	-	**7224371**	**1942**
Norm.	**+7.53%**	**2.35X**	**+15.15%**	**3.52X**	**+33.50%**	**4.76X**	**0.00%**	**1.00X**

[a]PE: Placement error

[b]UR: Unroutable placement

Table 5.9: Comparison with State-of-the-Art Academic FPGA Placers

Benchmark	RippleF2.0 [10]		UTPlaceF2.0 [4]		UTPlaceF3.0 [5]		GPlace-flat	
	Routed WL	Runtime(s)	Routed WL	Runtime(s)	Routed WL	Runtime(s)	Routed WL	Runtime(s)
FPGA-1	362563	74	384709	195	356769	152	**355720**	**71**
FPGA-2	677563	167	652690	350	**642108**	244	644438	**136**
FPGA-3	3617466	1037	3181331	1271	3215087	659	**3101107**	**525**
FPGA-4	6037293	621	5504083	1117	5409765	635	**5402715**	**569**
FPGA-5	10455204	1012	10068879	1189	**9659958**	803	10507039	**654**
FPGA-6	6960037	2772	6411247	2524	6487628	1360	**5820321**	**1060**
FPGA-7	10248020	2170	10040562	2429	10104837	1373	**9508871**	**1167**
FPGA-8	8874454	1426	8113483	2232	**7879022**	1321	8126475	**1210**
FPGA-9	12954350	2683	13616625	2925	12369055	2061	**11710738**	**1583**
FPGA-10	8564363	5555	8866049	3503	8794515	2626	**6836300**	**1587**
FPGA-11	11226088	3636	10834629	2900	**10196038**	1671	10260312	**1483**
FPGA-12	8928528	9748	8246410	4338	7755443	2335	**7224371**	**1942**
Norm.	**+11.83%**	**2.58X**	**+8.08%**	**2.08X**	**+4.24%**	**1.15X**	**0.00%**	**1.00X**

place contest winners RippleF2.0 [10], UTPlaceF2.0 [4], and UTPlaceF3.0 [5]. It is clear that GPlace-flat outperforms all placers in routed wirelength on the majority of the benchmarks (8 out of 12 benchmarks) and achieves the best overall routed wirelength and runtime. All runtime results reported in this paper for RippleF2.0 [10], UTPlaceF2.0 [4], and UTPlaceF3.0 [5] are based on binaries received from the authors of [5], [4], and [10].

GPlace-flat outperforms RippleF2.0 [10], UTPlaceF2.0 [4], and UTPlaceF3.0 [5] in routed wirelength by 11.83%, 8.08%, and 4.24%, respectively. In terms of runtime, GPlace-flat is about 2.58X, 2.08X, and 1.15X faster than RippleF2.0 [10], UTPlaceF2.0 [4], and UTPlaceF3.0 [5], respectively.

Looking more closely at the results in Table 5.9, we can observe that for **FPGA-10 benchmark**, GPlace-flat outperforms UTPlaceF3.0 [5] by 22.27% in routed wirelength, while for **FPGA-05 benchmark**, UTPlaceF3.0 [5] outperforms GPlace-flat by 8.06% in routed wirelength.

To understand this difference in performance, we first remind the reader of the relevant features of the two benchmarks. FPGA-05 is a small-to-medium size benchmark with 174K FFs, 264K LUTs, and 1281 control sets. It is also a highly-connected circuit, as witnessed by its high Rent exponent (i.e., 0.8). FPGA-10, on the other hand, is a large benchmark with 600K FFs, 264K LUTs, and 2541 control sets. However, its Rent exponent is lower (i.e., 0.6) indicating that it is less connected compared to FPGA-05.

In general, the high number of control sets in FPGA-10 significantly affects how FFs can be packed into slices without violating legality rules. This is problematic for UTPlaceF3.0 [5], because the FIP used to guide packing does not handle control-set constraints. This causes the packer to group FFs that may be physically far apart just to achieve a feasible packing. The result is a packed netlist that may be ”wirelength unfriendly” from the start. In contrast, GPlace-flat avoids this problem by not performing (early) packing, and optimizing globally (in Phases I and II) while enforcing all legalization constraints. Moreover, in Phase III of the GPlace-flat flow, the lesser connectivity in FPGA-10 allows DOISM to more easily improve the final placement of FFs and CLBs with an eye towards optimizing wirelength and routability.

In the case of FPGA-05, the fewer number of FFs and control sets makes the circuit much easier to pack into slices without violating legality rules. This fact, coupled with the fact that FPGA-05 is more connected than FPGA-10, benefits the packing approach in UTPlaceF3.0 [5]. This is because many external nets can be easily absorbed early on to produce a netlist that is less congested. In contrast, GPlace-flat, which uses cell inflation to alleviate congestion, may inflate cells more than necessary due to the abundance of interconnections, in which case DOISM may not be able to reduce the excess wirelength during detailed placement.

5.2.6 Vivado Router Runtime Comparisons

Table 5.10 compares Xilinx Vivado router runtime on placements produced by GPlace-flat to those produced by UTPlaceF3.0 [5].

Table 5.10: Vivado Router CPU Time in (seconds): GPlace-flat vs. UTPlaceF3.0 [5]

Benchmark	GPlace-flat Router Runtime(s)	UTPlaceF3.0 [5] Router Runtime(s)	Improvement (%)
FPGA-1	126	**121**	-3.97%
FPGA-2	137	**135**	-1.46%
FPGA-3	286	**255**	-10.84%
FPGA-4	320	**290**	-9.38%
FPGA-5	**2527**	3921	55.16%
FPGA-6	**425**	555	30.59%
FPGA-7	**655**	2269	246.41%
FPGA-8	442	**375**	-15.16%
FPGA-9	**739**	1094	48.04%
FPGA-10	**622**	1274	104.82%
FPGA-11	**763**	2036	166.84%
FPGA-12	**670**	882	31.64%
Total	**7712**	**13207**	**71.25%**

GPlace-flat produces placements that require on average less effort by Vivado router and can be routed on average faster by 71.25% than those produced by UTPlaceF3.0 [5]. This indicates that the quality of placements produced by GPlace-flat are less congested and of better quality than those produced by UTPlaceF3.0 [5]. Therefore, the compilation time of GPlace-flat flow on average is much better than that of UTPlaceF3.0 [5].

5.2.7 Timing Results Comparisons

Although the current implementation of GPlace-flat is non-timing driven, we include timing results in Table 5.11 and compare them with the other state-of-the-art routability-driven placers (which are also non-timing driven). Table 5.11 shows that GPlace-flat produces better timing results for most of the benchmarks compared to the other placers. In Table 5.11, the "R-WL" columns report post-routed wirelength, the "MHz" columns give the circuit speed in MHz, and the "Fmax Ra-

Table 5.11: Timing results Comparison with State-of-the-Art Academic FPGA Placers

Benchmark	RippleF2.0 [10]				UTPlaceF3.0 [5]				GPlace-flat			
	MHz	R-WL	FMax Ratio	R-WL Ratio	MHz	R-WL	FMax Ratio	R-WL Ratio	MHz	R-WL	FMax Ratio	R-WL Ratio
FPGA-1	125.0	435583	0.89	1.08	**153.8**	437373	**1.09**	1.09	140.8	**402469**	1.00	**1.00**
FPGA-2	107.5	752299	0.80	1.04	133.3	757241	0.99	1.05	**135.1**	**720714**	**1.00**	**1.00**
FPGA-3	78.1	4033650	0.95	1.16	**89.3**	3893436	**1.08**	1.12	82.6	**3473792**	1.00	**1.00**
FPGA-4	59.5	6608994	0.86	1.12	**70.9**	6043884	**1.02**	1.03	69.4	**5881605**	1.00	**1.00**
FPGA-5	38.3	10857194	0.89	0.99	40.3	**10319925**	0.93	**0.94**	**43.3**	10963581	**1.00**	1.00
FPGA-6	50.0	7542706	0.90	1.15	52.4	7806056	0.94	1.19	**55.6**	**6558411**	**1.00**	**1.00**
FPGA-7	38.6	11489394	0.84	1.13	27.9	10695627	0.80	1.05	**46.1**	**10187951**	**1.00**	**1.00**
FPGA-8	69.0	9662019	0.96	1.12	**73.0**	8818583	**1.01**	1.02	71.9	**8642408**	1.00	**1.00**
FPGA-9	31.4	14646682	0.69	1.17	42.4	13470206	0.92	1.08	**45.9**	**12486925**	**1.00**	**1.00**
FPGA-10	48.1	9122777	0.81	1.20	17.2	9650067	0.29	1.27	**59.2**	**7614759**	**1.00**	**1.00**
FPGA-11	NA[a]	-	-	-	NA	-	-	-	**40.8**	**10706748**	**1.00**	**1.00**
FPGA-12	42.9	9522550	0.87	1.22	**53.8**	9085992	**1.09**	1.16	49.5	**7820768**	1.00	**1.00**
geomean	57.0	5341755.8	0.85	1.12	57.3	5166029.2	0.86	1.09	**66.7**	**4751988.1**	**1.00**	**1.00**

[a]NA: Clock Time period is greater than 200ns

tio" and "R-WL Ratio" columns provide the ratio of the state-of-the-art placers UTPlace3.0 [5] and Ripple2.0 [10] to GPlace-flat for maximum frequency (Fmax) and routed wirelength, respectively. The last row of the Table 5.11 gives the geometric mean across all benchmarks. From Table 5.11, we see that UTPlace3.0 [5] produces routable placements with 9% higher routed wirelength and 14% worse Fmax than GPlace-flat, while Ripple2.0 [10] produces routable placements with 12% higher routed wirelength and 15% worse Fmax than GPlace-flat. Table 5.11 shows that on benchmark FPGA11 (i.e., a high resource utilization benchmark with high Rent exponent) both UTPlace3.0 [5] and Ripple2.0 [10] produce unroutable placements, even when the circuit runs at a very low clock speed (e.g., less than 5MHz). GPlace-flat, on the other hand, produces a routable placement with Fmax equal to 40.8MHz for this benchmark.

5.2.8 Extended Experimental Results

To further demonstrate the effectiveness of GPlace-flat, experiments are conducted using an additional 360 benchmarks generated by Xilinx Inc. using Gnl [47]. For each of the 12 benchmarks listed in Table 3.1, 30 new circuits are generated by varying the original circuit's parameters (i.e., the values shown in columns 2-7 of Table 3.1). This results in a total of 372 circuits (i.e., 12+12x30) being available. (The ranges of the circuit parameters were shown earlier in Table 3.2.)

5.2.8.1 The impact of different GPlace-flat modules on routability

In this section, we use several metrics to provide a breakdown of the impact of each individual GPlace-flat module. Table 5.12 compares the solution quality obtained by using mPFGR versus WLPA to estimate congestion. Column 2 shows that when WLPA is used to estimate congestion, the Vivado router is unable to route 19 of the 372 benchmarks. However, only 6 benchmarks fail to route when using mFPGR. Column 3 shows that WLPA results in placements with 2.08% more post-routing wirelength compared to mFPGR. Finally, column 4 reveals that the Vivado router is able to route placements produced using mPFGR 1.14x faster compared with placements produced with WLPA.

Table 5.12: Performance Comparison with mPFGR vs. WLPA

Congestion Method	#Failures	Routed-WL (Norm.)	Router Runtime (Norm.)
mPFGR	6	+0.00%	1.00X
WLPA	19	+2.08%	1.14X

Table 5.13 compares the solution quality obtained when using the proposed window-based legalization (i.e., WB-Legalization) versus non-window-based legalization (that uses the entire area of the FPGA). Results obtained in Table 5.13 clearly show that when non-window-based legalization is used, the Vivado router is unable to route 17 benchmarks. However, only 6 benchmarks fail to route when WB-Legalization is used. Moreover, non-window based legalization results in placements with 2.77% more post-routing wirelength compared to WB-legalization. The Vivado router (average) runtime is very similar when using either WB-Legalization or non-window-based legalization.

Table 5.13: Performance Comparison w. and w/o Window-Based Legalization

	#Failures	Routed-WL (Norm.)	Router Runtime (Norm.)
w. WB-Legalization	6	+0.00%	1.000X
w/o WB-Legalization	17	+2.77%	0.997X

Table 5.14 compares the solution quality obtained using DOISM versus post-global placement.

The results in Table 5.14 show that DOISM reduces the number of unroutable placement solutions from 13 to 6 compared with post-global placement. Moreover, using DOISM produces placements that have 13.47% less post-routing wirelength, and the Vivado router is able to route these placements 1.08x faster compared to the placements obtained using post-global placement.

Table 5.14: Performance Comparison of DOISM vs. Post Global Placement

	#Failures	Routed-WL (Norm.)	Router Runtime (Norm.)
Post Global Placement	13	+13.47%	1.08X
Post DOISM	6	+0.00%	1.00X

5.2.8.2 Comparisons with state-of-the-art placers

GPlace-flat is compared with the state-of-the-art placers in [5] and [10] based on the following metrics: (1) number of times each placer produces a placement that cannot be routed by Vivado router, (2) the percentage of successfully placed and routed circuits, (3) the number of times each placer produces a placement with the best routed wirelength, (4) the normalized routed wirelength of other placers compared with GPlace-flat, (5) the normalized placement runtime of other placers compared to GPlace-flat, (6) the normalized runtime for the (Vivado) router compared to GPlace-flat. Table 5.15 shows that GPlace-flat fails on only 6 benchmarks, whereas RippleF2.0 [10] and UTPlaceF3.0 [5] fail on 36 and 55 benchmarks, respectively.

Table 5.15: Comparision with the state-of-the-art on 372 Xilinx Benchmarks

Placer	#Failures	Success Rate (%)	#Best	Routed-WL (Norm.)	Place Runtime (Norm.)	Route Runtime (Norm.)
RippleF2.0 [10]	36	90.32%	3	+13.97%	3.20X	1.56X
UTPlaceF3.0 [5]	55	85.22%	103	+5.19%	1.15X	1.77X
GPlace-flat	6	**98.39%**	**260**	0.00%	1.00X	1.00X

Note that all three placers fail on the same 6 benchmarks, which have very high Rent exponent. In terms of QoR, GPlace-flat produces the best routed wirelength for 260 benchmarks, whereas UTPlaceF3.0 [5] and RippleF2.0 [10] produce the best results for 103 and 3 benchmarks, respectively. GPlace-flat outperforms UTPlaceF3.0 [5] and RippleF2.0 [10] by 5.19% and 13.97% in

routed wirelength, respectively. In terms of placement runtime, GPlace-flat is 1.15X and 3.20X faster than UTPlaceF3.0 [5] and RippleF2.0 [10], respectively. The last column in Table 5.15, shows that Xilinx Vivado router is able to route placements produced by GPlace-flat 1.77× and 1.56× faster than placements produced by UTPlaceF3.0 [5] and RippleF2.0 [10], respectively. Overall, GPlace-flat is able to produce placement solutions that are less congested and of better quality than those produced by UTPlaceF3.0 [5] and RippleF2.0 [10], while requiring much less total placement-and-routing time to do so.

5.3 Summary

In this chapter, we introduced a new, analytic routability-aware placement algorithm for Xilinx UltraScale FPGA architectures. The proposed algorithm, called GPlace-flat, seeks to optimize both wirelength and routability. GPlace-flat contains several unique features including a novel window-based procedure for satisfying legality constraints in lieu of packing, an accurate congestion estimation method based on modifications to the pathfinder global router, and a novel detailed placement algorithm that optimizes both wirelength and external pin count. Despite the superiority results that are achieved by GPlace-flat comparing to the state-of-the-art placers, there is no one best flow for all benchmarks. Due to variations in circuit characteristics and FPGA architectures, as well as the different optimization strategies employed by different placers, flows that perform well on some circuits may perform poorly on others. Therefore, we present a general machine-learning framework in the next chapter, that seeks to address the disconnect between different stages of the FPGA CAD flow that adversely affect the quality of results of the implemented designs. The framework consists of a suite of techniques (Artificial Neural Network, Decision Tree, Support Vector Machine, K Nearest Neighbour, and Random Forest) to model the underlying relationship between the characteristics of circuits and the best CAD algorithm (and parameters) to use for obtaining an optimized implementation on an FPGA.

Chapter 6

Automatic Flow Selection for FPGA Placement

In this chapter, we address the disconnect between different stages of the FPGA CAD flow that often adversely affects the quality of results of the implemented designs. In particular, a machine-learning framework is presented, consisting of a suite of classification and regression techniques, to model the underlying relationship between the characteristics of circuits and the best CAD algorithm (and parameters) to use for obtaining an optimized implementation on an FPGA. The efficiency of the framework is demonstrated by applying this framework to the placement stage to recommend the best placement flow for different circuits. Additionally, the framework is used to predict various quality metrics without actually incurring the cost of performing placement and routing.

6.1 Introduction

Most of the key optimization problems encountered in the *Field Programmable Gate Array (FPGA)* design process today are NP-complete. As FPGAs continue to scale in accordance with Moore's law, so does the size of the applications targeted for them. The result is increasing runtime for key optimization stages in the *Computer Aided Design (CAD)* flow, larger amounts of data generated

and passed from one stage to the next, and pressure at each stage to generate high-quality solutions appropriate for later stages in the flow. This trend provides impetus to continue to explore other approaches that can be applied to problems throughout the CAD flow.

Machine Learning (ML) can assist a designer with decision making and CAD tools with problem solving. For example, different placement tools may use different flows, resulting in placements with different wirelength, critical-path delay, power consumption, etc. Given a new circuit to place, an ML model can be trained and used to recommend the "best" placement flow, where best may refer to runtime or *Quality-of-Result (QoR)*. Similarly, ML models can be developed to efficiently and accurately estimate important parameters such as the circuit's Rent exponent or switch utilization. This information can then be used to guide a CAD algorithm to perform intelligent and effective optimization given parameters based on these predicted costs, rather than the traditional blind search performed by many current CAD algorithms. Current algorithms do not discriminate between benchmarks when performing the search for a near optimal solution. ML can act as a recommender and guide to perform specific optimization strategies either before a given stage or during a given stage in the CAD flow.

The ML framework uses training data to learn the underlying relationship between circuits and the CAD algorithms used to map them onto a particular FPGA device. The framework does not solve the problem at an arbitrary stage in the flow. Rather, it seeks to assist the designer or the tool to solve the problem.

ML algorithms comprise a set of powerful methods that use training data to develop optimized learning models. In order to create an accurate learning model, the training data must be sufficiently vast and rich to capture the underlying behavior of the problem. The ML models employed in the framework are trained using a set of 372 circuits with a broad range of features. These circuits were generated internally by Xilinx Inc., and target an UltraScale FPGA device.

The potential capabilities of the framework are demonstrated by applying it to the *placement* stage of the FPGA flow. The main contributions of this work include the following:

1. We propose a novel ML-based framework for recommending the most appropriate place-

ment procedure to use for a new circuit. Seven different academic placement flows are considered, each based on the award-winning placer in [9], as well as the Vivado placement tool. Four different placement objectives are considered: minimizing estimated wirelength, routed wirelength, critical-path delay, and power.

2. We demonstrate the flexibility of the proposed framework by also using it to efficiently and accurately predict various quality metrics without performing placement and routing, like estimated wirelength, routed wirelength, critical-path delay, power, post-routing column and row utilization, and a circuit's Rent exponent.

It is important to emphasize that the previous applications, datasets, and models are by no means exhaustive, but they serve to show the potential capabilities of the framework.

6.2 Methodology

In this section, we present a general machine-learning framework that seeks to address the disconnect between different stages of the FPGA CAD flow that adversely affect the quality of results of the implemented designs. The framework consists of a suite of techniques (Artificial Neural Network, Decision Tree, Support Vector Machine, K Nearest Neighbour, and Random Forest) to model the underlying relationship between the characteristics of circuits and the best CAD algorithm (and parameters) to use for obtaining an optimized implementation on an FPGA. The efficiency of the framework is demonstrated by training the framework to choose between different FPGA placement flows, and also by training the framework to predict various quality metrics related to FPGA placement.

6.2.1 Overall Framework

The proposed system contains two stages: an *offline* training stage (Figure 6.1) and an *online* testing/deployment stage (Figure 6.2). The offline stage involves creating either a classification

model or a regression model. Classification models are used for recommending the $best$ flow for a new circuit, where best can refer to any objective(s) a designer may wish the flow to optimize (e.g., estimated wirelength, routed wirelength, critical-path delay, etc.). Regression models, on the other hand, are used to estimate continuous values (e.g., routed wirelength, critical-path delay, a circuit's Rent exponent, etc.).

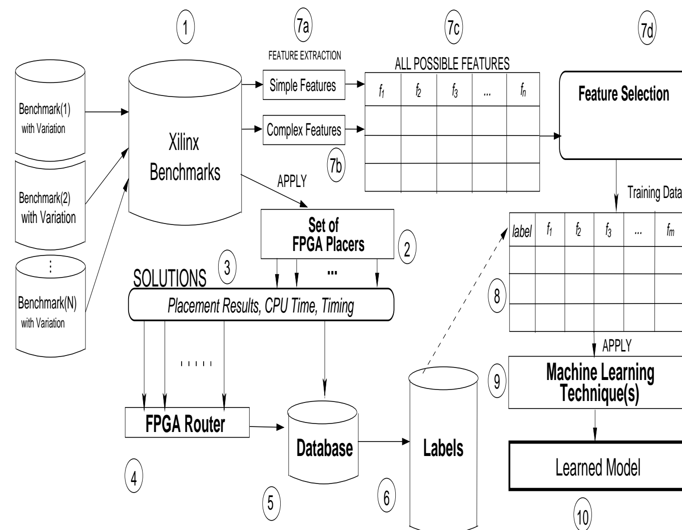


Figure 6.1: ML EDA Framework: Training

The important steps include:

1. Acquiring a set of relevant benchmark circuits for training the system (Step 1, Figure 6.1).

2. Identifying a suitable set of CAD flows (e.g., placement and routing flows) (Step 2) among existing flows.

3. Preparing the training data by running each CAD flow (Step 3 and possibly Step 4) on each benchmark to determine the result of the best flow for each benchmark for the various objectives that a designer may wish to consider. These values are used as class labels (Steps 5 & 6) for the records that make up the training data.

4. Automatically extracting features from the benchmarks (Steps 7a, 7b, 7c) that will allow the classification or regression models to learn the relationship that exists between the structure

of the circuits and the desired output (e.g., best placement flow, estimated power consumption, etc.).

5. Selecting features that can be used to mitigate the problem of dimensionality (Step 7d) if the original number of features is either too large to handle or many of the dimensions are highly correlated, and hence redundant.

6. The final features and computed class labels for each circuit form the training data (Step 8). These are used as inputs to various machine-learning techniques (Step 9) to train the classification and regression models (Step 10).

Given a new circuit to place (Figure 6.2), the online stage will perform the following steps:

1. Extract the feature values for the new circuit.

2. Use the trained model to either recommend the best placement flow to use (based on the objective(s) to be optimized), or to estimate an objective value.

3. Run the ML framework on the circuit.

4. Add the circuit to the offline stage's database of known circuits, enabling the framework's performance to further improve as it gains experience.

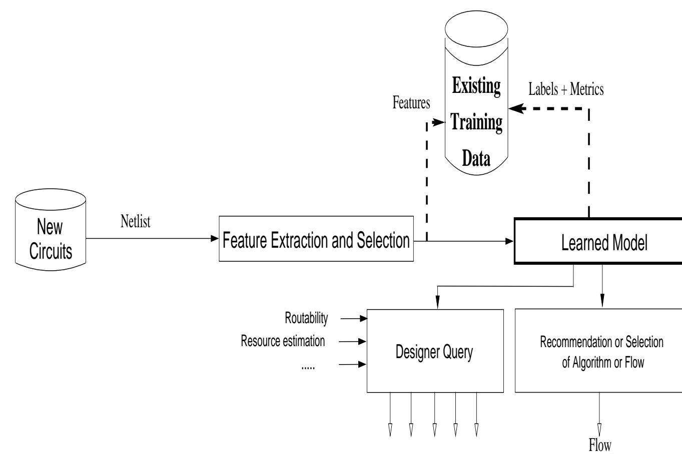


Figure 6.2: ML EDA Framework: Testing & Updating

Table 6.1: Range of Parameters for 372 Circuits

#LUTs	#FF	#BRAM	#DSP	#CSet	#IO	R.E
44K-518K	52K-630K	0-1035	0-620	11-2684	150-600	0.4-0.8

An important consideration in learning the target function from the available training data is how well the developed framework and model generalizes to new data. Generalization requires having a sufficiently large and broad set of data that reflects the complexity of the CAD problem to be modeled. Acquiring this data a priori may not be feasible. Thus, Step 4 (above) is crucial, as it *allows the circuit database to be updated over time with information about new circuits.*

6.2.2 Benchmarks used for Training

The proposed framework employs *supervised* learning algorithms. These algorithms require *training data* to learn the relationship that exists between the model's inputs (e.g., features of a new circuit) and its predicted output (e.g., the best placement flow for the circuit). Creating a prediction model that reflects the underlying behavior to be modeled typically requires a rich, large training set. Moreover, the academic placement flows that we employ in this chapter also require each circuit in the benchmark suite to be in enhanced bookshelf format [8]. As we were not aware of the existence any such corpus targeted for Xilinx UltraScale FPGAs, we relied on our industrial partner, Xilinx Inc., to create such a corpus for us. The resulting training set consists of 372 benchmarks. Each of these circuits was generated by Xilinx Inc. using an internal netlist-generation tool based on Generate Netlist (Gnl). When synthesizing circuits, several properties of the netlists were varied among the benchmarks, including the number of LUTs, DSPs, BRAMs, and FFs. The number of control resets - which poses challenges and restrictions on the placer - and the number of fixed IOs were also varied. Finally, different Rent exponents were used to vary interconnection complexity. The overall range of circuit parameters is shown in Table 6.1.

Table 6.2: Circuit Features

Feature	Description
LUTs	total number of lookup tables
FFs	total number of flip-flops
CSETs	total number of control sets
IOs	total number of fixed I/Os
Pin Count	total number of pins in the benchmark
% of 2 pin nets	percentage of 2 pin nets in circuit
% of 3 pin nets	percentage of 3 pin nets in circuit

6.2.3 Feature Extraction

Each of the 372 benchmarks in the training set must be transformed into a representation suitable for training the classification system. An important step in this transformation involves identifying the key features or properties of a circuit that cause one circuit to vary from another. Seven *basic* features were identified, each of which can be inexpensively computed directly from a circuit's netlist representation. These features are summarized in Table 6.2. LUTs, FFs, Control Sets, and fixed I/Os are all features related to the UltraScale architecture. Therefore, the ratio of the total number of these elements to the total resources available on the FPGA is used rather than the absolute values utilized by each benchmark.

When identifying features of a circuit it is important that the features themselves be inexpensive to compute. Extracting the features summarized in Table 6.2 from a netlist typically takes 2-3 seconds when performed using commodity hardware.

6.2.4 Placement Flows

While the literature has produced many different placement flows, there is no single flow that exhibits superior performance on all possible circuits. Due to variations in circuit characteristics and FPGA architectures, as well as the different optimization strategies employed by different placers, flows that perform well on some circuits may perform poorly on others. For example, in [1] the state-of-the-art academic simulated-annealing placer, VPR, is compared with three different academic ASIC placement flows adapted for FPGA placement: Capo [55], mPL [56], and Fast-

Place [57]. Capo is based on hierarchical partitioning and clustering, while mPL and FastPlace employ non-linear and quadratic placement strategies, respectively. The comparison in [23] shows that the QoR for VPR ranges from 23% better to 28% worse, while VPR's runtimes range from 1.6x to 32x slower. A wide range in performance is also observed when the three analytic placement algorithms are compared with each other. (Of course, running a placer for longer amounts of time can potentially improve QoR, though it does not guarantee design closure can be met, nor does it help designers reach design closure faster.) Therefore, in general, a designer or placement tool may wish to select the most appropriate flow based on characteristics of the new circuit, the objective to be optimized, and features of the target FPGA device.

In some of our experiments, we employ the Xilinx Vivado placement and routing CAD tools. We also experiment with seven academic configurations, each of which is based on our award winning placement flow [9] and which is described next.

ALGORITHM 9: Placement algorithm in [9]

1. Convert net-list to graph using Star+ net model
2. Pin Propogation Pre-placement
3. $N \leftarrow 2\sqrt{|Cells|}$
4. ***While*** $N \geq 1$
5. Perform $\lfloor N \rfloor$ Wirelength Opt Iter
6. Parital Legalization
7. $N \leftarrow 0.8N$
8. ***End While***
9. Select Packing Parameters
10. Packing
11. $N \leftarrow 2\sqrt{|Sites|}$
12. ***While*** $N \geq 1$
13. Perform $\lfloor N \rfloor$ Wirelength Opt Iter
14. Legalization
15. $N \leftarrow 0.8N$
16. ***End While***
17. Local Refinement

The placer in [9] is an analytic placer that minimizes total estimated wirelength, while also seeking to lessen any congestion in the final placement. (The latter is important when seeking to obtain a successful result during the subsequent routing stage.) Algorithm 9 shows a simplified template for the placer. Before the placer itself is executed, the circuit's netlist is converted to a graph representation based on the Star+ wirelength model (line 1). Then the cells in the netlist are initially pre-placed (line 2) by propagating the (fixed) positions of the I/O cells throughout the netlist and placing cells in locations near to their neighboring I/O pads. This is followed by the

execution of a while loop (lines 4-8) to optimize wirelength. On each loop iteration, Star+ wirelength cost is minimized using a Jacobi solver (line 5) and the placement is partially legalized (line 6) using bi-partitioning. The resulting placement is only partially legalized at this step, because any hard constraints related to control sets or the sharing of LUTs are not, yet, considered.

Once the while loop terminates, the location of each cell is used to make judicious decisions when performing packing.

The seven placement configurations differ based on the parameters that are used to control the flow (line 9). These parameters control a variety of options related to the packing of slices, such as the maximum number of LUTs and FFs that can be initially packed within a slice, the maximum number of external input and output signals associated with an individual slice, the maximum distance between LUTs and FFs when determining which LUTs and FFs to consider for packing into the same slice, the amount of LUT sharing to allow, whether or not to allow LUT and FF alignment to happen prior to packing, and whether or not to allow unrelated FFs and LUTs to appear in the same slice. All of these parameters have a significant affect on the quality-of-result achieved by the various flows. For example, a slice can have up to 105 external input and output signals and by reducing this number we can avoid overpacking in routing congestion regions. In other cases, reducing the number of external input and output signals of the slice in uncongested regions, unnecessarily depopulates the slices and increases the wirelength. Choosing the maximum distance to allow packing LUTs and FFs is a trade off between fitting the design in the FPGA area and the quality of the placement solution. Long distance of packing disturbs the solution of the flat placement but makes the design use a smaller number of slices that may fit into the FPGA area. In the seven placement configurations the maximum packing distance varies from 1 to 12 while the number of external input and output signals varies from 20 to 105. During packing (line 10), LUTs, FFs, and control sets are combined into clusters of logic, called *slices*. The packing process begins by dividing the FPGA into a regular grid (or set of bins). Within each bin, an arbitrary cell is selected as a seed to start the cluster. LUTs, FFs, and control-sets are then added to each cluster in a way that guarantees all of the hard constraints associated with LUT sharing, control sets, and

slice capacity are satisfied. With a fully legal placement as input, a second while loop (lines 12-16) is used to further optimize wirelength and maintain a legal placement.

As a final optimization step, a local optimization (line 17) is performed to further improve wirelength by moving a cell from its current position to an improving, neighboring position, assuming that the move does not violate any of the hard constraints imposed by the architecture.

6.2.5 Training the Classification System

The framework illustrated in Figure 6.1 supports supervised learning models, like *Artificial Neural Networks (ANNs)*, *Decision Tree (DT)*, *Support Vector Machine (SVM)*, *K Nearest Neighbor (KNN)*, and *Random Forest (RF)*. These classifiers represent five of the most popular and effective classifiers reported in the literature. These models are trained using records from the training data. Each record is represented by a tuple (S,C), where S is a set of attributes (e.g., a set of circuit features) and C is a class label. In a classification model, class labels are discrete attributes to be predicted (e.g., the identity of the best flow for placing a circuit, or the routability of a placed circuit). Continuous class variables are used for regression models, where the goal is to predict a continuous attribute (e.g., post-routing wirelength or a circuit's Rent exponent). Prior to training a learning model, each benchmark is passed through the steps of feature extraction (to create S) and placement and/or routing (to produce C), as illustrated in Figure 6.1. Once the training data is collected, each record in the training data is assigned a class label. The learning model is then trained to create either a classification model or a regression model depending on what the designer or CAD tool wishes to predict.

6.2.6 Evaluation Criteria

To obtain an unbiased estimate of a classification or regression model's performance, k-fold cross validation can be used [58]. This popular method involves dividing the data (e.g., 372 benchmarks) into k equal-sized partitions or folds. The model is then trained on $k-1$ folds and tested using the remaining fold. The procedure is then repeated k times to ensure that each fold has been used for

testing exactly once; the total error is computed by summing the errors for each of the k runs [58]. In our experiments, the performance of classification models is determined based on their cross-validated $accuracy$. The accuracy of a classification model is a measure of how well the classifier is able to make correct predictions and is defined as follows:

$$Accuracy = \frac{T_N + T_P}{T_N + T_P + F_P + F_N} \tag{6.1}$$

where T_P, T_N, F_P and F_N are the number of True Positives, True Negatives, False Positives, and False Negatives, respectively.

The Average Error (Err_{avg}), as defined below, and the standard deviation of the error values (Err_{σ}) are used to evaluate the performance of the regression models:

$$Err_{avg} = \frac{1}{n} \sum_{i=1}^{n} \frac{|x_i - y_i|}{|y_{max} - y_{min}|} \tag{6.2}$$

where x_i is the estimated attribute of circuit i, y_i is the actual value of the attribute for circuit i and is given by the CAD tool, and y_{max} and y_{min} are the maximum and minimum values, respectively, given by the CAD tool.

6.3 Results

In the following subsections, we introduce our experimental setup, followed by detailed experimental results for several different classification and regression models.

6.3.1 Experimental Setup

Experiments were carried out using a Linux machine running on a Xeon 3.2GHz processor with 16GB memory. The Scikit [59] Python module was used to implement the various classification and regression models. All routing was performed by Vivado (version 15).

Table 6.3: Predicting Best Flow: WL Placement

	Flow 1	Flow 2	Flow 3	Flow 4
Total	112/113 **(99%)**	44/46 **(95%)**	59/67 **(88%)**	141/146 **(96%)**

6.3.2 Framework Outcomes

6.3.2.1 Classification: Predicting Total Wirelength

Given a new circuit to place, the framework illustrated in Figure 6.2 is first used to predict which of the seven academic placement flows (discussed in Sec. 6.2.4) will best minimize total (estimated) wirelength. To create the necessary training data, feature extraction was performed on all 372 benchmark circuits, as described in Section 6.2.3. Each circuit was then run through each of the seven academic placement flows and, in each case, the total (estimated) wirelength was collected. Each circuit was then assigned a class label based on the identity of the flow (i.e., Flow 1-7) that produced the smallest wirelength estimate. Finally, the training data was used to train classification models based on SVM, KNN, RF, ANN and DT, respectively.

Ten-fold cross validation was used during the testing phase. Overall, the RF classifier was found to have the highest accuracy among all five classification models. Table 6.3 shows the number of flows accurately predicted by the RF classifier.

Notice that only four of the original seven flows (i.e., Flows 1, 2, 3 and 4) were recommended. Flows 5-7 were not recommended because in every case they produced higher wirelength estimates. Two observations can be made from these results. First, no one flow results in the smallest estimated wirelength for all circuits. Flow 1 is best for 113 circuits, while Flows 2, 3, and 4, are best for 46, 67, and 146 circuits, respectively. Second, the framework is able to achieve a high average accuracy of 95%. Moreover, the accuracy remains high (i.e., 88% to 99%) for the four individual flows despite the relatively large class imbalance that exists between the flows due, in part, to the diversity of the original benchmark suite.

With regards to QoR, it is important to note that the estimated wirelength, on average, increases by 24.33% when the models fail to predict the optimal flow, but one of the remaining three flows.

Table 6.4: Predicting Best Flow: WL Routing

k	Flow 1	Flow 2	Flow 3	Flow 5	Flow 6
Total	88/90 (**98%**)	9/10 (**90%**)	16/17 (**94%**)	15/15 (**100%**)	44/46 (**95%**)

This difference in QoR provides some justification for choosing between flows on a circuit-by-circuit basis.

6.3.2.2 Classification: Predicting Other Objectives

The proposed framework is flexible, and can be used to recommend a best flow for any objective that may interest a designer. Table 6.4 shows the result of repeating the previous experiment, where this time the goal is to recommend a flow that will produce the best *routed* wirelength. The training data for creating the classification models was generated as before, except that this time the placements produced by the seven academic flows were also passed through the Vivado router to determine the actual routed wirelength for each circuit.

According to Table 6.4, only 178 of the original 372 benchmarks were routable. (This result is not surprising, given the challenging nature of many of the benchmarks.) Notice that Flows 1, 2, and 3 are recommended when predicting the best flow based on estimated wirelength (Table 6.3) and the best flow based on routed wirelength (Table 6.4). However, Flow 4 and Flow 7 are not recommended when the objective is routed wirelength (Table 6.4), and Flows 5 and 6 are not recommended when the objective is estimated wirelength (Table 6.3). This is because for many benchmarks excessively optimizing wirelength during placement results in congested regions in the final placement. These congested regions, in turn, cause the router to fail. Therefore, although Flow 4 is useful if a designer wants to optimize estimated wirelength, it is not suitable when the designer's goal is to optimize routed wirelength. In the latter case, either Flow 5 or Flow 6 is recommended, because neither placer optimizes the estimated wirelength as much as Flow 4, thus avoiding congested placements and allowing the circuits to be successfully routed.

Overall, the framework is able to achieve an average accuracy of 94%, and the accuracy remains high (i.e., 90% to 100%) for the five individual flows. With regards to QoR, routed estimated

Table 6.5: Predicting Best Flow: Timing

	Flow 1	Flow 2	Flow 3	Flow 4	Flow 5
Total	16/17 (**94%**)	8/9 (**89%**)	13/17 (**76%**)	19/19 (**100%**)	116/116 (**100%**)

Table 6.6: Predicting Best Flow: Power

	Flow 4	Flow 5	Flow 6	Flow 7
Total	9/11 (**82%**)	139/140 (**99%**)	10/10 (**100%**)	16/17 (**94%**)

wirelength, on average, increases by 12.52% when the models fail to predict the optimal flow, but one of the remaining four flows. Again, this provides justification for selecting a flow on a circuit-by-circuit basis.

As another example, Table 6.5 shows the recommendation of the best flow based on critical-path delay. The average accuracy achieved to predict the critical path delay from Table 6.5 is 89.75%. Critical-path delay, on average, increases by 14.55% when the model fails to predict the optimal flow.

Table 6.6 presents the selection of the best flow based on power consumption. The average accuracy achieved to predict the power consumed in this case is 93.75%. The reported power consumption, on average, increases by 12.23%, on average, when the models fail to predict the optimal flow.

6.3.2.3 Regression: Predicting Wirelength

The previous classification models all seek to predict a discrete attribute - the identity of a placement flow. We now show how the proposed framework can employ regression models to predict continuous attributes. In this subsection, four regression models are developed: (i) estimates placement wirelength when the training set goes through the original placer in [9], (ii) estimates placement wirelength when the training set goes through the Vivado placer, (iii) estimates routed wirelength when the training set is goes through the placer in [9] followed by the Vivado router, and (iv) estimates routed wirelength when the training set goes through the Vivado placer followed by the Vivado router. The ability to accurately and efficiently estimate wirelength means that it is

Table 6.7: Regression Models: Linear vs. RF

CAD flow	Linear Model		Random Forest	
	Err_{avg}	Err_{σ}	Err_{avg}	Err_{σ}
(i) Placer in [9]	61.37%	10.72%	1.21%	1.89%
(ii) Xilinx Placer	64.14%	1.28%	1.28%	1.88%
(iii) Routed [9]	41.86%	7.36%	0.86%	1.42%
(iv) Routed Xilinx Placer	53.81%	1.1%	1.10%	2.07%

possible to provide earlier stages in the flow, like synthesis, with information that can be used to guide the optimization methods employed in those stages, while avoiding the prohibitive cost(s) of performing placement and routing.

The training data used to create the regression models was generated using a similar process to that described in Sec. 6.3.2.1. Circuit features were extracted from all 372 benchmarks. Each benchmark was then passed through the placer in [9], the Vivado placer, and the Vivado router[1] to determine the actual wirelength values. These values were then used as class labels to complete the training data. For each of the previous five cases, SVM, KNN, RF, ANN, and DT regression models were trained. Finally, 10-fold cross validation was used when evaluating and comparing the performance of the different regression models.

In all cases, the RF regression models were found to have the best performance. Table 6.7 shows both the average and standard error values achieved by the four RF models. (The individual performances of four linear regression models are also included in Table 6.7 to show that the underlying behavior of the CAD tools is non-linear; thus more powerful learning models are required.) It can be seen that all the average error values are very small and range between 0.86% and 1.21%. Unlike the placement wirelength estimation model in [42] that targets a homogeneous, island-style architecture, all of the previous models target a modern heterogeneous (UltraScale) device.

Table 6.8 shows the runtimes for the four RF regression models. Columns 2 and 3 show the total training times (performed offline) and testing times (performed online) for each model. It can be seen that all of the runtimes are very small. Column 4 shows the speedup that results when

[1]Placement solutions that failed to route were not used in cases (iii) and (iv)

Table 6.8: Regression: Runtimes and Speedup

CAD Flow	Time (sec.)		Speedup
	Training	Testing	
(i) Placer in [9]	0.3193	0.0040	158 $\times 10^3$
(ii) Xilinx Placer	0.3333	0.0041	23 $\times 10^6$
(iii) Routed [9]	0.2619	0.0038	30 $\times 10^6$
(iv) Routed Xilinx Placer	0.2462	0.0038	56 $\times 10^6$

Table 6.9: Performance of Other Regression Models

Attribute	RF Model		Runtime (sec.)	
	Err_{avg}	Err_{σ}	Training	Testing
Critical-Path Delay	3.59%	3.78%	0.3017	0.0039
Power	0.38%	1.77%	0.2782	0.0038
Rent Exponent	0.50%	1.80%	0.2849	0.0038
Row Utilization	1.39%	2.76%	0.5493	0.0063
Column Utilization	1.51%	2.22%	0.5542	0.0063
Slice Utilization	0.61%	0.94%	0.6655	0.0064

using each model to estimate wirelength rather than running the CAD tool(s) to compute the actual wirelength. For example, the average time required by the placer to produce an actual wirelength estimate is 634.006 seconds. According to column 3 of Table 6.8 the proposed framework requires an average time of 0.0040 seconds to predict and estimate wirelength. This results in a speedup of 634/0.0040 = 158,501 times (approximately 158K). The ability to efficiently and effectively predict wirelength provides opportunities to perform more effective design exploration in earlier stages of the CAD flow.

6.3.2.4 Regression: Predicting Other Attributes

Using the proposed framework, a wide variety of regression models can be created. Table 6.9 lists the results for six additional regression models.

The first estimates critical-path delay, the second estimates power consumption for a fixed clock frequency, the third estimates a circuit's Rent exponent, the fourth estimates the horizontal (row) utilization after routing, the fifth estimates the vertical (column) utilization after routing, and the sixth estimates the number of slices used. Each model was implemented using RF regression. The

training data for the first model was obtained by passing all 372 benchmark circuits through the Vivado placement and routing flow to obtain the critical-path delays. These delays were also used to compute the power consumption values (they were used to train the second model). Finally, the third model was trained using the Rent exponents that were already computed and available as a circuit feature. It can be seen from Table 6.9 that all of the error values and all of the runtimes are small. We believe that the proposed framework's ability to efficiently and effectively predict these attributes provides motivation to develop additional prediction models, for other QoR metrics and circuit attributes, that can be then used to perform more effective design exploration in earlier stages of the CAD flow.

6.4 Summary

In this chapter, we proposed a general machine-learning framework that uses training data to learn the underlying behavior between circuit features and the performance of FPGA placement tools. The efficacy of the framework was demonstrated by training the framework to select between different variants of FPGA placement flows, for different placement objectives, and also by training the framework to predict various quality metrics. The framework was trained using a relatively large corpus of 372 circuits generated internally by Xilinx and targeted for a state-of-the-art UltraScale FPGA device. Overall, the classification models used in the framework achieved average accuracies in the range of 92% to 95%, while the regression models exhibited an average error rate in the range of 0.5% to 3.6%. Given that the proposed framework is capable of quickly and accurately predicting quality metrics, like estimated and routed wirelength, critical-path delay, power, post-routing row and column utilization, a circuit's Rent exponent, and other domain-specific metrics, as part of our future work we plan to use the framework to achieve the following important objectives:

* Provide feedback to the user to modify and rewrite their HDL code such that more optimized and efficient hardware (with shorter critical path) is generated that can be placed and routed

with ease at later stages.

* Provide feedback to the front end tool (synthesizer) to create a more appropriate netlist given both the FPGA architecture and outcomes of placed circuits that are either not routable or partially congested.

* Dynamically guide the FPGA placer to use a more appropriate flow that can lead to better solutions that are routable.

* Recommending a more appropriate and suitable congestion estimation flow or legalization method to the analytic placement to achieve better measures in terms of wirelength or timing.

* Adaptively tune the parameters of the placement/routing algorithm given the circuit structure, instead of relying on the user to decide upon which parameters really matter and how to change them.

Chapter 7

Conclusions and Future Work

Most of the key optimization problems encountered in the Field Programmable Gate Array (FPGA) design process today are NP-complete. As FPGAs continue to scale in accordance with Moore's law, so does the size of the applications targeted for them. The result is increasing runtime for key optimization stages in the Computer Aided Design (CAD) flow, larger amounts of data generated and passed from one stage to the next, and pressure at each stage to generate high-quality solutions appropriate for later stages in the flow. This trend provides impetus to continue to explore other approaches that can be applied to problems throughout the CAD flow. Placement is one of the most important and time-consuming steps in the FPGA CAD flow. Given a synthesized netlist, FPGA placement is responsible for assigning all blocks onto their target physical locations in the FPGA array. As the size of FPGA designs and the complexity of modern FPGA architectures increase, sophisticated rules/constraints pose more restrictions on FPGA placement. Traditional FPGA placement flows that optimize only for wirelength without considering routability often produce poor placement that fail to route due to excessive congestion. Routability is becoming an important objective to consider in FPGA placement, witnessed by the recent ISPD 2016 Routability-driven FPGA Placement Contest.

7.1 Conclusions

This thesis sought to address the disconnect between stages of the FPGA CAD flow that often adversely affects the quality of results of the implemented designs. Several features were considered to improve the placement quality-of-results in terms of wirelength and routability such as; simultaneous packing and placement algorithms, integrating global routing within a global placement stage, and employing a machine learning framework to predict quality metrics, such as estimated and routed wirelength, critical-path delay, power, post-routing row and column utilization, and circuit's Rent exponent. The thesis was carried out in three main phases.

1. The **first phase** of the thesis involved the development of a new congestion-aware analytic placer tool for Xilinx's Ultrascale FPGA architecture. This work involved the development of placement-aware packing that incorporates physical information provided by flat initial placement to avoid packing cells that are physically far apart. A fast method for estimating congestion that is independent of the placement quality was implemented, making its use early in the placement process feasible.

2. The **second phase** of the thesis proposed a novel routability-driven analytic placer for modern Xilinx UltraScale FPGAs. The proposed placer sought to optimize both wirelength and routability. A novel window-based procedure for satisfying legality constraints in lieu of packing, a global router for accurately estimating congestion, and a detailed placement that optimizes both wirelength and routability were proposed. The performance of the placer was compared to that of i) the three ISPD 2016 Placement contest winners using the original 12 placement benchmarks, and ii) to the most recent (improved) versions of the contest winners using an additional 360 benchmarks provided by Xilinx Inc. Experimental results obtained indicated that the performance of the proposed placer was superior in terms of routed wirelength, placement runtime, and routing runtime, as well as in terms of the number of routable placements, the fewest non-routable placements, and the number of best placements.

3. The **third and final phase** of this thesis proposed a general machine-learning framework

that uses training data to learn the underlying behavior between circuit features and the performance of FPGA placement tools. The efficacy of the framework was demonstrated by training the framework to select between different variants of FPGA placement flows, for different placement objectives, and also by training the framework to predict various quality metrics. The framework was trained using a relatively large corpus of 372 circuits generated internally by Xilinx and targeted for a state-of-the-art UltraScale FPGA device. Overall, the classification models used in the framework achieved average accuracies in the range of 92% to 95%, while the regression models exhibited an average error rate in the range of 0.5% to 3.6%.

Although, we targeted the architecture of Xilinx Ultrascale VU095, which is very typical in modern FPGAs, our proposed research methodologies are also generic and can be easily adapted to other commercial families or FPGAs from other vendors.

7.2 Future Work

Despite the advances of FPGA placement in the recent works, still many challenges arise from growing complexity, diverse objectives, and high heterogeneity. Accordingly, in our future work we will attempt to pursue the following directions:

- Clock-aware FPGA Placement: Modern FPGAs contains complex clocking architecture with complicated clock legalization rules which impose a great influence on FPGA design performance and routability. Therefore, the FPGA placement problem is becoming very difficult with clock legalization constraints.

- Machine Learning for FPGA Placement: Machine learning models can provide a useful metrics prediction, like congestion, that can guide the placement process to enable high-quality results.

- Timing-driven FPGA Placement: Clock-aware timing-driven FPGA placement that consid-

ers both the locations of the clock buffers and the clock networks can be a new research direction.

- Parallel Placement for Heterogeneous FPGAs: Star+ with jacobi solver is amenable to parallelism as proved in [60], so a completely parallel GPlace tool is considered as a near future work.

Bibliography

[1] H. Bian, A. C. Ling, A. Choong, and J. Zhu, “Towards Scalable Placement for FPGAs,” in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, pp. 147–156.

[2] M. Tom, D. Leong, and G. Lemieux, “Un/DoPack: Re-Clustering of Large System-on-Chip Designs with Interconnect Variation for Low-Cost FPGAs,” in *Proceedings of the IEEE/ACM international conference on Computer-aided design*, 2006, pp. 680–687.

[3] R. Tessier and H. Giza, “Balancing Logic Utilization and Area Efficiency in FPGAs,” in *International workshop on Field Programmable Logic and Applications*. Springer, 2000, pp. 535–544.

[4] W. Li, S. Dhar, and D. Z. Pan, “UTPlaceF: A Routability-driven FPGA Placer with Physical and Congestion Aware Packing,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 66:1–66:7.

[5] W. Li, S. Dhar, and D. Pan, “UTPlaceF: A Routability-driven FPGA Placer with Physical and Congestion Aware Packing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability*. wh freeman New York, 2002, vol. 29.

[7] L. McMurchie and C. Ebeling, “PathFinder: A Negotiation-based Performance-driven Router

for FPGAs," in *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pp. 111–117.

[8] Xilinx, "ISPD 2016 Routability-Driven FPGA Placement Contest," http://www.ispd.cc/contests/16/ispd2016_contest.html, [accessed 2017-03-17].

[9] R. Pattison, Z. Abuowaimer, S. Areibi, G. Grewal, and A. Vannelli, "Invited Paper: GPlace - A Congestion-aware Placement tool for UltraScale FPGAs," in *Int' Conference on Computer Aided Design*, Austin, Texas, November 2016, pp. 1–7.

[10] C. Pui, G. Chen, W. Chow, K. Lam, P. Tu, H. Zhang, E. Young, and B. Yu, "RippleFPGA: A Routability-driven Placement for Large-Scale Heterogeneous FPGAs," in *International Conference on Computer-Aided Design*, 2016, pp. 1–8.

[11] Xilinx, ""UltraScale Architecture Configurable Logic Block User Guide"," http://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[12] J. Cong and S. K. Lim, "Edge Separability-based Circuit Clustering with Application to Multilevel Circuit Partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 3, pp. 346–357, 2004.

[13] J. Z. Yan, C. Chu, and W.-K. Mak, "Safechoice: A Novel Approach to Hypergraph Clustering for Wirelength-driven Placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 7, pp. 1020–1033, 2011.

[14] V. Betz and J. Rose, "Cluster-based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size," in *Custom Integrated Circuits Conference*. IEEE, 1997, pp. 551–554.

[15] A. S. Marquardt, V. Betz, and J. Rose, "Using Cluster-based Logic Blocks and Timing-driven Packing to Improve FPGA Speed and Density," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pp. 37–46.

[16] E. Bozorgzadeh, S. Ogrenci-Memik, and M. Sarrafzadeh, "RPack: Routability-driven Packing for Cluster-based FPGAs," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. ACM, 2001, pp. 629–634.

[17] S. T. Rajavel and A. Akoglu, "MO-Pack: Many-Objective Clustering for FPGA CAD," in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 818–823.

[18] A. Singh, G. Parthasarathy, and M. Marek-Sadowska, "Efficient Circuit Clustering for Area and Power Reduction in FPGAs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, no. 4, pp. 643–663, 2002.

[19] J. Luu, J. Rose, and J. Anderson, "Towards Interconnect-Adaptive Packing for FPGAs," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 21–30.

[20] D. T. Chen, K. Vorwerk, and A. Kennings, "Improving Timing-driven FPGA Packing with Physical Information," in *Field Programmable Logic and Applications (FPL) 2007. International Conference on*. IEEE, pp. 117–123.

[21] W. Feng, "K-way Partitioning based Packing for FPGA Logic Blocks without Input Bandwidth Constraint," in *Field-Programmable Technology (FPT), 2012 International Conference on*. IEEE, 2012, pp. 8–15.

[22] S. Brooks and B. Morgan, "Optimization Using Simulated Annealing," *The Statistician*, pp. 241–257, 1995.

[23] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *Field-Programmable Logic and Applications*. Springer, 1997, pp. 213–222.

[24] P. Maidee, C. Ababei, and K. Bazargan, "Timing-driven Partitioning-based Placement for Island Style FPGAs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 3, pp. 395–406, 2005.

[25] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, *et al.*, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, p. 6, 2014.

[26] Y. Xu and M. A. Khalid, "QPF: Efficient Quadratic Placement for FPGAs," in *Field Programmable Logic and Applications. International Conference on*. IEEE, 2005, pp. 555–558.

[27] M. Xu, G. Grewal, and S. Areibi, "StarPlace: A New Analytic Method for FPGA Placement," *Integration, The VLSI Journal*, vol. 44, no. 3, pp. 192–204, June 2011.

[28] M. Gort and J. H. Anderson, "Analytical Placement for Heterogeneous FPGAs," in *Field Programmable Logic and Applications (FPL), 22nd International Conference on*. IEEE, 2012, pp. 143–150.

[29] D. Xie, J. Xu, and J. Lai, "A New FPGA Placement Algorithm for Heterogeneous Resources," in *ASICON'09 Conference*, 2009, pp. 742–746.

[30] P. Gopalakrishnan, X. Li, and L. Pileggi, "Architecture-Aware FPGA Placement Using Metric Embedding," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 460–465.

[31] Y.-C. Chen, S.-Y. Chen, and Y.-W. Chang, "Efficient and Effective Packing and Analytical Placement for Large-Scale Heterogeneous FPGAs," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pp. 647–654.

[32] S. Chen and Y. Chang, "Routing-Architecture-Aware Analytical Placement for Heterogeneous FPGAs," in *Design Automation Conference*. ACM, 2015, p. 27.

[33] P. Spindler and F. M. Johannes, "Kraftwerk: a Fast and Robust Quadratic Placer Using an Exact Linear Net Model," in *Modern Circuit Placement*. Springer, 2007, pp. 59–93.

[34] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "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*, vol. 27, no. 7, pp. 1228–1240, 2008.

[35] L. Wang and M. Abadir, "Data Mining in EDA - Basic Principles, Promises, and Constraints," in *IEEE Design Automation Conference*, San Francisco, California, 2014, pp. 23–26.

[36] N. Kapre, H. Ng, K. Teo, and J. Naude, "InTime: A Machine Learning Approach for Efficient Selection of FPGA CAD Tool Parameters," in *IEEE Int'l Symposium on Field Programmable Gate Arrays*, Monterey, California, 2015, pp. 23–26.

[37] W. Chang, L. Chen, C. Lin, S. Mu, M. Chao, C. Tsai, and Y. Chiu, "Generating Routing-Driven Power Distribution Networks with Machine-Learning Technique," in *IEEE Int'l Symposium on Physical Design*, Santa Rosa, California, 2016, pp. 145–152.

[38] X. Xu, T. Matsunawa, S. Nojima, C. Kodama, T. Kotani, and D. Pan, "A Machine Learning Based Framework for Sub-Resolution Assist Feature Generation," in *IEEE Int'l Symposium on Physical Design*, Monterey, California, 2016, pp. 161–168.

[39] A. Mametjanov, P. Balaprakash, C. Choudary, P. Hovland, S. Wild, and G. Sabin, "Auto-tuning FPGA Design Paramters for Performance and Power," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, Vancouver, BC, May 2015, pp. 84–91.

[40] R. Manimegalai, E. Soumya, V. Muralidharan, B. Ravindran, and V. Kamakoti, "Placement and Routing for 3D-FPGAs using Reinforcement Learning and Support Vector Machines," in *IEEE International Conference on VLSI Design*, Kolkata, India, January 2005, pp. 1–6.

[41] N. Kapre, B. Chandrashekaran, H. Ng, and K. Teo, "Driving Timing Convergence of FPGA Designs through Machine Learning and Cloud Computing," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, Vancouver, BC, May 2015, pp. 119–126.

[42] Q. Liu, J. Ma, and Q. Zhang, "Neural Network Based Pre-Placement Wirelength Estimation," in *IEEE International Conference on Field Programmable Technology (FPT)*, Seoul, Korea, December 2012, pp. 16–22.

[43] M. Kurek, P. Deisenroth, W. Luk, and T. Todman, "Knowledge Transfer in Automatic Optimization Reconfigurable Designs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, Washington DC, USA, May 2016, pp. 84–87.

[44] M. Kurek, T. Becker, T. Chau, and W. Luk, "Automating Optimization of Reconfigurable Designs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, Washington DC, USA, May 2014, pp. 210–213.

[45] Q. Liu, M. Gao, and Q. Zhang, "Knowledge-Based Neural Network Model for FPGA Logical Architecture Developement," vol. 24, no. 2, pp. 664–677, February 2016.

[46] GNL, ""Netlist-Generator Tool"," http://users.elis.ugent.be/~dstrooba/gnl/.

[47] G. Grewal, S. Areibi, M. Westrik, Z. Abuowaimer, and B. Zhao, "Automatic Flow Selection and Quality-of-Result Estimation for FPGA Placement," in *24th Reconfigurable Architectures Workshop*, Orlando, Florida, USA, May 2017, pp. 115–123.

[48] T. Ahmed, P. D. Kundarewich, and J. H. Anderson, "Packing Techniques for Virtex-5 FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 3, p. 18, 2009.

[49] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "NCTU-GR 2.0: Multithreaded Collision-aware Global Routing with Bounded-Length Maze Routing," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 32, no. 5, pp. 709–722, 2013.

[50] D. Yeager, D. Chiu, and G. Lemieux, "Congestion Estimation and Localization in FPGAs:

A Visual Tool for Interconnect Prediction," in *International Workshop on System Level Interconnect Prediction*. ACM, 2007, pp. 33–40.

[51] J. Hu, J. A. Roy, and I. Markov, "Completing High-Quality Global Routes," in *ISPD*. ACM, 2010, pp. 35–41.

[52] M. Pan, Y. Xu, Y. Zhang, and C. Chu, "FastRoute: An Efficient and High-Quality Global Router," *VLSI Design*, pp. 1–19, 2012.

[53] W. How, H. Yu, X. Hong, Y. Cai, W. Wu, J. Gu, and W. Kao, "A New Congestion-driven Placement Algorithm based on Cell Inflation," in *Design Automation Conference, Asia and South Pacific*. IEEE, 2001, pp. 605–608.

[54] E. Bozorgzadeh, S. O. Memik, X. Yang, and M. Sarrafzadeh, "Routability-driven Packing: Metrics and Algorithms for Cluster-based FPGAs," *Journal of Circuits, Systems, and Computers*, vol. 13, no. 01, pp. 77–100, 2004.

[55] J. Roy, D. Papa, S. Adya, H. Chan, A. NG, F. Lu, and L. Markov, "Capo: Robust and Scalable Open-Source Min-Cut Floorplacer," in *International Symposium on Physical Design*. ACM, 2005, pp. 1–3.

[56] T. F. Chan, J. Cong, J. R. Shinnerl, K. Sze, and M. Xie, "mPL6: Enhanced Multilevel Mixed-Size Placement," in *Proceedings of the 2006 international symposium on Physical design*. ACM, pp. 212–214.

[57] N. Viswanathan and C. C. 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*, vol. 24, no. 5, pp. 722–733, 2005.

[58] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (1st Edition)*. Boston, MA, USA: Addison-Wesley Publishing Co., Inc., 2005.

[59] F. P. et al, "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078195

[60] R. Pattison, C. Fobel, G. Grewal, and S. Areibi, "Scalable Analytic Placement for FPGA on GPGPU," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015, pp. 1–6.