The Dissertation Committee for Wuxi Li
certifies that this is the approved version of the following dissertation:

Placement Algorithms for Large-Scale Heterogeneous FPGAs

Committee:

David Z. Pan, Supervisor

Nur A. Touba

Ross Baldick

Derek Chiou

Stephen Yang

Placement Algorithms for Large-Scale Heterogeneous FPGAs

by

Wuxi Li

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2019

Acknowledgments

I would like to express my deepest appreciation to my advisor, Professor David Z. Pan, for his fundamental role in my doctoral work. David is an impeccable mentor who gave me the academic freedom while consistently contributing valuable feedback, advice, and encouragement. In addition to our academic collaboration, his persistent long-distance running habit motivated me to become a better person not only mentally, but physically and spiritually. This dissertation would not have been possible without his unfailing guidance and support. David is the best advisor I can ever imagine.

I would also like to extend my sincere thanks to the rest of my committee members. I am thankful to Dr. Stephen Yang for exposing me to real-world problems that the industry truly cares about. His enthusiasm and patience made working with him enjoyable and productive. I am also grateful to Professor Nur A. Touba, Professor Ross Baldick, and Professor Derek Chiou for their kindness and support to this dissertation.

I would also like to extend my gratitude to my industrial colleagues during my internships. Lama Mouayad, Ala Qumsieh, and Dr. Abhijit Choudhury were mentors for my internship at Apple, Cupertino and Austin. They offered inspiring vision and experiences on VLSI design from designers' perspective. Dr. Wen-Hao Liu, Dr. Zhuo Li, and Dr. Charles J. Alpert were mentors for my

Above all, I am deeply indebted to my beloved wife, Muqiao Lin, for her constant support, understanding, and sacrifices. Without her, this dissertation would never have been written. I am extremely grateful to Muqiao for being my editor, proofreader, and loyal audience in all those late nights and, most of all, for being my best companion and getting me through this long journey in the most positive way possible.

Placement Algorithms for Large-Scale Heterogeneous FPGAs

Publication No. _____

Wuxi Li, Ph.D.
The University of Texas at Austin, 2019

Supervisor: David Z. Pan

In recent years, the drastically enhanced architecture and capacity of Field-Programmable Gate Array (FPGA) devices have led to the rapid growth of customized hardware acceleration for modern applications, such as machine learning, cryptocurrency mining, and high-frequency trading. However, this growing capability raises ever more challenges to FPGA placement engines. A modern FPGA device consists of heterogeneous logic resources that are unevenly distributed across the layout. This heterogeneity and nonuniformity bring difficulties to achieve smooth and high-quality placement convergences. Furthermore, FPGA devices contain complex clocking architectures to deliver flexible clock networks. The physical structure of these clock networks, however, are pre-manufactured, unadjustable, and of only limited routing resources. Conventional placement approaches without clock feasibility consideration, hence, can easily lead to clock routing failures and fail the entire FPGA implementation flow. Lastly, given the special standing of FPGAs in

fast prototyping and frequent reprogramming, its implementation time is becoming a crucial determining factor to get customers' favor. Therefore, as a runtime bottleneck of the FPGA implementation flow, ultra-fast and efficient placement engines are also in great demand.

This dissertation provides a set of placement algorithms and methodologies for large-scale heterogeneous FPGAs. To essentially improve the quality of FPGA implementation, we propose three core analytical placement engines with distinct methodologies: (1) UTPlaceF, a quadratic placer with physical-aware packing; (2) UTPlaceF-DL, a quadratic placer with simultaneous packing and legalization; (3) elfPlace, an electrostatic-based nonlinear placer. To honor the clock feasibility, we propose an efficient clock-aware placement algorithm, UTPlaceF 2.0, as well as its generalized version, UTPlaceF 2.X, which produces feasible clock routing solutions together with high-quality placement. To reduce the turn-around time of FPGA implementation, we propose an ultra-fast placement engine, UTPlaceF 3.0, which exploits the parallelism on multi-core systems. The effectiveness and efficiency of proposed approaches are demonstrated with extensive experiments on industrial-strength benchmarks.

Table of Contents

List of Tables

List of Figures

xvii

xix

Chapter 1

Introduction

The field programmable gate array (FPGA) is a type of premanufactured integrated circuit designed to be configured by designers. FPGAs are becoming increasingly attractive for their reconfigurability, shorter time-to-market, and lower non-recurring engineering costs. Beyond the success in traditional applications like fast application-specific integrated circuit (ASIC) prototyping, FPGAs have also demonstrated their applicability as hardware accelerators in modern applications, such as machine learning, cryptocurrency mining, and high-frequency trading.

The advancement of FPGA-based design methodologies is inseparable from the support of FPGA computer-aided design (CAD). A typical FPGA CAD flow, as illustrated in Figure 1.1, starts from the system specification, which characterizes the functionality of the target system. Then architectural design describes the logical behavior in either high-level synthesis languages (e.g., OpenCL [71] and Vivado HLS [29]) or behavior-level hardware description languages (e.g., Verilog and VHDL). After that, high-level synthesis, logic synthesis, and technology mapping steps translate the architectural design into a gate-level netlist. Given the results of logic design, the task of physical design
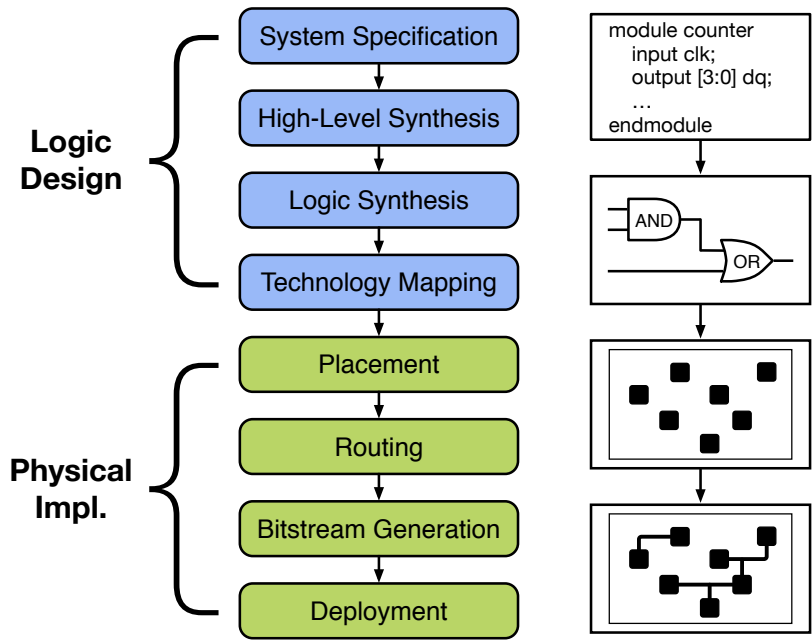
Figure 1.1: A typical FPGA CAD flow.

is to realize these abstract logical representation into geometric implementation on the FPGA fabric. Physical design contains the placement step to determine instance physical locations and the routing step to realize physical connections among instances. At last, a bitstream is generated based on the physical design results and deployed on the target FPGA device to finish the entire implementation flow.

1.1 FPGA Placement Problem

Placement, as the step to bridge logic design and physical layout, is essential to the FPGA implementation. Modern FPGA has thousands of dig-
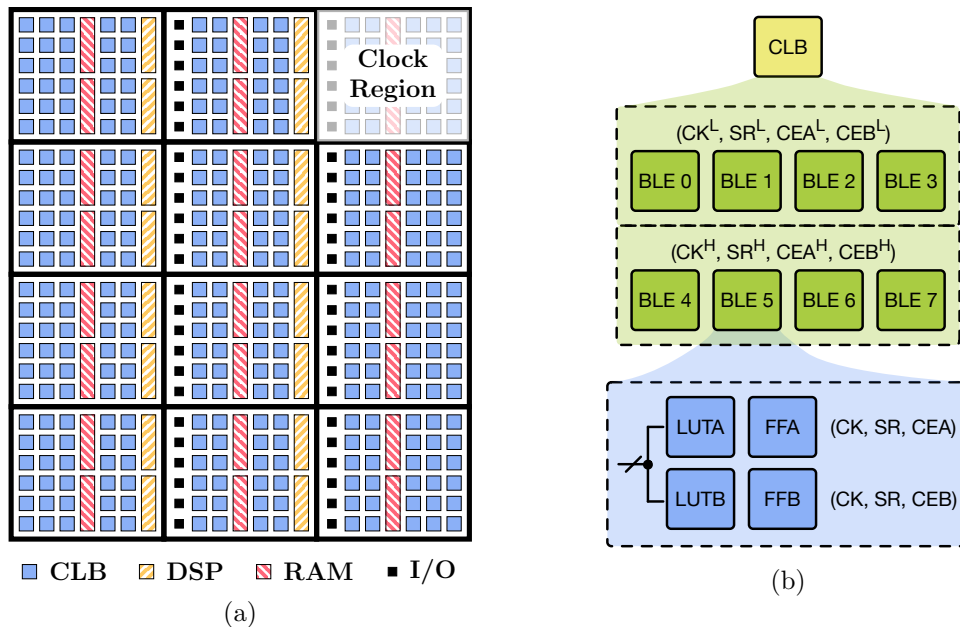
Figure 1.2: Illustrations of (a) a heterogeneous FPGA and (b) a configurable logic block (CLB).

ital signal processing (DSP) and random-access memory (RAM) blocks and millions of lookup table (LUT) and flip-flop (FF) instances. These heterogeneous resources are exclusively scattered over discrete locations on the FPGA fabric, as illustrated in Figure 1.2. The task of placement is to assign exact locations for these heterogeneous components within the FPGA layout. Placement greatly determines the overall FPGA implementation quality and efficiency. An inferior placement solution can burden the downstream routing step by producing excessive wirelength, which not only hampers the design performance but also exacerbates or even fails the implementation closure. Consequently, a favorable placement engine has to optimize various objectives to meet the design targets and ensure smooth implementation conver-

gence. Typical placement objectives include wirelength, routability, timing, and power. Minimizing total wirelength is often regarded as the major objective since it is a reasonable first-order approximation of routability, timing, and power. Besides minimizing the wirelength to reduce the total routing resource usage, it is also necessary to ensure local routing feasibility. A placer, therefore, must be able to predict and distribute routing demand in a way that the routing demand is no greater than the routing resource over all local regions in the FPGA fabric. The maximum frequency of a design is determined by its longest timing path, which usually referred as the critical path. To meet a given frequency target, a placer has to ensure the delay of the critical path is no greater than the according delay requirement. Power optimization often involves detecting and redistributing thermal hotspots to ease the cooling issue of FPGA devices.

Besides the aforementioned optimization objectives, an FPGA placer also has to honor a set of architectural constraints to produce feasible placement solutions. As shown in Figure 1.2, the majority of an FPGA layout is composed of configurable logic blocks (CLBs), where each CLB consists of a set of basic logic elements (BLEs) and each BLE further contains multiple LUTs and FFs. Due to the prefabricated and limited hardware resources in each CLB, LUTs and FFs in the same CLB must follow certain rules, usually referred as packing rules, to maintain placement feasibility. Typical packing rules include the maximum input constraint and the control set constraint. The maximum input constraint, in general, requires that the total number of

input pins in a BLE/CLB is no greater than the available pin resource. The control set constraint, on the other hand, requires that FFs in a BLE/CLB must share the same clock, set/reset, and clock enable signals. In addition to these subtle packing rules, the complex clocking architectures of modern FPGAs also impose extra constraints on placement. As shown in Figure 1.2(a), an FPGA can often be divided into a grid of clock regions (CRs), each of which has only limited routing resources dedicated to clock networks. Consequently, placement without careful clock network planning can easily lead to unroutable clocks and fail the entire FPGA implementation.

Given its complexity, the FPGA placement problem is usually decomposed into several easier sub-problems, namely, global placement, packing, legalization, and detailed placement. Global placement targets at producing a nearly-legal placement solution while globally optimizing the aforementioned placement objectives. Packing aims at addressing the packing-legality issue, which is usually not considered during global placement, by grouping LUTs and FFs into architectural-legal and placement-friendly CLBs. The execution order of global placement and packing varies from methodology to methodology, and they can also be interleaved together to achieve even smoother placement convergence. Legalization produces a feasible placement solution by locally perturbing the result of global placement and packing. Detailed placement further conducts local refinement while maintaining the solution feasibility.

1.2 Evolution of FPGA Placement

In the early age of FPGA placement, simulated-annealing (SA) approaches [7, 11] dominated industry and academic research. SA approaches iteratively perform probabilistic swapping to progressively improve placement solutions. Despite working well for small designs, it was no longer scalable as the FPGA capacity grows rapidly. Industry and academia then turned to minimum-cut approaches [56], which distribute placable instances by recursively partitioning the netlist. By leveraging the advancement in graph and hypergraph partitioning [21, 30–32], minimum-cut approaches were able to provide leading-edge performance on designs with tens of thousands of gates. As the gate count of FPGAs reached the scale of millions, analytical approaches started to steadily outperform minimum-cut approaches in both runtime and quality. Different from the combinatorial-driven SA and minimum-cut approaches, analytical approaches formulate the placement as a more sophisticated continuous optimization problem and solve it using various principled gradient descent methods. Analytical approaches can be further divided into quadratic approaches [3, 12, 24, 25, 40, 44, 77, 78] and nonlinear approaches [14, 43, 50]. Quadratic approaches approximate placement objectives using efficient quadratic functions, while nonlinear approaches use more expressive higher-order ones.

The first milestone packing algorithm in the literature, VPack [6], was proposed in the late 1990s. VPack pioneered the seed-based packing approaches. It constructs each CLB by first choosing a seed instance and it-

6

eratively growing it based on an attraction function until no more instances can be added. Motivated by the success of VPack, many seed-based packing algorithms with various optimizations, such as timing, power, and routability, were successively proposed [8,52,58,66,67,73–75]. In contrast to the bottom-up strategy of seed-based approaches, minimum-cut packing approaches adopt the top-down recursive partitioning scheme to produce packing solutions [20, 59]. However, minimum-cut approaches often cannot honor complicated packing rules and require a sequence of legalization moves to achieve architectural-feasible solutions. This made the simple yet effective seed-based approaches dominant for a decade. In the late 2000s, the concept of physical packing was first introduced to smooth the gap between packing and placement [9]. Besides considering netlist information, physical packing approaches further incorporate spatial information from initial rough placement to ease the actual downstream placement stage. As the design size increases, physical packing approaches started to demonstrate their significant advantages in solution quality over other pure-logical approaches, which made it prevalent in both industry and academic research [3, 12, 40, 68].

1.3 Challenges of FPGA Placement

Although the FPGA placement problem has been extensively studied for decades, it still remains challenging due to the drastically growing FPGA scale and complexity. The major challenges can be categorized as follows.

Improving solution quality. For modern FPGA applications, such

7

as deep learning and data center, the operating frequency and power consumption are important determining factors to occupy market share. Placement, as the first physical design step, has a great impact on the overall FPGA implementation performance, and therefore, essentially improving placement quality has always been a fundamental need. However, the increasingly complex and heterogeneous FPGA architectures have been consistently challenged the effectiveness and capability of modern placement engines.

Honoring architectural constraints. The rapid evolution of FPGA architectures also imposes extra layout constraints to placement. One such example is the clock feasibility constraint, which requires placers to guarantee the existence of feasible clock routing solutions while optimizing other objectives. These extra architectural constraints are mandatory for solution feasibility, however, their discreteness and irregularity often make them extremely difficult to handle in placement.

Enhancing runtime scalability. With the increasing FPGA scale and the growing need for fast and frequent reconfigurability, the implementation time of FPGA-based design methodologies is becoming a concern. Given the fact that placement is one of the most time-consuming optimization steps, improving algorithm scalability and runtime is another urgent demand for modern FPGA placement.

1.4 Dissertation Overview

This dissertation provides a set of placement algorithms and methodologies to tackle design challenges of large-scale heterogeneous FPGAs.

Chapter 2 explores different core placement methodologies and proposes three distinct analytical placement engines to essentially improve FPGA implementation quality: (1) UTPlaceF [40], a quadratic placer with a sophisticated physical-aware packing algorithm; (2) UTPlaceF-DL [44], a quadratic placer with a novel simultaneous packing and legalization algorithm inspired by the real-world college admission process; (3) elfPlace [43], a nonlinear placer motivated by the analogy between electrostatic system and placement.

Chapter 3 proposes two clock-aware placers that are capable of ensuring clock network feasibility while achieving satisfiable solution quality: (1) UTPlaceF 2.0 [42], an extension of UTPlaceF that resolves clock routing congestion by a novel iterative minimum-cost flow method; (2) UTPlaceF 2.X [38], a generalization of UTPlaceF 2.0 that explores an even larger clock routing solution space based on a branch-and-bound algorithm.

Chapter 4 proposes a parallelization framework, UTPlaceF 3.0 [41], for quadratic placement approaches. To achieve massive parallelism, UTPlaceF 3.0 tailors the traditional block-Jacobi preconditioning technique to the placement problem. Besides, the additive multi-grid method is also adopted to mitigate the quality degradation introduced by the parallelization. By leveraging the power of multi-core systems, UTPlaceF 3.0 demonstrates satisfiable

9

speedup with competitive solution quality.

Chapter 5 concludes this dissertation and discusses potential future directions for FPGA placement.

Chapter 2

Core FPGA Placement Algorithms

2.1 Introduction

Placement is one of the most fundamental problems in FPGA design automation. Given its significance in determining the overall implementation quality and efficiency, lots of research efforts have been devoted to the core placement algorithms over the past decades. Among all existing approaches, analytical placers, which formulate placement as sophisticated continuous optimization problems, are usually considered to be the most promising ones. Based on the modeling functions, analytical placers can be divided into quadratic placers and nonlinear placers.

Quadratic placers approximate the wirelength objective using quadratic functions [3, 12, 19, 24, 25, 77, 78], which can be efficiently minimized by solving symmetric and positive-definite (SPD) linear systems through various Krylov-subspace methods. Since pure-wirelength minimization tends to collapse cells together, spreading techniques are required for quadratic placers to reduce cell overlaps. Among all cell spreading techniques, force-directed approaches have demonstrated the most success. In each placement iteration, a force-directed spreading technique evenly spreads cells out and uses the resulting cell positions as anchor points. Then extra spreading forces directing to these anchor points are applied to help the cell spreading in the next iteration. This process is repeated until a nearly overlap-free placement is achieved.

Nonlinear placers, on the other hand, approximate placement objectives using higher-order nonlinear functions [14, 34, 50]. Different from quadratic placers, nonlinear placers formulate the nonoverlap constraint using differentiable nonlinear functions and solve it together with the wirelength in a unified objective function. Since high-order nonlinear functions are more expressive than quadratic ones, nonlinear placers can often outperform quadratic approaches in solution quality. However, this quality improvement also comes with longer runtime due to the more expensive nonlinear optimization.

Besides core placement algorithms, there are also various FPGA placement methodologies have been explored in the literature. Figure 1 summarizes several representative placement methodologies in previous works. In the early age, *Pack-Place-Legalize* approaches (the red path), such as [6–8, 20, 52, 58, 59,

12
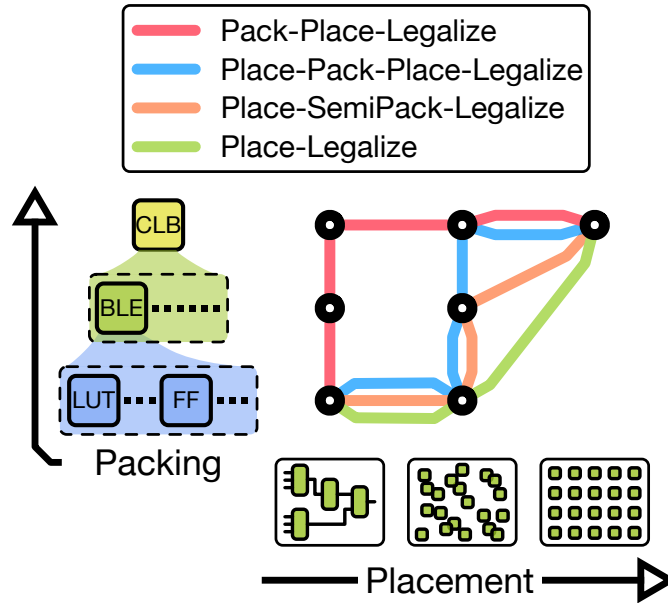
Figure 2.1: Representative FPGA placement methodologies.

66, 67], dominate industry and academic research. In this type of flow, the packing solution is first determined based on logical interconnects, then the placement and legalization are performed successively to produce a legal solution. Despite the efficiency, *Pack-Place-Legalize* approaches do not incorporate placement/spatial information during their packing decision making, thus are likely to cause poor design quality. *Place-Pack-Place-Legalize* approaches (the blue path) then emerged as a remedy to this issue [9, 34, 68]. In such a flow, a flat initial placement (FIP) is first performed. Then, both logical and spatial information is considered during the packing stage before the final placement and legalization. Another category of approaches, namely *Place-SemiPack-Legalize* (the orange path), blur the boundary between placement

and packing [12]. In particular, after an FIP similar to that in *Place-Pack-Place-Legalize* approaches, they only group LUTs and FFs into intermediate clusters (e.g., BLEs). Then, the rest of packing work and the final placement are combined into a single legalization process.

In this chapter, we propose three new analytical placement engines with different core algorithms and methodologies:

- **UTPlaceF**, a quadratic placer with the *Place-Pack-Place-Legalize* flow.

- **UTPlaceF-DL**, a quadratic placer with a novel simultaneous packing and legalization flow, as shown by the green path in Figure 2.1.

- **elfPlace**, a nonlinear placer that models density constraints using electrostatic systems.

In the rest of this chapter, Section 2.2 introduces the target FPGA architecture of these three placers. Section 2.3 describes UTPlaceF. Section 2.4 details UTPlaceF-DL, and Section 2.5 presents elfPlace.

2.2 Target FPGA Architecture

All three placers that will be proposed in this chapter are developed based on Xilinx *UltraScale VU095* [29]. It contains a representative column-based FPGA architecture that has been also adopted by many other state-of-the-art commercial FPGAs, e.g., Xilinx *UltraScale+* series. As shown in Figure 2.2(a), each column in this architecture provides one type of logic resources among CLB, DSP, RAM, and I/O. Columns of different resource types are unevenly interleaved over the FPGA fabric. Figure 2.2(b) details the CLB structure in this architecture, where each CLB consists of 8 basic logic elements (BLEs) and each BLE further contains 2 LUTs and 2 FFs. The 2 LUTs in a BLE can be either implemented as a single 6-input LUT or two smaller LUTs with a total number of distinct inputs no greater than 5. While FFs in the same CLB are subject to control set constraints. More specifically, as shown in Figure 2.2(b), a CLB can be divided into two half CLBs, and each of which consists of 4 BLEs that share the same clock (CK), set/reset (SR), and clock enable (CEA/CEB) signals. Therefore, in each half CLB, FFs must share the same CK/SR and FFs with the same polarity (FFA/FFB) must further share the same CEs (CEA/CEB).

CLB DSP RAM ▪ I/O
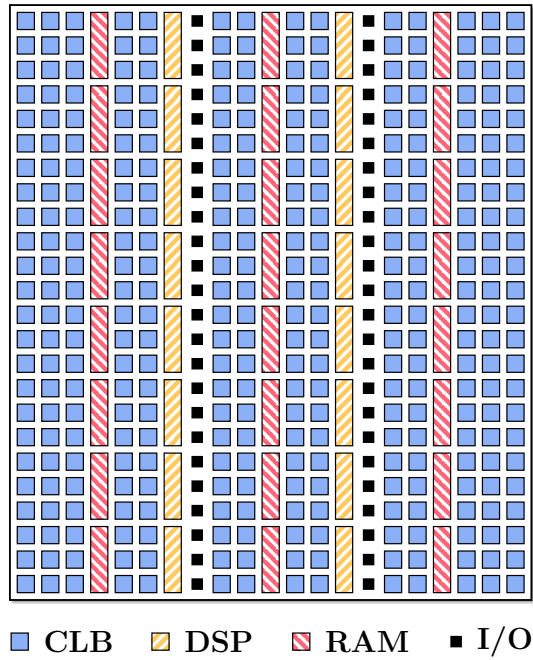
(a)
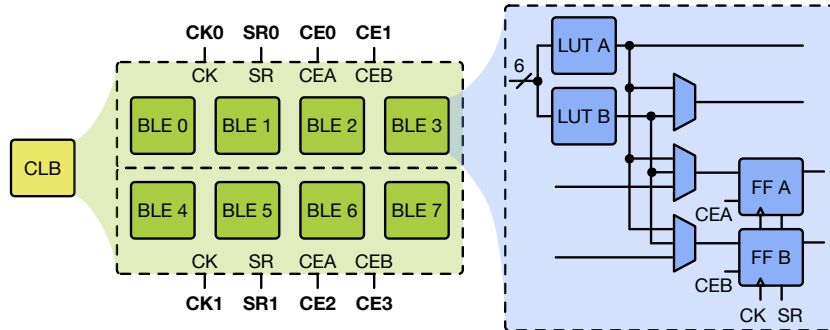
(b)

Figure 2.2: Illustration of (a) the layout and (b) the configurable logic block (CLB) structure of Xilinx UltraScale architecture.

2.3   UTPlaceF: A Packing-Based FPGA Placer

Packing-based methodologies are among the most successful FPGA placement approaches. In a packing-based placement methodology, the packing engine is responsible for grouping lookup tables (LUTs) and flip-flops (FFs) into architectural-legal configurable logic blocks (CLBs). While a placement engine is dedicated to determining the physical locations of movable cells while optimizing some cost metrics (e.g., wirelength, routability, timing, and power).

Although the packing-based placement methodologies have been intensively studied in the literature [6–8, 20, 52, 58, 59, 66, 67], existing approaches still have the following limitations:

- Existing packing algorithms do not have good knowledge of cell physical locations. Logical packing, which performs packing based only on logical connectivity, could cluster cells that are physically far apart. As a result, it may lead to wirelength-unfriendly netlists and worsen routability.

- Existing packing algorithms are unaware of actual congestion information, which is crucial for efficient and high-quality packing depopulation. Blindly applying uniform depopulation would inevitably worsen wirelength and area, and it is more efficient to only avoid overpacking in routing congested regions.

In order to remedy these deficiencies, in this section, we propose UTPlaceF, a new packing-based FPGA placers that simultaneously optimize wirelength and routability. Our main contributions are listed as follows.

17

- We propose a novel packing algorithm that incorporates accurate physical information based on a high-quality analytical global placement.

- We propose a routing congestion-aware depopulation technique to efficiently balance wirelength and routability in a correct by construction manner.

- We propose a hierarchical congestion-aware detailed placement technique to improve wirelength without degrading routability.

- We perform experiments on the ISPD 2016 routability-driven FPGA placement contest [79] benchmark suite released by Xilinx. Compared with the top-3 winners of the contest and other state-of-the-art FPGA placers, UTPlaceF achieves better routed wirelength with shorter runtime.

The rest of this section is organized as follows. Section 2.3.1 reviews the preliminaries and overviews the UTPlaceF framework. Section 2.3.2, Section 2.3.3, and Section 2.3.4 give the details of UTPlaceF packing and placement algorithms. Section 2.3.5 shows the experimental results, followed by the summary in Section 2.3.6.

2.3.1 Preliminaries and Overview

2.3.1.1 Quadratic Placement

A FPGA netlist can be represented as a hypergraph $H = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$ is the set of cells, and $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$ is the set of nets.

Let $\boldsymbol{x} = \{x_1, x_2, ..., x_{|\mathcal{V}|}\}$ and $\boldsymbol{y} = \{y_1, y_2, ..., y_{|\mathcal{V}|}\}$ be the x and y coordinates of all cells. The wirelength-driven global placement problem is to determine position vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ that minimize the total wirelength and obey the cell density constraint. Wirelength is measured by the half-perimeter wirelength (HPWL),

$$\text{HPWL}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{e \in \mathcal{E}} \{ \max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j| \}. \tag{2.1}$$

As HPWL is not differentiable everywhere, quadratic placers approximate it by squared Euclidean distance between cells. Therefore, the wirelength cost function in quadratic placer is defined as,

$$W(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2} \boldsymbol{x}^T Q_x \boldsymbol{x} + \boldsymbol{c}_x^T \boldsymbol{x} + \frac{1}{2} \boldsymbol{y}^T Q_y \boldsymbol{y} + \boldsymbol{c}_y^T \boldsymbol{y} + const. \tag{2.2}$$

2.3.1.2 UTPlaceF Overview

Figure 2.3 shows the flowchart of UTPlaceF. The overall flow is composed of four parts: (1) flat initial placement (FIP); (2) physical- and congestion-aware packing (PCAP); (3) global placement; (4) legalization and detailed placement.

The flat initial placement (FIP) is responsible for generating cell physical locations and detecting cells that are likely to be placed into routing congested regions to better guide packing. In the packing stage (PCAP), LUTs and FFs are first grouped into BLEs, then BLEs are clustered into CLBs. We assume that FIP yields the optimal cell relative position and packing should

19
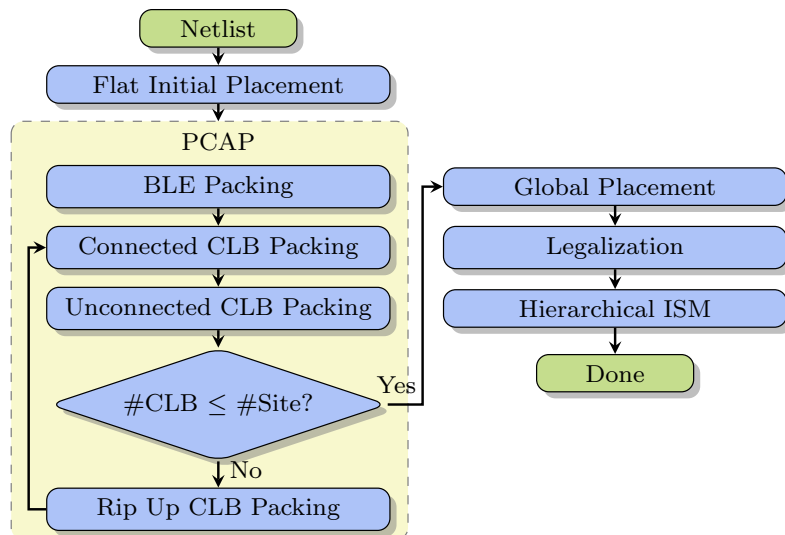
Figure 2.3: The overall flow of UTPlaceF.

not perturb it too much. Therefore, PCAP, with cell physical locations information, forbids long-distance packing and prefers close packing. Similar to iRAC [67], absorbing small nets is treated as the main objective during the packing stage of PCAP to reduce channel width and routing demand, which in turn improves wirelength and routability. Besides considering grouping connected cells, PCAP also packs unconnected cells based on their physical locations to further reduce the number of CLBs. By leveraging the routing congestion information from FIP, PCAP can perform loose packing only for cells that are likely to be placed into routing hotspots and avoid blindly depopulating throughout the whole netlist for routability enhancement. By using this congestion-aware depopulation technique, PCAP is able to achieve both good wirelength and routability.

Our global placement basically shares the same framework with FIP but handles CLBs instead of LUTs and FFs. In the detailed placement stage, a bipartite matching-based minimum-pin-movement legalization is applied first. Then a hierarchical independent set matching is performed to further reduce wirelength. To preserve the routability being optimized during the global placement, white spaces and cells in congested regions are handled specially throughout the detailed placement stage.

2.3.2 Flat Initial Placement
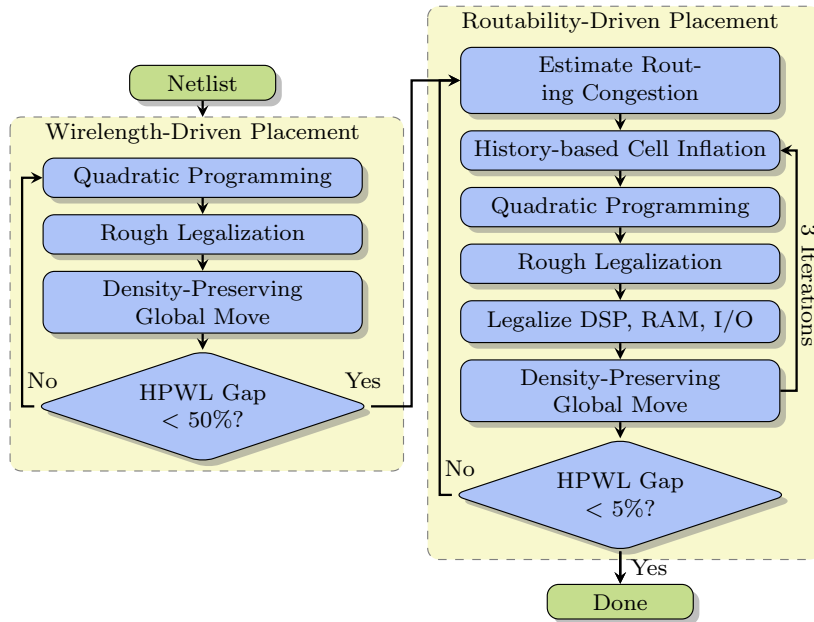
Figure 2.4: The overall flow of FIP.

Our FIP adopts the main framework of an ASIC placer POLAR 2.0 [47]. Its overall flow is shown in Figure 2.4. In each iteration of the wirelength-driven
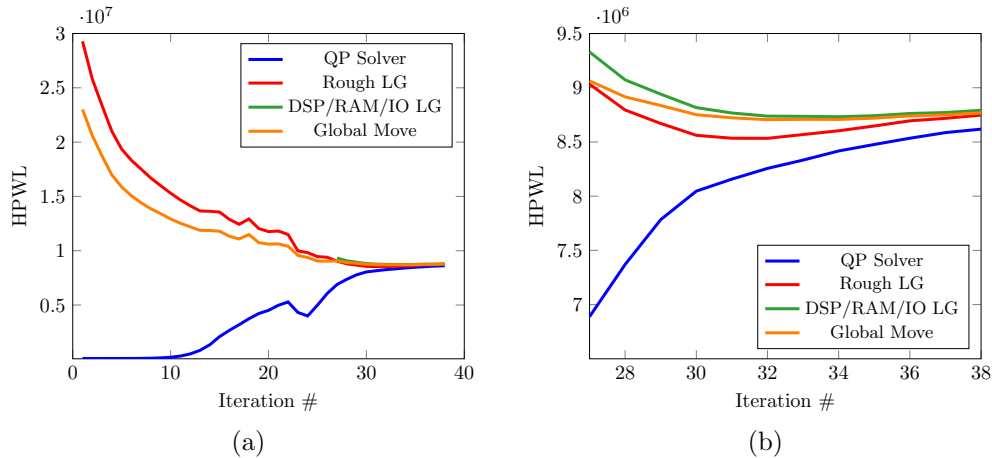
Figure 2.5: HPWL at different steps in FIP of benchmark FPGA-5. (a) All placement iterations (b) Placement iterations in the routability-driven stage.

placement, a quadratic program is solved followed by the rough legalization [33] for reducing cells overlaps. Then the density preserving global move [48] is applied to improve the wirelength of the roughly-legalized placement while preserving bin densities. A sequence of pure wirelength-driven placement iterations is performed until the gap between the upper-bound wirelength and the lower-bound wirelength is less than 50%. In the routability-driven placement stage, after a certain number of placement iterations, a fast global router NC-TUgr [53] is called for routing congestion estimation, then similar to POLAR 2.0, cells in congested regions are inflated by a small ratio and this inflation accumulates to the end of FIP. Different from the first stage, DSPs, RAMs, and I/O blocks are immediately legalized after rough legalization in this stage. Since sites for these cells are typically discrete and scattered on FPGAs, if we handle them like LUTs and FFs in rough legalization, they might be far away

from their legal positions in the final FIP solution. This discrepancy would introduce inaccuracy of cell relative positions. To eliminate this discrepancy, UTPlaceF performs an extra legalization step for DSPs, RAMs, and I/Os right after the conventional rough legalization in the second stage of FIP. By simply doing this, DSPs, RAMs, and I/Os will use their legal positions as their anchor points in placement iterations and the discrepancy will be eliminated in the final FIP solution. The legalization approach here will be further discussed in Section 2.3.4.2. Figure 2.5(a) illustrates the progression of the placement solution with respect to HPWL in different steps. A zoomed-in view of the last few iterations, with the legalization for DSPs, RAMs, and I/Os enabled, is shown in Figure 2.5(b).

The two objectives of FIP are: (1) generating cell physical locations; (2) detecting cells that are in routing congested regions. Cell physical locations are explicitly generated by the wirelength and routability co-optimized placement. Congestion information associated with cells is implicitly obtained from the history-based cell inflation. The insight of our cell inflation is to quantify the possibility of a cell lying in routing congested regions using its area – a larger cell area indicates a larger possibility of being placed into congested regions. After FIP, each cell would have a physical location and an area, which indicates the congestion level associated with it.
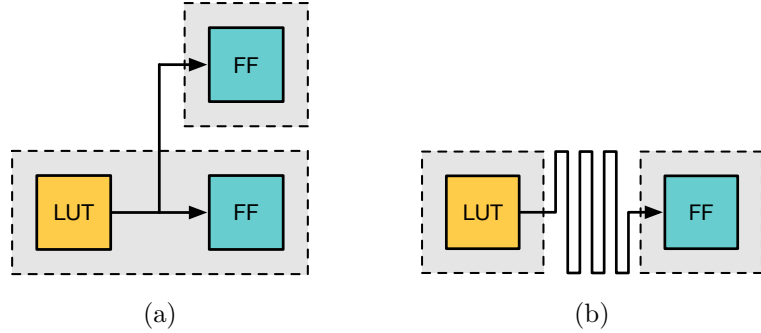
Figure 2.6: Different LUT and FF pairing scenarios. (a) A LUT fan out to multiple FFs and groups with the closest one. (b) A LUT fan out to one FF that is far away and the grouping is rejected.

2.3.3 Physical and Congestion Aware Packing

2.3.3.1 Maximum-Weighted Matching-Based BLE Packing

As a BLE in our target FPGA architecture, Xilinx Ultrascale, contains 2 LUTs and 2 FFs, existing VPack-style BLE packing algorithms cannot be directly applied. To address this new BLE architecture, we propose a two-step BLE packing algorithm that consists: (1) LUT and FF pairing; (2) LUT-FF pairs matching.

In PCAP, we apply the LUT and FF pairing in a similar manner to the BLE packing in VPack. As shown in Figure 2.6, we group each LUT to the closest FF in its fanout. Besides, any long-distance packing is rejected and only packing within *maximum packing distance of BLE* (D_b^U) is allowed. This step is mainly to make full use of fast connections between LUTs and FFs that are in the same BLEs.

In the second step, the *maximum-weighted matching* is used to finalize

the BLE packing. We build an undirected weighted graph $UWG = (\mathcal{V}, \mathcal{E})$, where each v_i in $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$ is a LUT-FF pair, a single LUT, or a single FF. $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$ represents the set of legal mergings. To apply high-attraction and close packing, we say a merging (i, j) , $\forall i, j \in \mathcal{V}$, is legal if and only if:

- i and j are connected in the netlist.

- Merging i and j into the same BLE does not violate any packing rules.

- The merging attraction is greater than the *minimum packing attraction for BLEs* (ϕ_b^L).

In the UWG, edge weights are set as merging attractions. The attraction value for a merging is defined as

$$\phi_b(i, j) = \left(1 - exp\left(\gamma_b(d_{i,j} - D_b^U)\right)\right) \sum_{e \in \mathcal{E}_i \cap \mathcal{E}_j} \frac{k_b}{|e| - 1}. \qquad (2.3)$$

where γ_b is a constant value being experimentally set as 0.2, $d_{i,j}$ is the Manhattan distance between i and j, D_b^U is the maximum packing distance of BLE which is 4 in PCAP, $\mathcal{E}_i \cap \mathcal{E}_j$ is the set of nets shared by i and j, $|e|$ is the total number of pins of net e exposed in the cluster level, and k_b is 2 for 2-pin nets and 1 for other nets.

The first term, $1 - exp(\gamma_b(d_{i,j} - D_b^U))$, is a packing distance penalty factor in the range $(-\infty, 1 - exp(-\gamma_b D_b^U)]$. This factor is very close to 1 for mergings with distances much less than D_b^U. It drops quickly as $d_{i,j}$ approaches

25

to D_b^U and becomes negative once $d_{i,j}$ is greater than D_b^U. By using the first term, short-distance mergings in PCAP are always preferred. The second term, $\sum_{e \in \mathcal{E}_i \cap \mathcal{E}_j} \frac{k_b}{|e|-1}$, is introduced to reduce the number of nets exposed in the cluster level, which in turn improves wirelength and routability. With the second term, merging two clusters sharing more small nets is granted a high priority and 2-pin nets are given even higher weights by the factor $k_b = 2$. In PCAP, the minimum packing attraction for BLEs (ϕ_b^L) is set to 0 by default.
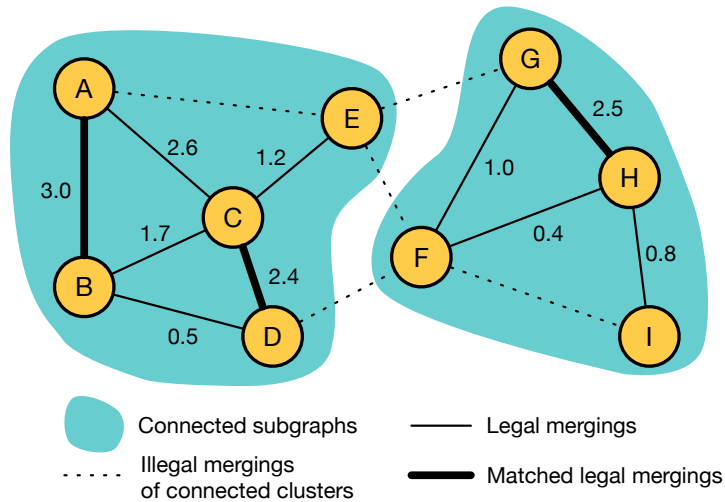
Figure 2.7: A simple maximum-weighted cluster matching example.

Figure 2.7 shows a simple cluster matching example. Due to our merging legality rules, the UWG typically is not connected and consists of many small connected subgraphs. Mergings in different connected subgraphs are independent, so PCAP performs a maximum-weighted matching algorithm on each of these subgraphs and all matched cluster pairs would be merged. The location of a merged cluster is set as the average location of all cells (LUTs

and FFs) it contains and the cluster area is simply the sum of cell areas. After each pass of matching and merging, PCAP rebuilds the UWG for clusters generated from the previous pass and solves the maximum-weighted matching again for each new connected subgraph until no more legal merging exists.

The pseudo-code of our maximum-weighted matching-based BLE packing is summarized in Algorithm 1. Each connected subgraph is built by the function `BuildConnUWG` from line 14 to line 29. Nodes and edges are added into the subgraph in a breadth-first search manner through a queue from line 20 to line 28. When the subgraph stops growing, the maximum-weighted matching would be performed on the constructed subgraph at line 7 and mergings corresponding to the matched edges would be committed to the netlist from line 8 to line 12. The loop of constructing subgraphs, solving maximum-weighted matching, and committing matched edges are iteratively executed from line 1 to line 13 and stops at line 13 when no more merging can be performed. In PACP, it is very time-consuming to consider all neighbors of cells incident to high-degree nets at line 22. So for each cell, we only consider its neighbors connected by nets with at most 16 pins.

2.3.3.2 Connected CLB Packing with Congestion-Aware Depopulation

After BLE packing, we create a CLB for every single BLE and our CLB packing is done by successively merging smaller CLBs into larger ones. CLBs that share common nets are said to be connected. In this stage, only

Algorithm 1: Maximum-Weighted Matching-Based BLE Packing

Input : FIP and LUT-FF pairing are done.
Output: BLE-level netlist with external nets reduced.

1 **while** *true* **do**
2 $numMatch \leftarrow 0$;
3 $status[u] \leftarrow Untouched, \forall u \in \mathcal{V}$;
4   **foreach** $s \in \mathcal{V}$ **do**
5   **if** $status[s] \neq Untouched$ **then continue**;
6 $g \leftarrow \texttt{BuildConnUWG}(s)$;
7   Run maximum-weighted matching on g;
8   **foreach** *matched edge* $(u, v) \in g$ **do**
9 $c \leftarrow \{u, v\}$;
10 $\mathcal{V} \leftarrow \mathcal{V} \setminus \{u, v\} \cup \{c\}$;
11 $status[c] \leftarrow Popped$;
12 $numMatch \leftarrow numMatch + 1$;
13   **if** $numMatch == 0$ **then return**;

14 **Function** $\texttt{BuildConnUWG}(s)$:
15   UWG $g \leftarrow \varnothing$;
16   Queue $q \leftarrow \varnothing$;
17   Push s into q;
18 $status[s] \leftarrow InQueue$;
19   Add s into g;
20   **while** q *is not empty* **do**
21 $t \leftarrow$ fetch and pop q top;
22   **foreach** v *connected to* t **do**
23   **if** $status[v] \neq Popped$ *and* $\phi_b(t, v) \geq \phi_b^L$ **then**
24   **if** $status[v] == Untouched$ **then**
25   Push v into q;
26 $status[v] \leftarrow InQueue$;
27   Add v into g;
28   Add edge $(t, v, \phi_b(t, v))$ into g;
29   **return** g;

connected CLBs mergings are considered.

The *BestChoice (BC)* clustering [60] is used as our main engine for connected CLB packing. In BC, attractions of all legal CLB mergings are calculated first, then the algorithm iteratively merges CLB pairs with the highest attraction based on a priority queue (PQ). The location and area of a merged CLB are set as the average location and total area of cells it contains. After each merging, the legality and attractions of mergings involving the new CLB are updated accordingly. In PCAP, a connected CLB merging (i, j) is said to be legal if and only if:

- i and j are connected in the netlist.

- Merging i and j into the same CLB does not violate any packing rules.

- The merging attraction is greater than the *minimum packing attraction for connected CLBs* (ϕ_{cc}^L).

- The total area of i and j is no greater than the *maximum CLB area* (A_c^U).

The first three rules are inherited from our BLE packing. The fourth rule is introduced to avoid overpacking in routing congested regions. Note that all cell areas used in the fourth rule are from the accumulated cell inflation in our FIP. As discussed in Section 2.3.2, cells with larger areas indicate a higher possibility to be placed into routing congested regions. By constraining the area of each CLB, PCAP would apply loose packing in routing con-

29

gested regions, and tight packing in other regions. This congestion awareness makes PCAP able to achieve a good trade-off between wirelength and routability. Figure 2.8 illustrates this congestion-aware depopulation technique where BLEs with larger areas are in routing congested regions.
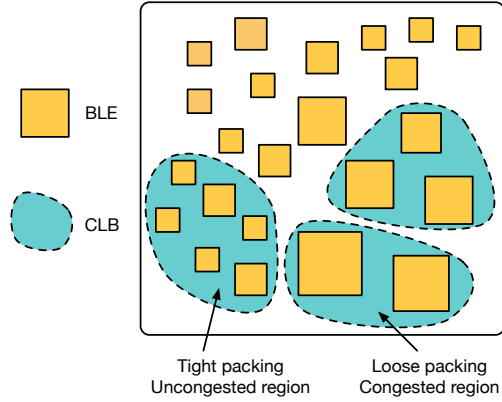
Figure 2.8: Congestion-aware depopulation of PCAP during CLB packing.

The attraction function of connected CLB mergings (i, j) is defined as

$$\phi_{cc}(i, j) = \left(1 - exp\Big(\gamma_{cc}(d_{i,j} - D_{cc}^U)\Big)\right) \sum_{e \in \mathcal{E}_i \cap \mathcal{E}_j} \frac{k_{cc}}{|e| - 1}. \qquad (2.4)$$

Equation (2.4) is basically a replica of our BLE packing attraction function defined in Equation (2.3) but differs only by some constant parameters. We experimentally set γ_{cc} to 0.2, set D_{cc}^U to 6, and set k_{cc} to 2 for 2-pin nets and 1 for other nets. The minimum packing attraction for connected CLBs (ϕ_{cc}^L) is set to 0.1 and the maximum CLB area (A_c^U) is set as 1.8 times the average CLB area by default.

Our BC-based connected CLB packing algorithm has several major
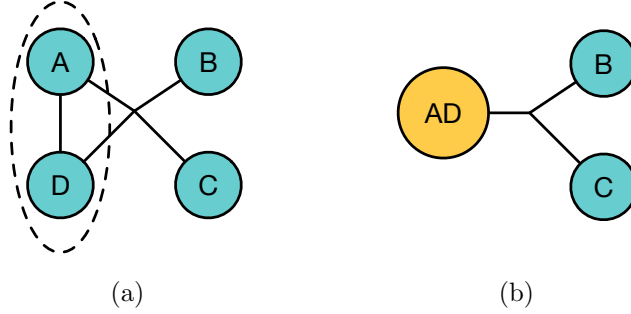
Figure 2.9: Merging a pair of clusters A and D, where $d_{B,C} = 1$, $\gamma_{cc} = 0.2$, $D_{cc}^{U} = 6$. (a) Before merging A and D, $\phi_{cc}(B,C) \approx 0.211$. (b) After merging A and D, $\phi_{cc}(B,C) \approx 0.316$.

differences compared with the traditional BC clusterings. In traditional BC, a speedup technique called *lazy update* [60] is widely adopted and this technique relies on the observation that the vast majority of attraction updates decrease their ranking in the PQ. However, this observation does not apply to our packing algorithm. Figure 2.9 shows a simple example. Before merging clusters A and D, B and C only share a 4-pin net. After the merging, the 4-pin net becomes a 3-pin net in the cluster-level netlist, as a result, the attraction between B and C increases due to the second term in Equation (2.4). We can see that this type of attraction increases can happen to any cluster pairs that share nets with more than 3 pins and all attraction updates in our BC-based connected CLB packing would increase their ranking in the PQ. Therefore, the lazy update cannot be adopted here. To guarantee the globally best merging candidate can always be fetched in each iteration, instead of only considering the best neighbor for each cluster in the traditional BC, we push all possible mergings for each cluster into the PQ and dynamically update all merging

31

attractions affected by each committed merging.

Algorithm 2: BC-based Connected CLB Packing

    Input : BLE packing is done.
    Output: CLB-level netlist with external nets reduced. Cells in
                routing congested regions are not overpacked.

1 Priority queue $PQ \leftarrow \varnothing$;
2 $valid[u] \leftarrow true, \forall u \in \mathcal{V}$;
3 **foreach** $u \in \mathcal{V}$ **do**
4   **foreach** v *connected to* u **do**
5   **if** $\phi_{cc}(u,v) \geq \phi_{cc}^L$ **then**
6   Push $(u,v,\phi_{cc}(u,v))$ into PQ
7 **while** PQ *is not empty* **do**
8 $(i,j,\phi_{cc}(i,j)) \leftarrow$ fetch and pop PQ top;
9   **if** $valid[i]$ *and* $valid[j]$ **then**
10 $m \leftarrow \{i,j\}$;
11 $\mathcal{V} \leftarrow \mathcal{V} \setminus \{i,j\} \cup \{m\}$;
12 $valid[i], valid[j] \leftarrow false$;
13 $valid[m] \leftarrow true$;
14   **foreach** k *connected to* m **do**
15   **if** $\phi_{cc}(k,m) \geq \phi_{cc}^L$ **then**
16   Push $(k,m,\phi_{cc}(k,m))$ into PQ
17   **foreach** *net* e *shared by* i *and* j **do**
18   **foreach** (p,q,ϕ_{cc}^{pq}) *such that that* p *and* q *share* e **do**
19   **if** $valid[p]$ *and* $valid[q]$ **then**
20   Update the ranking of $(p,q,\phi_{cc}(p,q))$ in PQ;

The pseudo-code of our BC-based connected CLB packing is summarized in Algorithm 2. Initially, all legal merging candidates are pushed into a PQ from line 3 to line 6. In each iteration, the merging candidate with the highest attraction is popped from the top at line 8 and is committed to the netlist from line 10 to line 11. New merging candidates involving the new cluster are pushed into the PQ from line 14 to line 16. All attractions that

are affected by the new merging are updated from line 17 to line 20. The loop from line 7 to line 20 keeps executing until no more merging candidates exist in the PQ. Similar to our BLE packing, we ignore nets that have more than 16 pins in Algorithm 2 at line 4 and line 14 to speed up the runtime.

2.3.3.3 Size-Prioritized K-Nearest-Neighbor Unconnected CLB Packing

CLBs without common nets are said to be unconnected. After connected CLB packing stage, unconnected CLB mergings are considered. Different from connected CLB packing, in which reducing external nets is the main objective, unconnected CLB packing aims at reducing the number of CLBs.

BC-based approaches typically can produce very good packing solutions for given attraction functions. However, they have an inherent drawback – the inability of making tight packing. Generally, BC would generate a large number of medium-sized clusters that are difficult to merge further due to the cluster capacity constraint. To mitigate this issue, we proposed a size-prioritized BC-based unconnected CLB packing. By assigning higher priority to mergings producing larger CLBs, medium-sized CLBs would grow quickly. As a result, much tighter packing solutions can be achieved.

Unlike the connected CLB packing in Algorithm 2, where all merging candidates are in a single PQ, we have a separate PQ for each merging size in the unconnected CLB packing. In other words, merging candidates are separated by the number of BLEs in their resulting CLBs, and only mergings

resulting in the same BLE count can be placed in the same PQ. The PQ corresponding to the largest merging size is granted the highest priority and always be processed first.

Within each PQ, BC-based unconnected CLB packing is performed in a manner similar to our connected CLB packing. For a CLB, however, instead of considering all its connected CLBs, its K-nearest neighbors (in terms of physical distances) within distance D_{uc}^U are considered in our unconnected CLB packing. Besides, a different attraction function defined in Equation (2.5) is used.

$$\phi_{uc}(i,j) = 1 - exp\Big(\gamma_{uc}(d_{i,j} - D_{uc}^U)\Big). \tag{2.5}$$

The attraction function ϕ_{uc} is a packing distance penalty factor similar to the first terms of Equation (2.3) and Equation (2.4). In UTPlaceF, we set γ_{uc} to 0.2, set D_{uc}^U to 8, and set K to 30 by default. Note that, although the objective of our unconnected CLB packing is to reduce the number of CLBs and deliver tight packing, the congestion-aware depopulation technique described in Section 2.3.3.2 is still applied in this stage to maintain good routability.

The pseudo-code of our size-prioritized K-nearest-neighbor unconnected CLB packing is summarized in Algorithm 3. Initially, all merging candidates are pushed into a PQ array ($pq[*]$) from line 2 to line 4. The merging candidates of each cluster are added by the function AddKNN from line 21 to line 27. In each iteration, among all non-empty PQs in $pq[*]$, the one corresponding to the largest merging size is fetched at line 6. The merging

34

candidate with the highest attraction in the fetched PQ is popped from the PQ top at line 7 and is committed to the netlist from line 9 to line 10. New merging candidates that include the new cluster are pushed into $pq[*]$ by the function `AddKNN` at line 14. Line 15 to line 20 guarantee that each cluster has merging candidates in $pq[*]$ if legal merging exists. The loop from line 5 to line 20 keeps executing until no more merging candidates exist in $pq[*]$.

For high-utilization designs, our default unconnected CLB packing might still not be able to generate tight enough packing solutions that satisfy the FPGA capacity constraint. In this case, the existing packing solution will be ripped up, and new connected and unconnected CLB packing parameters will be adjusted to generate a tighter packing solution in the next packing pass. PCAP iteratively performs this rip-up and repacking loop until the FPGA capacity constraint is satisfied. The details of this rip-up and repacking phase will be further discussed in Section 2.3.3.4.

2.3.3.4 Net Reduction and Packing Tightness Trade-off

Our connected CLB packing works effectively for reducing the number of external nets. However, it often produces relatively loose packing solutions due to the inherent shortcoming of BC mentioned in Section 2.3.3.3. On the other hand, our unconnected CLB packing is capable of aggressively reducing the number of CLBs and achieving tight packing solutions. Therefore, if more packing is performed in the connected CLB packing stage, a loose packing solution with less external nets would be produced. However, if we only perform

35

Algorithm 3: Size-Prioritized K-Nearest-Neighbor Unconnected CLB Packing

---

Input : Connected CLB packing is done.
Output: CLB level netlist with the total number of CLBs reduced. Cells in routing congested regions are not overpacked.

1   $pq[k] \leftarrow \varnothing, \forall k \in 2, 3, ..., N$;
2   $valid[u] \leftarrow true, \forall u \in \mathcal{V}$;
3   $numMerging[u] \leftarrow 0 \ \forall u \in \mathcal{V}$;
4   **foreach** $u \in \mathcal{V}$ **do** AddKNN (u);
5   **while** $\exists k \in 2, 3, ..., N : pq[k] \neq \varnothing$ **do**
6     $PQ \leftarrow pq[k]$ where $pq[k] \neq \varnothing$ and $pq[j] == \varnothing \ \forall j \in k+1, ..., N$;
7     $(i, j, \phi_{uc}(i, j)) \leftarrow$ fetch and pop PQ top;
8   **if** $valid[i]$ *and* $valid[j]$ **then**
9       $m \leftarrow \{i, j\}$;
10      $\mathcal{V} \leftarrow \mathcal{V} \setminus \{i, j\} \cup \{m\}$;
11      $valid[i], valid[j] \leftarrow false$;
12      $valid[m] \leftarrow true$;
13      $numMerging[m] \leftarrow 0$;
14   AddKNN (m);
15   **foreach** $u \in \{i, j\}$ **do**
16   **foreach** v *such that* $(u, v, \phi_{uc}(u, v)) \in pq[*]$ **do**
17   **if** $valid[v]$ **then**
18            $numMerging[v] \leftarrow numMerging[v] - 1$;
19   **if** $numMerging[v] == 0$ **then**
20   AddKNN (v)

21 **Function** AddKNN(u)**:**
22   **foreach** v *such that* $d_{u,v} \leq D_{uc}^{U}$ *(sorted by $d_{u,v}$)* **do**
23   **if** $\phi_{uc}(u, v) \geq \phi_{uc}^{L}$ **then**
24   Push $(u, v, \phi_{uc}(u, v))$ into $pq[numBLE(u \cup v)]$;
25        $numMerging[u] \leftarrow numMerging[u] + 1$;
26        $numMerging[v] \leftarrow numMerging[v] + 1$;
27   **if** $numMerging[u] == K$ **then return**;

a small amount of packing in the connected CLB packing stage and leave most of the work to our unconnected CLB packing, the final packing solution would be more inclined to the "tight" side with more external nets.

In PCAP, the minimum connected CLB packing attraction (ϕ_{rc}^L) and the maximum unconnected CLB packing distance (D_{uc}^U) are used to control the amount of packing work for each (connected/unconnected) CLB packing stage. Initially, ϕ_{rc}^L is set as 0.1 to aggressively reduce the number of external nets, and D_{uc}^U is set as 8 to only allow close packing in unconnected CLB packing stage. This initial setting typically results in a loose packing with a large amount of net reduction. For high-utilization designs, however, the packing solution generated by the initial setting could be sparse to the extent that the number of CLBs exceeds the FPGA capacity. In this case, PCAP will discard the existing CLB packing solution (but respect the BLE packing solution) and perform a repacking step, which applies our connected and unconnected CLB packing again. In the repacking phase, however, ϕ_{rc}^L is increased to reduce the amount of connected CLB packing and D_{uc}^U is also increased to allow longer-distance unconnected CLB packing. As a result, the repacking step can achieve tighter packing but sacrifice net reduction. The repacking step is repeated until the CLB utilization target is satisfied.

The pseudo-code of our rip-up and repacking is summarized in Algorithm 4. In UTPlaceF, we experimentally set U_{max} to 0.999, set $\Delta\phi_{rc}^L$ to 0.3, and set $\beta_{D_{uc}^U}$ to 1.414.

Algorithm 4: CLB Packing and Rip-up and Repacking
Input : BLE packing is done. Maximum FPGA CLB utilization U_{max}, maximum unconnected CLB packing distance increasing rate $\beta_{D_{uc}^U}$, and minimum connected CLB packing attraction increasing rate $\Delta\phi_{rc}^L$.
Output: The maximum FPGA CLB utilization constraint is satisfied.
1 while *true* **do**
2   Perform connected CLB packing;
3   Perform unconnected CLB packing;
4   **if** *CLB utilization* $\leq U_{max}$ **then return**;
5 $D_{uc}^U \leftarrow D_{uc}^U \beta_{D_{uc}^U}$;
6 $\phi_{rc}^L \leftarrow \phi_{rc}^L + \Delta\phi_{rc}^L$;
7 end

2.3.4 Post-Packing Placement

2.3.4.1 Global Placement

After PCAP, the global placement is performed immediately to further optimize wirelength and routability. Our global placement shares the same framework and parameter settings with FIP, but instead of optimizing the flat LUT/FF netlist, it considers each CLB as a whole. It is an incremental placement using the FIP solution as the starting point to speed up the wirelength convergence. Since the initial CLB-level placement induced from FIP is more or less close to the optimal solution, we skip the wirelength-driven phase in Figure 2.4 and directly apply the routability-driven phase to further reduce the runtime. To avoid global placement being stuck in the local optimal around FIP, the weight of pseudo-nets for cell spreading is reduced at the beginning of our global placement.

2.3.4.2   Minimum-Cost Bipartite Matching Based Legalization

A notable difference between ASIC and FPGA legalization is that ASIC standard cells have different dimensions whereas FPGA CLBs have the same size. Because of this special property, FPGA legalization problem can be formulated as a minimum-cost-maximum-cardinality bipartite matching problem with pin movement as cost. By solving the corresponding bipartite matching problem, global placement can be legalized with the minimum total pin movement. However, solving a complete bipartite matching for large designs is impractical in terms of runtime. To address this problem, we partition the placement region into a set of uniform rectangle partitions, then apply a minimum-cost bipartite matching for each partition. To further speed up the runtime, edges in bipartite graphs are pruned based on Manhattan distance and are incrementally added when necessary.

Our legalization approach is summarized in Algorithm 5. The placement region is partitioned into a set of uniform rectangle regions at line 1. For each partition, we put all cells and all unoccupied sites into two sets at line 3 and line 4. To ensure that each minimum-cost bipartite matching has enough sites to accommodate all cells, neighbor available sites are added when necessary from line 5 to line 6. The bipartite graph is constructed at line 7 with all cells as left vertices and all sites as right vertices. Edges between left vertices (cells) and right vertices (sites) are added from line 11 to line 14 and the edge pruning based on Manhattan distance is applied at the same time. The minimum-cost bipartite matching is solved at line 15 and cells are moved

Algorithm 5: Minimum-Cost Bipartite Matching Based Legalization

Input : Packing and global placement are done. Initial max displacement D_{max} and max displacement increasing rate ΔD_{max}.

Output: Legalized placement with the minimum total pin movement.

1 Partition the placement region into a set of rectangle regions P;

2 **foreach** $p \in P$ **do**

3 $L \leftarrow$ unlegalized CLBs in p;

4 $R \leftarrow$ unoccupied sites in p;

5   **while** $|L| > |R|$ **do**

6   Add the closest unoccupied site to p into R;

7   Construct a $|L| \times |R|$ bipartite graph g;

8 $d_{min} \leftarrow 0$;

9 $d_{max} \leftarrow D_{max}$;

10   **while** *true* **do**

11   **foreach** $l \in L, r \in R$ **do**

12   **if** $d_{min} \leq dist(l,r) < d_{max}$ **then**

13 $cost \leftarrow dist(l,r) \cdot numPins(l)$;

14   Add edge $(l, r, cost)$ into g;

15   Run minimum-cost bipartite matching on g;

16   **if** *number of matched edges $==$ $|L|$* **then**

17   **foreach** *matched edge (l, r)* **do**

18   Move l to r's location;

19   Mark l as legalized;

20   Mark r as unoccupied;

21   **return**;

22 $d_{min} \leftarrow d_{max}$;

23 $d_{max} \leftarrow d_{max} + \Delta D$;

to their matched sites from line 16 to line 21. If no feasible solution is found, the maximum displacement constraint is increased at line 22 and line 23, and the loop from line 10 to line 23 is repeated. In UTPlaceF, we set partition width and height to 42 and 60 CLBs, set the initial maximum displacement D_{max} to 4, and set the maximum displacement increasing rate ΔD_{max} to 2. Similar to CLBs, heterogeneous blocks like DSPs, RAMs, and I/Os also have uniform sizes, so they can be legalized by Algorithm 5 with minor variations as well.

2.3.4.3 Congestion-Aware Hierarchical Independent Set Matching

The idea of bipartite matching can also be applied to optimize wire-length. For a given set of legalized cells, a wirelength optimization problem can be formulated as a minimum-cost bipartite matching with edge weights as HPWL increase of moving cells to different sites. However, solving this matching problem cannot guarantee the optimal HPWL improvement, since the edge weight of a cell depends on the positions of other connected cells in the same matching set. To overcome this drawback, we adopt the *independent set matching (ISM)* idea from NTUplace3 [13] and only apply matching within a set of cells that do not share any nets. Besides, white spaces are also considered in our matching to further increase the solution space that we can explore.

In UTPlaceF, ISM is hierarchically applied to CLBs, BLEs and LUT pairs. The main objective of our packing stage (PCAP) is to absorb small nets

41

into clusters (BLEs, CLBs). Therefore, most CLBs essentially are clusters of
LUTs and FFs that have strong connections. One of our key observations is
that moving cells with strong connections together helps to jump out of local
optima in terms of wirelength, so ISM is applied to CLBs first in UPlaceF.
However, even though most CLBs contain strongly connected cells, they are
clustered only based on physical distance and connectivity without any wire-
length awareness. Thus, ISM for BLEs and LUT pairs are also introduced to
improve our CLB packing and BLE packing, respectively, after the CLB-level
ISM.
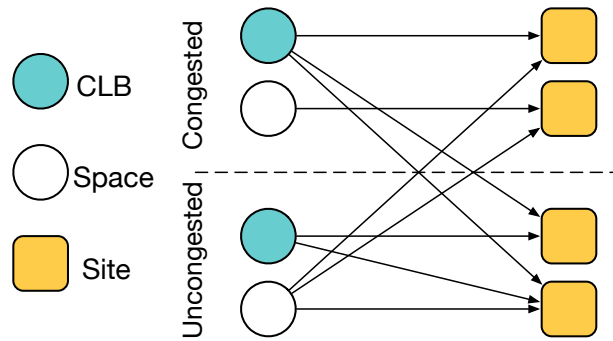
Figure 2.10: Illustration of our congestion-aware ISM.

The ISM works effectively for optimizing HPWL. However, it could ruin
the local cell density optimized for routability, especially when spaces are con-
sidered in our ISM. To mitigate this problem, we propose a congestion-aware
ISM with three extra constraints introduced: (1) cells can be moved out of but
not into routing congested regions; (2) spaces can be moved into but not out
of congested regions; (3) moves within congested regions are forbidden. Fig-
ure 2.10 shows a simple matching example with the extra constraints applied.

To get accurate congestion information, the routing congestion map is updated after a certain number of ISM iterations. By applying our congestion-aware ISM, HPWL can be optimized without routability degradation.

The pseudo-code of our congestion-aware ISM is summarized in Algorithm 6. Each independent set is generated at line 2 by the function `BuildIndepSet` from line 11 to line 18. The bipartite graph is constructed from line 3 to line 7. The cost of each edge is calculated by the function `CalcMovingCost` from line 19 to line 24 and the three extra routability rules are applied here as well. The minimum-cost bipartite matching is solved at line 8 and cells are moved to their matched locations from line 9 to line 10. This algorithm is sequentially executed for CLBs, BLEs, and LUT pairs to successively optimize wirelength at different levels. Note that, for the BLE ISM, the control set legality of CLBs must be preserved, so each independent set can only contain BLEs belong to the same control set (and may also contain BLEs without control sets). In UTPlaecF, U_{th} is set to 0.7, N_{is} is set to 50, and D_{is} is set to 10 by default.

2.3.5 Experimental Results

UTPlaceF is implemented in C++ and tested on a Linux machine with 3.40 GHz CPU and 32 GB RAM. The benchmark suite released by Xilinx for the ISPD 2016 FPGA placement contest was used to validate the effectiveness and efficiency of UTPlaceF. The executable, placement solutions, and benchmarks are released at the link `http://wuxili.net/project/utplacef/`.

Algorithm 6: Congestion-Aware Independent Set Matching

Input : A legal placement. Routing utilization threshold U_{th}, maximum independent set size N_{is}, and maximum independent set radius D_{is}.

Output: A legalized placement with shorter wirelength and no routability degradation.

1 **foreach** $u \in \mathcal{V}$ **do**

2 $S \leftarrow$ BuildIndepSet (u);

3   Construct a $|S| \times |S|$ minimum-cost bipartite graph g;

4   **foreach** $l \in$ *left vertex set of g* **do**

5   **foreach** $r \in$ *right vertex set of g* **do**

6 $cost \leftarrow$ CalcMovingCost (l, r);

7   Add the edge $(l, r, cost)$ into g;

8   Run minimum-cost bipartite matching on g;

9   **foreach** *matched edge* (l, r) **do**

10   Move l to r's location;

11 **Function** BuildIndepSet(s):

12 $isIndep[u] \leftarrow true \ \forall u \in \mathcal{V}$;

13   **foreach** $u \in \{s\} \cup \{x \in \mathcal{V} | dist(s, x) \leq D_{is}\}$ **do**

14   **if** $isIndep[u]$ **then**

15 $S \leftarrow S \cup \{u\}$;

16   **if** $|S| \geq N_{is}$ **then return** S;

17   **foreach** v *connected to* u **do**

18 $isIndep[v] \leftarrow false$;

19 **Function** CalcMovingCost(l, r):

20   **if** *l is a cell and routing utilization at r* $> U_{is}$ **then**

21   **return** ∞;

22   **if** *l is a white space and routing utilization at l* $> U_{is}$ **then**

23   **return** ∞;

24   **return** HPWL increase of moving l to r's location;

2.3.5.1 Benchmark Characteristics

Table 2.1: ISPD 2016 Placement Contest Benchmarks Statistics

Benchmark	#LUT	#FF	#RAM	#DSP	#Ctrl Set
FPGA-01	50K	55K	0	0	12
FPGA-02	100K	66K	100	100	121
FPGA-03	250K	170K	600	500	1281
FPGA-04	250K	172K	600	500	1281
FPGA-05	250K	174K	600	500	1281
FPGA-06	350K	352K	1000	600	2541
FPGA-07	350K	355K	1000	600	2541
FPGA-08	500K	216K	600	500	1281
FPGA-09	500K	366K	1000	600	2541
FPGA-10	350K	600K	1000	600	2541
FPGA-11	480K	363K	1000	400	2091
FPGA-12	500K	602K	600	500	1281
Resources	538K	1075K	1728	768	N/A

The characteristics of the ISPD 2016 benchmark suite are listed in Table 2.1. This benchmark suite has cell count ranging from 0.1 to 1.1 million, which is much larger than existing academic FPGA benchmarks. Note that several benchmarks have extremely high cell utilization, which raises two requirements to FPGA placement packing and placement engines: (1) the capability of producing tight packing solutions to satisfy the CLB capacity constraint; (2) the capability of reducing routing resource demand since little white space is available for cell and routing demand spreading.

2.3.5.2 Comparison with Previous Works

We compare our results with the top-3 winners of the ISPD 2016 placement contest and other state-of-the-art FPGA placers. The results are shown

Table 2.2: Routed Wirelength (WL in 10^3) and Runtime (RT in seconds) Comparison with the Contest Winners

Design	1st Place		2nd Place		3rd Place		UTPlaceF	
	WL	RT	WL	RT	WL	RT	WL	RT
FPGA-01	*	-	380	118	582	97	357	185
FPGA-02	678	435	680	208	1047	191	642	305
FPGA-03	3223	1527	3661	1159	5029	862	3215	831
FPGA-04	5629	1257	6497	1149	7247	889	5410	824
FPGA-05	10265	1266	†	-	†	-	9660	1237
FPGA-06	6630	2920	7009	4166	6823	8613	6488	1041
FPGA-07	10237	2703	10416	4572	10973	9169	10105	1721
FPGA-08	8384	2645	8986	2942	12300	2741	7879	1686
FPGA-09	†	-	13909	5833	†	-	12369	2537
FPGA-10	*	-	*	-	†	-	8795	3182
FPGA-11	11091	3227	11713	7331	†	-	10196	2151
FPGA-12	9022	4539	*	-	†	-	7755	2944
Norm.	1.062	1.55	1.116	2.30	1.291	3.10	1.000	1.00

*: Placement error

†: Unroutable placement

in Table 2.2 and Table 2.3. All routed wirelength is reported by Xilinx Vivado v2015.4 and the runtime of the contest winners are evaluated on a Linux machine with 3.20 GHz CPU and 32 GB RAM. The normalized results in the last row of Table 2.2 and Table 2.3 are based on comparisons with our results, and only benchmarks that other placers completed are considered in each comparison. It can be seen that UTPlaceF achieves the best overall routed wirelength. On average, UTPlaceF outperforms by 6.2%, 11.6%, 29.1%, 3.7%, 7.3%, and 16.8% in routed wirelength compared with the top-3 contest winners, [39], RippleFPGA [63], and GPlace [62], respectively. It should be noted that only UTPlaceF and RippleFPGA are able to produce routable solutions for all 12

Table 2.3: Routed Wirelength (WL in 10^3) and Runtime (RT in seconds) Comparison with State-of-the-Art Academic FPGA Placers

Design	[39] *		RippleFPGA [63]		GPlace [62]		UTPlaceF	
	WL	RT	WL	RT	WL	RT	WL	RT
FPGA-01	385	215	363	74	494	30	357	185
FPGA-02	653	399	678	167	903	61	642	305
FPGA-03	3181	1555	3617	1037	3908	289	3215	831
FPGA-04	5504	1289	6037	621	6278	280	5410	824
FPGA-05	10069	1237	10455	1012	†	-	9660	1237
FPGA-06	6411	2827	6960	2772	7643	600	6488	1041
FPGA-07	10041	2588	10248	2170	11255	691	10105	1721
FPGA-08	8113	2705	8874	1426	9323	734	7879	1686
FPGA-09	13617	3407	12954	2683	14003	974	12369	2537
FPGA-10	8866	4091	8564	5555	†	-	8795	3182
FPGA-11	10835	3267	11226	3636	12368	923	10196	2151
FPGA-12	8246	4625	8929	9748	†	-	7755	2944
Norm.	1.037	1.47	1.073	1.61	1.168	0.38	1.000	1.00

* This is the preliminary version of UTPlcaeF that was published in ICCAD 2016.

benchmarks. In terms of runtime, as all placers are evaluated on different machines, it is not fair to compare them directly. However, we still can see that the runtime of UTPlaceF is only worse than GPlace and is about 1.5× to 3.1× faster than other placers.

2.3.5.3 Runtime Analysis

The runtime breakdown of UTPlaceF is shown in Table 2.4. On average, 55.0% of the total runtime is taken by FIP, while PCAP, global placement, and hierarchical ISM take 16.2%, 5.1% and 21.4% of the total runtime, respectively, and legalization only takes 1.6% of the total runtime. PCAP is further divided into three components: BLE packing takes 0.5% of the total runtime,

Table 2.4: Runtime Breakdown of UTPlaceF

Design	FIP	PCAP			GP	LG	ISM			Others	Total
		BLE	Conn CLB	Unconn CLB			CLB	BLE	LUT Pair		
FPGA-01	115	1	2	3	9	1	5	26	22	1	185
FPGA-02	187	2	4	4	17	1	8	38	42	2	305
FPGA-03	528	5	12	13	58	1	28	57	120	9	831
FPGA-04	500	6	12	17	56	2	33	61	127	10	824
FPGA-05	663	7	13	23	77	3	41	67	136	11	1041
FPGA-06	1029	8	44	171	101	30	49	77	197	15	1721
FPGA-07	974	9	52	195	126	19	54	81	205	16	1731
FPGA-08	1004	9	26	39	91	16	49	174	274	4	1686
FPGA-09	1217	13	100	538	119	41	60	116	314	19	2537
FPGA-10	1343	11	65	1154	95	59	60	148	229	18	3182
FPGA-11	1248	12	50	133	115	23	55	203	308	4	2151
FPGA-12	1720	12	66	279	116	107	54	226	346	18	2944
Norm.	0.550	0.005	0.023	0.134	0.051	0.016	0.026	0.067	0.121	0.007	1.000

connected CLB packing takes 2.3% of the total runtime, and the remaining 13.4% is taken by the unconnected CLB packing. In hierarchical ISM, CLB, BLE, and LUT-pair ISMs take 2.6%, 6.7%, and 12.1% of the total runtime, respectively.

2.3.5.4 Effectiveness Validation of Congestion-Aware Hierarchical Independent Set Matching

To show the effectiveness of our proposed congestion-aware hierarchical ISM, we compared the routed wirelength of each intermediate placement solution after different ISM steps. Table 2.5 summarizes the comparison results. Compared with post-legalization solutions, placements after CLB ISM, BLE ISM, and LUT-pair ISM are 2.4%, 3.7%, and 5.5% shorter in routed wirelength, respectively.

Table 2.5: Routed Wirelength (WL in 10^3) at Different Stages of the Congestion-Aware Hierarchical Independent Set Matching

Design	LG WL	CLB ISM WL	BLE ISM WL	LUT-pair ISM WL
FPGA-01	407	394	359	357
FPGA-02	702	677	647	642
FPGA-03	3339	3264	3236	3215
FPGA-04	5603	5484	5433	5410
FPGA-05	9908	9697	9641	9660
FPGA-06	6824	6686	6613	6488
FPGA-07	10500	10322	10253	10105
FPGA-08	8156	7992	7913	7879
FPGA-09	13181	12937	12834	12369
FPGA-10	9916	9633	9285	8795
FPGA-11	10687	10332	10307	10196
FPGA-12	8480	8208	7956	7755
Norm.	1.000	0.976	0.963	0.945

2.3.6 Summary

With the utilization of FPGA designs being pushed to the upper limit, routability optimization is becoming a fundamental issue in modern physical design flow for FPGA. In this section, we have proposed a routability-driven FPGA packing and placement engine called UTPlaceF. A novel packing algorithm, PCAP, and congestion-aware detailed placement techniques for wirelength and routability co-optimization are proposed. The experimental results show that UTPlaceF achieves high-quality packing and placement solutions, which outperform the top-3 winners of the ISPD 2016 placement contest and other state-of-the-art FPGA placers.

2.4 UTPlaceF-DL: A New FPGA Placement Paradigm without Explicit Packing

As we have discussed in Section 2.1 and Section 2.3, the *Place-Pack-Place-Legalize* methodology currently dominates the state-of-the-art industrial FPGA CAD tools [68]. However, by using such a methodology, large discrepancies between the initial placements and the final legal solutions can still be observed, which implies that metrics (e.g., wirelength, timing, and routability) being carefully optimized in initial placements can be ruined in final legal solutions. The reason is usually twofold. Firstly, most existing initial placement approaches only seek to optimize the objectives without considering the effect of packing. As a result, large perturbations can be introduced in the later packing stage, especially for those hard-to-pack designs. Secondly, when forming a BLE/CLB in the packing stage, its location is typically estimated by the average location of its containing cells, which, however, can be far away from its final legal position.

To remedy these deficiencies in previous works, we propose UTPlaceF-DL, a new paradigm for FPGA placement without explicit packing. In UTPlaceF-DL, a final legal solution can be achieved directly from an initial placement by combining the previously separated packing and final placement/legalization steps. Our experiments show that the overall implementation quality, as well as the correlation between flat initial placements (FIPs) and final legal solutions, can be significantly improved by using UTPlaceF-DL. Our major contributions are highlighted as follows:

- We present a new paradigm, UTPlaceF-DL, for FPGA placement without an explicit packing stage, which is drastically different from the conventional placement and packing approaches.

- We propose an accurate packing estimation model to incorporate the effect of packing in the flat initial placement (FIP), which significantly improves the correlation between FIPs and final legal solutions compared with the previous state-of-the-art.

- We propose a fully parallelizable direct legalization (DL) technique that can produce legal solutions straightly from FIPs by simultaneously exploring the solution spaces of placement and packing.

- UTPlaceF-DL outperforms the winners of the ISPD 2016 contest as well as three state-of-the-art academic placers UTPlaceF [40], RippleF-PGA [12], and GPlace [62] in routed wirelength with competitive runtime on the ISPD 2016 benchmark suite [79].

The rest of this section is organized as follows. Section 2.4.1 formally defines the problem of direct legalization. In Section 2.4.2, we point out the inherent challenges of the proposed flow. Section 2.4.3 details our proposed algorithms. Section 2.4.4 shows the experimental results, followed by the summary in Section 2.4.5.

Table 2.6: Notations used in the direct legalization problem

$\mathcal{V}$	The set of LUTs and FFs
$\mathcal{S}$	The set of CLB slices available on the target FPGA device
$\boldsymbol{x}', \boldsymbol{y}'$	The x and y coordinates of cells in $\mathcal{V}$ in FIP
$\boldsymbol{x}, \boldsymbol{y}$	The x and y coordinates of cells in $\mathcal{V}$ in the final legal solution
$z_{v,s}$	Binary variables that represent if cell $v \in \mathcal{V}$ is assigned to CLB slice $s \in \mathcal{S}$
$\phi(c)$	The clustering score of a set of LUTs and FFs c
D	The cell maximum displacement constraint

2.4.1 The FPGA Direct Legalization Problem

In UTPlaceF-DL, the *direct legalization* (DL) is a step that produces a legal solution directly from an FIP. Given the notations defined in Table 2.6, the FPGA DL problem can be defined as follows:

$$\max_{\boldsymbol{x}, \boldsymbol{y}} \quad \sum_{s \in \mathcal{S}} \phi(\{v \in \mathcal{V} \mid z_{v,s} = 1\}) - \lambda \, \mathrm{HPWL}(\boldsymbol{x}, \boldsymbol{y}), \tag{2.6a}$$

$$\text{s.t.} \quad \sum_{s \in \mathcal{S}} z_{v,s} = 1, \forall v \in \mathcal{V}, \tag{2.6b}$$

$$\{v \mid v \in \mathcal{V}, z_{v,s} = 1\} \text{ is legal}, \forall s \in \mathcal{S}, \tag{2.6c}$$

$$|x_v - x'_v| + |y_v - y'_v| \le D, \forall v \in \mathcal{V}. \tag{2.6d}$$

The objective (2.6a) is to maximize the total clustering score $\sum_{s \in \mathcal{S}} \phi(\{v \in \mathcal{V} \mid z_{v,s} = 1\})$ and minimize a wirelength term $\lambda \, \mathrm{HPWL}(\boldsymbol{x}, \boldsymbol{y})$ normalized by a positive parameter λ. The clustering score function $\phi(c)$ typically captures the pin/net sharing, timing impact, and so on, within each CLB slice, like attraction functions used in conventional packing algorithms [58, 66, 67]. In general, $\phi(c)$ can be customized for different optimization targets. The details of the objective setting used in our framework will be elaborated in Section 2.4.3.3.4. The constraint (2.6b) guarantees that each LUT and FF is assigned to exactly

one CLB slice. The constraint (2.6c) ensures that all the architecture/packing rules stated in Section 2.2 are satisfied. The maximum displacement constraint (2.6d) is introduced to better preserve the FIP. We treat (2.6d) as a soft constraint since a legal solution may not always exist for a given D.

Unlike previous approaches, our DL formulation guarantees both placement and packing legality while optimizing the objective (2.6a). Besides, the maximum displacement is explicitly considered. It is, however, much harder to be dealt with in traditional methods (e.g., packing-based flows).

2.4.2 Challenges of Direct Legalization-Based Flow

Although the DL-based flow has lots of potential merits, there are several new challenges come into the field with it.

To achieve a smooth DL process, the flat initial placement (FIP) needs to be as close as possible to a legal solution. Otherwise, the final legal solution cannot be obtained with a small placement perturbation. In an FIP, each cell is associated with an area and the final FIP solution heavily depends on the cell area assignment. Therefore, a proper cell area assignment is essential for achieving a reasonably legal FIP. In most of the previous works, cell areas were set statically based on some empirical estimations [12, 62]. However, the area of a cell should be largely determined by its resource demand, which is, in fact, determined by its packing solution. For example, if a FF occupies (the FF portion of) a half CLB slice alone in the final solution due to the control set conflict, it should be assigned a larger area in the FIP. Considering packing
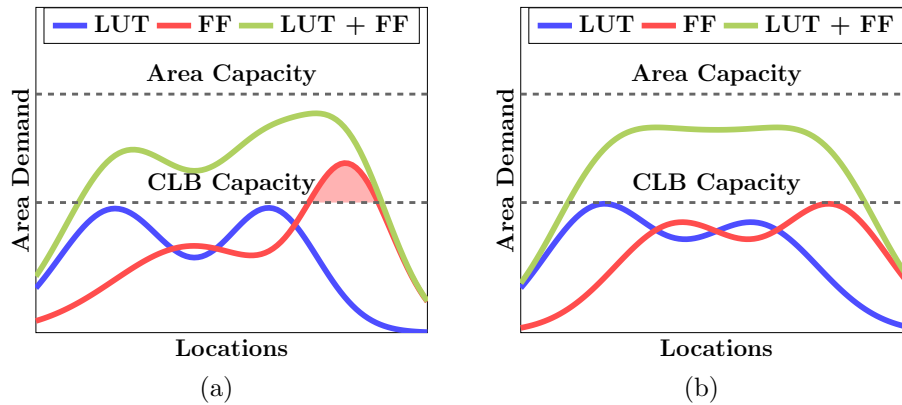
Figure 2.11: (a) An area overflow-free placement but with FF demand overflow (the shaded red region). (b) A legal placement.

solutions are not actually available in FIPs, how to model the effect of packing in cell area assignment is indeed a challenge.

Another important problem in FIP is how to properly distribute cells of different types. In quadratic placers, to achieve a roughly-legalized placement, overlap removal techniques (e.g., rough legalization) distribute cells by evening out area demands throughout the layout. However, an area overflow-free placement can be far away from a truly legal solution since the area metric alone cannot capture utilizations of different cell types. This issue can be better illustrated in Figure 2.11. Here the LUT and FF area demands (the blue curve and the red curve) represent the numbers of CLBs required by LUTs and FFs, respectively, in different locations. The CLB capacity (the lower dashed line) represents the numbers of CLB slices available in each location, and the area capacity (the upper dashed line) is the maximum LUT + FF area constraint used by the placers. In Figure 2.11(a), despite the satisfaction

of the LUT + FF area constraint, large displacement will still be introduced in the later legalization step due to the FF demand overflow (the shaded red region). Figure 2.11(b) gives a legal case where both LUT and FF demands are lower than the CLB capacity. This issue, of course, can be avoided by over-constraining the area capacity but at the cost of resource wasting.

To legalize a placement, many previous works adopted Tetris-like approaches [27]. In such a greedy approach, only one cell/cluster is considered and legalized at a time, which leads to a narrow solution space exploration. To explore a broader solution space, however, much more expensive computational effort is typically required. The scalability issue is even more severe in the DL-based flow since the flat netlist without any pre-clustering is directly considered. Therefore, how to explore a sufficiently large solution space while maintaining good runtime scalability is also a challenge for the DL-based flow.

To sum up, there are several issues discussed in this section that need to be resolved before we can confidently adopt the DL-based flow. No existing work has considered these issues, therefore, how to overcome them is a major challenge of this work.

2.4.3 UTPlaceF-DL Algorithms
2.4.3.1 Overall Flow

The overall flow of UTPlaceF-DL is illustrated in Figure 2.12. The whole flow starts with a flat initial placement (FIP). Besides the conventional quadratic program solving and rough legalization, a new dynamic LUT/FF
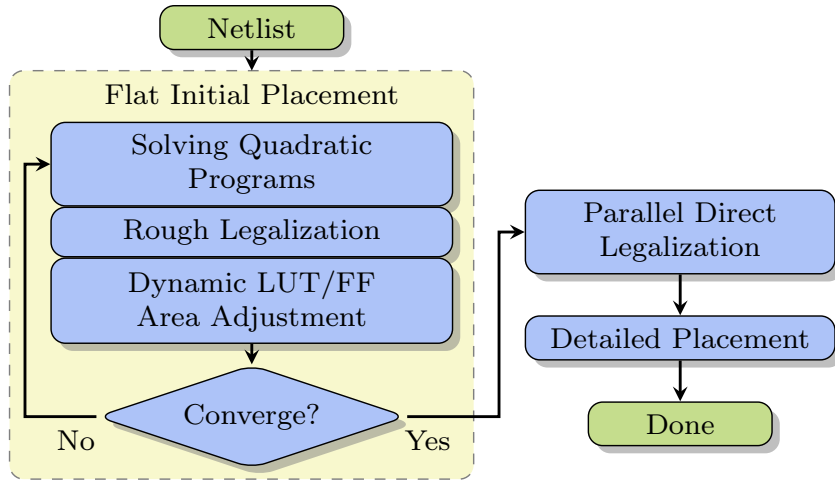
Figure 2.12: The overall flow of UTPlaceF-DL.

area adjustment step is performed after each placement iteration. This new step aims at resolving the two FIP-related issues pointed out in Section 2.4.2. More specifically, the area of each LUT and FF is dynamically adjusted to account for the impact of both packing and utilizations of different cell types. Once the FIP converges to a roughly legal solution, a high-quality legal placement can be straightly produced by the parallel direct legalization, in which the Formulation (2.6a) – (2.6d) is solved. The detailed placement then conducts further optimization on the legal placement before the final solution is delivered.

As the centerpieces of this work, the dynamic LUT/FF area adjustment and the parallel direct legalization techniques will be detailed in Section 2.4.3.2 and Section 2.4.3.3, respectively.

56

2.4.3.2 Dynamic LUT/FF Area Adjustment

Since packing and cell utilization calculation are both very local-scoped operations, it is reasonable to analyze them in a small spatial neighborhood context for each cell. Additionally, considering that cells of different types do not affect the packing legality or compete for logic resources against each other, it is also reasonable to consider each cell type individually from the perspective of solution legality. Therefore, the area of each cell can be largely determined based on its spatial neighbors of the same type.

In this section, a cell u is said to be a neighbor of a cell v if u and v have the same cell type and u falls into the box region $(x_v - L, y_v - L, x_v + L, y_v + L)$ centering at v, where (x_v, y_v) is the location of v and L is a constant being empirically set to 5 CLB slices in our framework. In the rest of this section, we will use $\mathcal{N}_v$ to denote the set of neighbors of v, and use $\mathcal{N}_v^+$ to denote $\{v\} \cup \mathcal{N}_v$.

2.4.3.2.1 Area Updating

To determine a cell area, the following three aspects are considered in our framework: (1) the cell resource demand; (2) the local resource utilization; (3) the routability impact. As discussed in Section 2.4.2, the resource demand of a cell is mainly determined by its packing solution, so the effect of packing is our major consideration here. Besides, the local resource utilization also needs to be considered and, as pointed out in Section 2.4.2, this should be done for different cell types (e.g., LUT and FF) individually. We also take the routability impact into consideration to avoid over-congested solutions.

57

Given a LUT (FF) v, we first define the local LUT (FF) utilization at v, denoted by U_v, as follows:

$$U_v = \frac{\sum_{i \in \mathcal{N}_v^+} A_i}{C_v}, \tag{2.7}$$

where C_v denotes the number of CLB slices within the box region $(x_v - L, y_v - L, x_v + L, y_v + L)$ that we used to define $\mathcal{N}_v$ and A_i denotes the LUT (FF) resource demand of the cell i. The magnitude of A_i here can also be interpreted as the degree of difficulty to pack i. The methods to compute A_i for LUTs and FFs will be detailed in Section 2.4.3.2.2 (Equation (2.9)) and Section 2.4.3.2.3 (Equation (2.10)), respectively.

We then define the new area of v, denoted by a_v, as follows:

$$a_v = \begin{cases} \min(a_v' \beta_+, \ A_v U_v \gamma_v), & \text{if } A_v U_v \gamma_v > a_v', \\ \max(a_v' \beta_-, \ A_v U_v \gamma_v), & \text{if } A_v U_v \gamma_v < a_v' \text{ and } R_v < R_{\max}, \\ a_v', & \text{otherwise}, \end{cases} \tag{2.8}$$

where a_v' denotes the current area of v, $\gamma_v \geq 1$ denotes the cumulative inflation ratio of v for routability optimization, R_v denotes the local routing utilization at v, and $R_{\max}$ is an empirically determined constant representing the maximum routing utilization for area shrinking. Besides, $\beta_+ > 1$ and $\beta_- < 1$ are two parameters to control the rates of area increasing/decreasing.

Intuitively, if a LUT (FF) v is hard to pack (large A_v) and is located in a region with high LUT (FF) utilization (large U_v), as well as high routing congestion (large γ_v), its area will be inflated. Otherwise, we will shrink its area to allow other cells to get in, but only when the region is not routing-

congested ($R_v < R_{\max}$). To achieve a smooth area adjustment, β_+ and β_- are set to 1.1 and 0.95, respectively. The γ_v is cumulatively updated using a history-based cell inflation technique similar to [12,40]. The routing congestion is estimated by a fast global router NCTUgr [53], and $R_{\max}$ is empirically set to 0.65 in our framework. The impacts of different β_+, β_-, and $R_{\max}$ settings on the solution quality will be further discussed in Section 2.4.4.2.

In UTPlaceF-DL, we perform the rough legalization for LUTs and FFs together using the same fence regions to maintain the relative order between them. Considering LUTs and FFs do not occupy the same logic resources, adjusting their areas directly targeting to their resource demand A_v (Equation (2.9) and Equation (2.10)) can result in low resource usage. This issue is prevented by scaling A_v using the local resource utilization U_v in Equation (2.8). By doing so, cells in low-utilization regions will be assigned areas that are much smaller than their actual resource demand to leave spaces for other cells.

Our dynamic area adjustment technique can effectively mitigate the "unbalanced resource issue" illustrated in Figure 2.11 and resolve "packing hotspots" that are harmful to the smoothness of the subsequent direct legalization process. Figure 2.13 shows the LUT/FF utilization (Equation (2.7)) maps of a design with/without this technique when the area constraint is satisfied (see Figure 2.11). It can be seen that, with this technique, a huge FF hotspot (Figure 2.13(b)) is resolved (Figure 2.13(d)). More interestingly, the LUTs that are original in the FF hotspot (Figure 2.13(a)) tend to follow the

59
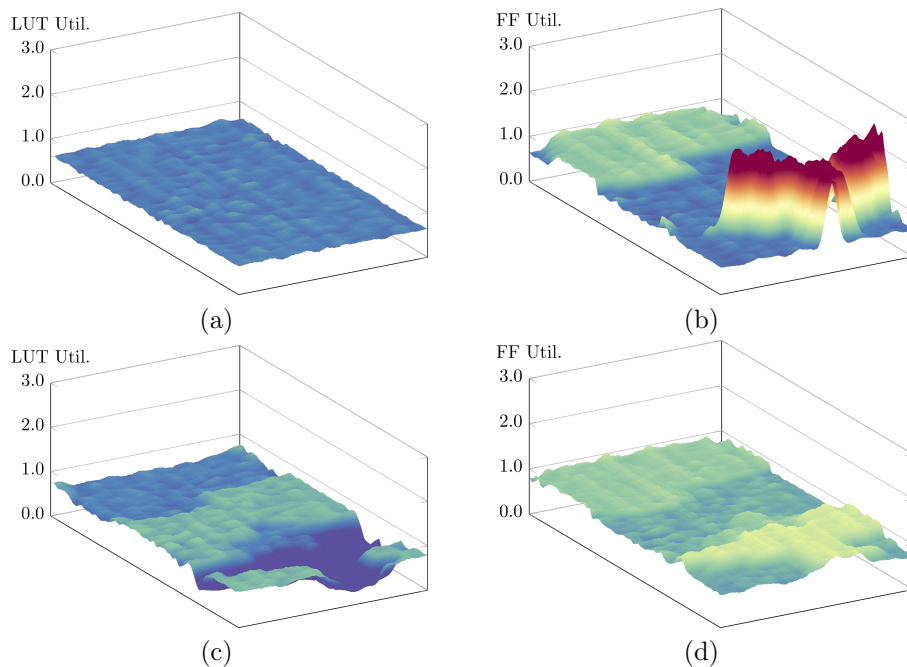
Figure 2.13: (a) and (b) are the 3-D LUT and FF utilization (Equation (2.7)) maps without our dynamic area adjustment. (c) and (d) are the ones with it applied. The area constraint is satisfied in both placement solutions. This is an example of the challenging case illustrated in Figure 2.11. The experiments are based on design `FPGA-10` in the ISPD 2016 benchmark suite [79].

movement of their connected FFs, which results in a "valley" (Figure 2.13(c)). This is actually an expected behavior since it implicitly preserves the relative cell ordering, which is important for the placement quality.

2.4.3.2.2 LUT Resource Demand Estimation

Now we present the method to estimate the resource demand (the A_i in Equation (2.7)) of a LUT, or more precisely, the amount of LUT portion in a CLB needed by a given LUT. Recall that, in the target device, each CLB contains

8 BLEs and each BLE can accommodate up to 2 LUTs if they are architecturally compatible. Therefore, each LUT can take 1 or 1/2 BLE, which are equivalent to 1/8 and 1/16 CLB in a compact packing solution. Considering a packing solution is not yet available in an FIP, a probabilistic estimation defined in Equation (2.9) is used instead for a LUT v with its neighbors $\mathcal{N}_v$.

$$A_v^{(\mathrm{LUT})} = \frac{1}{16} \frac{|\widehat{\mathcal{N}}_v|}{|\mathcal{N}_v|} + \frac{1}{8} \frac{|\mathcal{N}_v| - |\widehat{\mathcal{N}}_v|}{|\mathcal{N}_v|}, \qquad (2.9)$$

where $\widehat{\mathcal{N}}_v$ denotes the set of LUTs in $\mathcal{N}_v$ that can be fitted into the same BLE with v. Intuitively, if a LUT is architecturally compatible with most of its neighbors, its resource demand will tend to be small, otherwise, it will end up with a larger resource demand.

2.4.3.2.3 FF Resource Demand Estimation

The resource demand (the A_i in Equation (2.7)) estimation for FFs is more subtle due to their complicated control set rules. Given a FF v, one can, of course, enumerate all possible packing solutions of $\mathcal{N}_v^+$ and come up with a weighted sum similar to Equation (2.9) to get an average-case estimation. However, this method is too computationally expensive to be practical. Instead, we turn to a best-case estimation based on the tightest packing solution, which can provide us a firm lower bound of the FF resource demand.

For the notation simplicity, here we define every 4 FFs that share the same (CK, SR, CE) in a CLB slice as a *quarter CLB*, and define every 8 FFs that share the same (CK, SR) in a CLB slice as a *half CLB*. For instance, in

Figure 2.2(b), the 4 "FF A"s in BLE $0-3$ is a quarter CLB and the 8 FFs in BLE $0-3$ is a half CLB.

An instant observation is that, to produce the tightest packing solution, FFs with the same control set need to be packed together as much as possible. Given this observation, our approach to estimating the lower-bound FF demands can be illustrated by Figure 2.14. Using the (CK, SR) group of (B, P) in Figure 2.14 as an example, there are 5 and 2 FFs with CE of X and Z. In the tightest packing solution, at least $\lceil 5/4 \rceil = 2$ and $\lceil 2/4 \rceil = 1$ quarter CLBs are required for the control sets (B, P, X) and (B, P, Z). Since any two of these $2 + 1 = 3$ quarter CLBs can be fitted into the same half CLB, there will be a minimum of $\lceil 3/2 \rceil = 2$ half CLBs for the (B, P) group. Considering the control sets of any two half CLBs are independent, we can safely set the resource demand of each half CLB as $1/2$ (of a CLB). Therefore, the minimum demand of the (B, P) group will be $1/2 \cdot 2 = 1$. After that, we can divide this demand of 1 into $2/(2 + 1) = 2/3$ and $1/(2 + 1) = 1/3$ based on the quarter CLB counts in (B, P, X) and (B, P, Z). Finally, the resource demands of each FF in (B, P, X) and (B, P, Z) can be estimated as $2/3/5 = 2/15$ and $1/3/2 = 1/6$, respectively, by evenly distributing the demand of $2/3$ and $1/3$ to 5 and 2 FFs.

To generalize the above discussion, let v be a FF with control set (CK$_0$, SR$_0$, CE$_0$), and let {CE$_0$, CE$_1$, ..., CE$_m$} be the set of CE nets in $\mathcal{N}_v^+$. If we denote the number of FFs in $\mathcal{N}_v^+$ with the control set (CK$_0$, SR$_0$, CE$_i$) as n_i,

CK, SR	A P			B P		B Q
CE	X	Y	Z	X	Z	Y
#FF	6	11	4	5	2	1
#Quarter CLB	2	3	1	2	1	1
#Half CLB		3			2	1
(CK, SR) Dem.		3/2			1	1/2
Est. FF Dem.	1/12	3/44	1/16	2/15	1/6	1/2
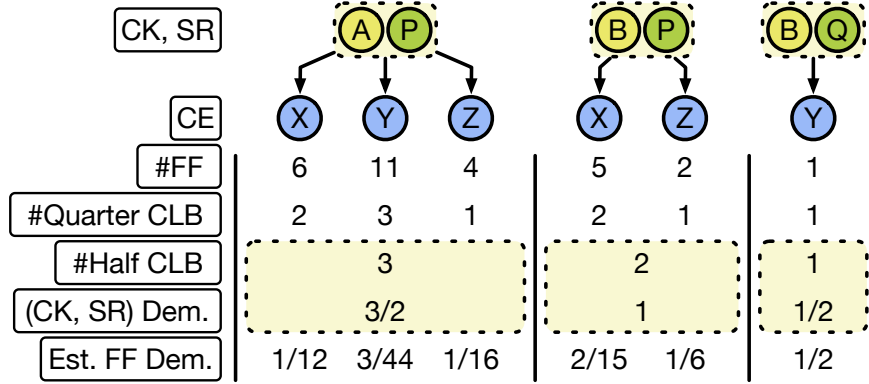
Figure 2.14: Illustration of FF resource demand estimation.

for $0 \leq i \leq m$, the estimated FF demand of v can be expressed as follows:

$$A_v^{(\text{FF})} = \frac{1}{2} \frac{\lceil \frac{n_0}{4} \rceil}{\sum_{i=0}^{m} \lceil \frac{n_i}{4} \rceil} \left\lceil \frac{\sum_{i=0}^{m} \lceil \frac{n_i}{4} \rceil}{2} \right\rceil \frac{1}{n_0} \alpha^{(\text{FF})}. \tag{2.10}$$

Note that our analysis in Figure 2.14 corresponds to the tightest packing, which realizes the lower bound FF demands. In practice, however, a packing solution is likely to be looser than that when net sharing and other optimization metrics are considered. Therefore, we introduce $\alpha^{(\text{FF})} \geq 1$ in Equation (2.10) to give more flexibility to the packing. In our framework, we empirically set $\alpha^{(\text{FF})}$ to 1.1.

If we ignore the $\alpha^{(\text{FF})}$ term, the FF demand estimated by Equation (2.10) is in the range $[1/16, 1/2]$, which implies that FF demands can vary up to $8\times$ (e.g., the FF demands of (A, P, Z) and (B, Q, Y) in Figure 2.14). In a traditional flow, the discrepancy between FIPs and legal solutions is mainly from their incapability of capturing this large demand variance, as shown in Figure 2.13. Therefore, our FF demand estimation method detailed in this

section is essential for the proposed direct legalization-based flow.

It is worthwhile to mention that, although the LUT/FF resource demand estimation elaborated in Section 2.4.3.2.2 and Section 2.4.3.2.3 are specific to the architecture detailed in Section 2.2, the same idea is also applicable to other FPGA devices with different architectures.

2.4.3.2.4 Area Adjustment Scheduling

Another important question needs to be answered is when to start the area adjustment. There is a trade-off between the area estimation accuracy (Equation (2.9) and Equation (2.10)) and the required area adjustment rates. In general, the later we start the dynamic area adjustment in the FIP the more accurate area estimation we can make. However, it also requires faster adjustment rates (larger β_+ and smaller β_-) to guarantee that cells can converge to their target areas within the reduced number of adjustment iterations. As will be discussed in Section 2.4.4.2, a too fast adjustment rate can be harmful to the placement convergence and quality. Therefore, in the proposed flow, we choose to start the dynamic area adjustment from the beginning of the FIP with relatively slow adjustment rates $\beta_+ = 1.1$ and $\beta_- = 0.95$.

2.4.3.3 Fully Parallelizable Direct Legalization

Our direct legalization (DL) takes a roughly-legalized FIP and produces a legal solution by solving the Formulation (2.6a) – (2.6d). The idea is partially similar to the clustering algorithms in [57, 68], but we are aiming at a legal

64

placement directly. Besides, the cell displacement is explicitly considered in our formulation. Compared with traditional greedy methods, our method can explore a significantly larger solution space by exploiting the strength of parallelism.

2.4.3.3.1 A College Admission Analogy

The proposed DL algorithm is inspired by the *College Admission Problem* [23]. One can think that each CLB slice is a "college", each cell is a "student", and cells sharing common nets are "friends". Then, DL can be regarded as a process of colleges admitting students. By using this analogy, the DL process is fully parallelizable in nature in the sense that all colleges can make decisions simultaneously. Our DL formulation, however, is more complicated than the original college admission problem for the following two reasons: (1) a student (cell) rates a college (slice) not only based on the college (slice) itself but also depending on the decision making of its friends (connected cells); (2) some students (cells) cannot be in the same college (slice) for the FPGA architectural legality.

2.4.3.3.2 The Node-Centric Algorithm

The key idea of our DL algorithm is that each slice finds the cluster of cells that fits it best, then offers this cluster to all the cells involved. The "admission" happens when all the cells in this cluster accept this "offer". The process of sending and accepting/rejecting "offers" between slices and cells is iteratively

performed until no potential "admission" could happen anymore. Different slices here can create cluster candidates and make their own decisions independently. Moreover, a cluster can be simultaneously created and considered by multiple slices, which implicitly explores the solution space of placement together with packing. This elegant property overcomes the drawback of all existing methods, where placement and packing cannot be considered at the same time.

In our DL algorithm, the computation is mainly centered at each slice. Thus, we call each slice as a *computation node* and each computation node can run in parallel against each other.

The node-centric DL algorithm flow at each computation node is shown in Figure 2.15. Each computation node maintains a set of cells that have been determined in the slice (det), a priority queue of cluster candidates (pq), a set of neighbor cells (nbr), and a list of seed clusters (scl) for new candidate generation. The pq stores the K best clusters found so far at each computation node and we use K = 10. Clusters are created by adding cells in nbr into seed clusters in scl in our algorithm. Besides, a variable i is also maintained by each computation node to record the number of iterations since the last change of the best candidate in pq. We will use $pq.top()$ to denote the best candidate in a pq. Before the DL starts, for each computation node, we initialize i to 0, set det and pq to $\varnothing$, initialize nbr as the set of cells within a predefined distance d (default is 1) to the slice, and make scl to contain an empty cluster without any cells.
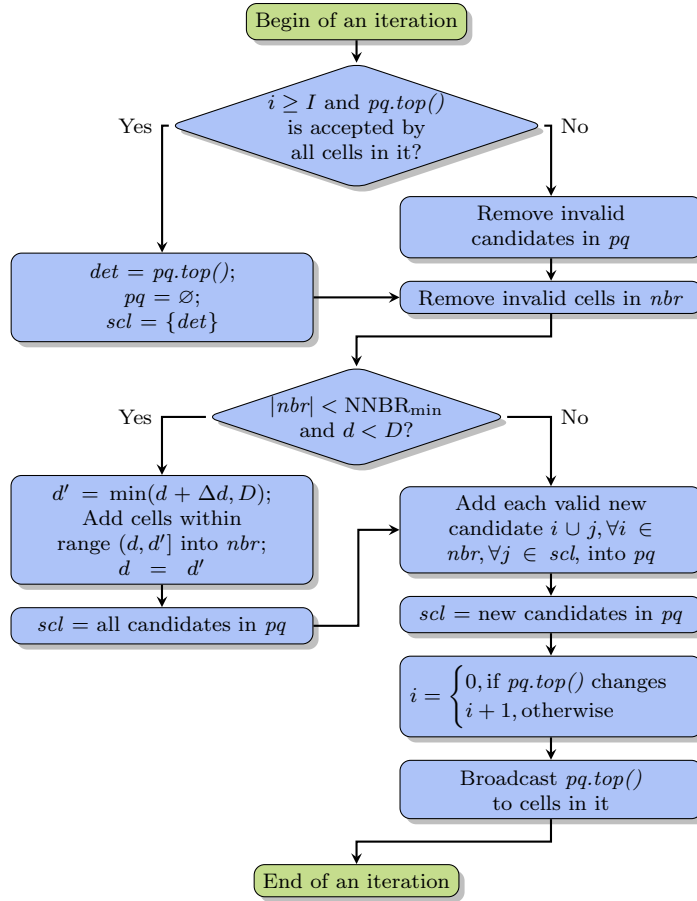
Figure 2.15: The node-centric DL algorithm flow at each computation node (CLB slice).

In every DL iteration, as shown in Figure 2.15, each computation node first checks if its *pq.top()* (the best candidate in *pq*) can be committed into the slice. The *pq.top()* will be committed only if it has been stable for a long enough time ($i \geq I$) and it has been accepted by all the cells in it. I is set to 3 in our algorithm to prevent committing premature candidates. Once the *pq.top()* is valid to commit, we will update *det* as *pq.top()* and reset *pq*. To

guarantee that all subsequent candidates contain the set of determined cells *det*, *scl* is set to only have *det* after each *pq.top()* committing.

If the *pq.top()* is not yet ready to commit, invalid candidates and cells will be removed from *pq* and *nbr*. A cell can become invalid if it has been added to another slice or it is no longer architecturally compatible with *det*. A cluster becomes invalid if one or more cells in it are invalid. After that, if there are too few available cells in *nbr* ($|nbr| < \text{NNBR}_{\min}$), the current maximum cell displacement constraint d will be relaxed by Δd, and more neighbor cells will be added to guarantee a large enough solution space that we can explore. Besides, we will also copy all candidates in *pq* to *scl* to ensure that they can be considered by these newly added neighbor cells. To honor the ultimate maximum displacement constraint D, we will stop increasing d when it reaches D. $\text{NNBR}_{\min}$, Δd, and D are set to 10, 1, and 12, respectively, in our algorithm.

To create new cluster candidates, we add each cell in *nbr* into each cluster in *scl*. Those valid new candidates will be added into *pq*. Meanwhile, we also store them in *scl* for the candidate generation in the next DL iteration. After that, i will be increased by 1 if *pq.top()* does not change, otherwise, we reset it to 0.

Finally, the computation node broadcasts its *pq.top()* to all involved cells and let each of them make a decision. The decisions of these cells will determine if this *pq.top()* can be committed in the next DL iteration.

After all computation nodes finish their broadcasting, a cell may receive multiple "offers" from a set of different slices $\mathcal{S}$. In such a case, the cell will select the slice $s \in \mathcal{S}$ with the highest score improvement $\Delta\text{SCORE}(s)$. The score improvement of a slice s is defined as

$$\Delta\text{SCORE}(s) = \text{SCORE}(s.pq.top(), s) - \text{SCORE}(s.det, s), \qquad (2.11)$$

where $\text{SCORE}(c, s)$ represents the score of cluster c in slice s and it will be given in Section 2.4.3.3.4. Intuitively, $\Delta\text{SCORE}(s)$ is the amount of score improvement by committing the $pq.top()$ in slice s.

Algorithm 7: Fully Parallelizable Direct Legalization

    Input : A roughly-legalized placement, the set of cells $\mathcal{C}$, the set of slices (computation nodes) $\mathcal{S}$, and the initial maximum displacement constraint d.

    Output: A legal placement.

1 **parallel foreach** $s \in \mathcal{S}$ **do**
2     $s.i \leftarrow 0$;
3     $s.det \leftarrow \varnothing$;
4     $s.pq \leftarrow \varnothing$;
5     $s.nbr \leftarrow \{c \in \mathcal{C} \mid \text{dist}(c, s) \leq d\}$;
6     $s.scl \leftarrow \{\text{an empty cluster}\}$;
7 **while** *exists valid pq.top() for* $s \in \mathcal{S}$ **do**
8   **parallel foreach** $s \in \mathcal{S}$ **do**
9   Run the node-centric algorithm shown in Figure 2.15;
10   **parallel foreach** $c \in \mathcal{C}$ **do**
11   Among $\{s \in \mathcal{S} \mid c \in s.pq.top()\}$, pick s^* that has the highest $\Delta\text{SCORE}(s^*)$ defined in Equation (2.11) and inform s^* that c accepts its $pq.top()$;

Algorithm 7 summarizes the whole fully parallelizable DL process. All computation nodes are initialized in parallel from line 1 to line 6. In each

DL iteration, the node-centric algorithm presented in Figure 2.15 is executed by each slice individually from line 8 to line 9. After that, each cell receives "offers" from slices with *pq.top()* containing it, and it picks the one with the highest score improvement ΔSCORE (Equation (2.11)) and informs its acceptance to the selected slice from line 10 to line 11. The loop from line 7 to line 11 is repeated until no more valid candidate exists.

2.4.3.3.3 Convergence Requirement

In general, a valid *pq.top()* may never be accepted by all the cells in it. In such a case, our DL algorithm will fall into an infinite loop. To guarantee the convergence of the algorithm, an extra requirement is imposed as stated in Lemma 2.4.1.

Lemma 2.4.1 *Suppose s_1 and s_2 are two different computation nodes (slices). If $\Delta SCORE(s_1) \neq \Delta SCORE(s_2)$ holds for any choice of s_1 and s_2 in the same DL iteration, the convergence of the proposed DL algorithm is guaranteed.*

Lemma 2.4.1 basically requires a tie-breaking mechanism for the case that multiple computation nodes offer the same score improvement (Equation (2.11)) to a set of cells. In our implementation, a tie is broken based on the unique identifier of each computation node since a cell can receive at most one "offer" from a computation node in each iteration.

70

2.4.3.3.4 Score Function

Since our DL explores the solution spaces of placement and packing simultaneously, the score function needs to capture both placement- and packing-related metrics. Given a slice s and a cluster c, the score of c in s is defined as follows:

$$\text{SCORE}(c, s) = \sum_{e \in \text{Net}(c)} \frac{\text{InternalPins}(e, c) - 1}{\text{TotalPins}(e) - 1} - \lambda \, \Delta\text{HPWL}(c, s), \qquad (2.12)$$

where $\text{Net}(c)$ denotes the set of nets that have at least one cell in c, $\text{TotalPins}(e)$ denotes the total pin count of net e, $\text{InternalPins}(e, c)$ represents the number of pins of net e in c, and $\Delta\text{HPWL}(c, s)$ represents the HPWL increase of moving cells in c from their FIP locations to s. λ is a positive weighting parameter, which is empirically set to 0.02 in our algorithm. The first term defines the clustering score $\phi(c)$ in Equation (2.6a) and it here grants a higher score to clusters that absorb more external nets as internal ones, which can effectively reduce routing demands and improve routability. The second term gives a higher preference to candidates that lead to a large wirelength reduction.

2.4.3.3.5 Parallelization Scheme

As illustrated in Algorithm 7, our DL algorithm is massively parallelizable. Like colleges independently making decisions in the analogy of college admission (Section 2.4.3.3.1), different computation nodes (slices) can execute the flow illustrated in Figure 2.15 perfectly in parallel in each DL iteration (Algo-

rithm 7 line 8 to line 9). Moreover, cells can also process the results generated by slices and send back their decisions individually (Algorithm 7 line 10 to line 11).

Since the number of slices and cells in a modern FPGA is typically at the scale of 10^4 or more (e.g., our target FPGA contains 67K slices), a fine-grained parallelization with even tens of threads is potentially viable in our DL algorithm. Our experiments in Section 2.4.4.3 demonstrates the nearly-linear runtime scalability of our DL algorithm with respect to the number of threads. This extreme parallelizability can further facilitate efficient implementations of our DL algorithm on hardware accelerators, like FPGAs and GPUs.

Another strong property of our DL algorithm is its serial equivalency. Serial equivalency is a property to guarantee that a parallel algorithm always produces exactly the same solution as its serial version does. That is, our DL algorithm guarantees to produce the same solution, regardless of the number of threads used. Therefore, we can enable as much as available parallel computational resources without sacrificing the quality of results.

2.4.3.3.6 Post-DL Exception Handling

Although the convergence of our DL algorithm can be guaranteed by Lemma 2.4.1, after the regular DL process described in Algorithm 7, a small portion (typically $< 1\%$) of cells may still not be able to find legal positions within a given maximum displacement constraint D. To legalize these remaining cells, one can, of course, keep relaxing D until a legal solution is reached,

like a *Tetris*-based legalizer [27]. However, this approach can result in a huge displacement. Instead of *Tetris*-based approaches, we choose to rip up some already determined clusters and reallocate these ripped cells together with those originally illegal ones to produce a legal solution. Since our FIP is nearly legal, this method is likely able to find a legal solution with very little placement perturbation.

One of the key questions here is which cluster to break. For an illegal cell v and a slice s with its determined cluster c, we first define the score of breaking c in s for v as

$$\text{SCORE}_{\text{ripup}}(v, s, c) = -\lambda_1 \, \Delta\text{HPWL}(v, s) - \lambda_2 \, \text{SCORE}(c, s) - \lambda_3 \, \text{Area}(c),$$

$$(2.13)$$

where $\Delta\text{HPWL}(v, s)$ denotes the HPWL increase of moving v to s, $\text{SCORE}(c, s)$ denotes the score of c in s as defined in Equation (2.12), and $\text{Area}(c)$ represents the total cell area (from FIP) in c. λ_1, λ_2, and λ_3 are three positive weighting parameters. In our experiments, we empirically set them to 0.02, 1.0, and 4.0, respectively. Intuitively, we prefer to move v to a low-score slice with little wirelength increase. The $\text{Area}(c)$ term is introduced to evaluate if the ripped cells are easy to legalize. If c has a large area, it either contains many cells or the cells it contains are hard to pack (recall Equation (2.9) and Equation (2.10)). For both cases, we tend to not break it.

Algorithm 8 summarizes our post-DL exception handling technique that legalizes illegal cells after the regular DL process (Algorithm 7). In the

73

Algorithm 8: Post-DL Exception Handling

    Input : A post-DL placement with the set of illegal cells $\mathcal{C}'$, the set of all cells $\mathcal{C}$, the set of all slices $\mathcal{S}$, and the maximum displacement constraint D.

    Output: A legal placement.

1 **foreach** $c \in \mathcal{C}'$ **do**

2 $D^{(c)} \leftarrow D$;

3   **while** Legalize $(c, D^{(c)})$ *is fail* **do**

4 $D^{(c)} \leftarrow D^{(c)} + 1$;

5 **Function** Legalize(c, D):

6 $ls^{(c)} \leftarrow \{s \in \mathcal{S} \mid \text{dist}(c, s) \leq D\}$;

7   Sort slices in $ls^{(c)}$ by their $\text{SCORE}_{\text{ripup}}$ defined in Equation (2.13) in descending order;

8   **foreach** $s \in ls^{(c)}$ **do**

9   **if** RipUpAndLegalize (s, c, D) *is success* **then**

10   **return** success;

11   **return** fail;

12 **Function** RipUpAndLegalize(s, c, D):

13 $lv \leftarrow \{v \in \mathcal{C} \mid v \in s.det\}$;

14 $s.det \leftarrow \{c\}$;

15   **foreach** $v \in lv$ **do**

16 $ls^{(v)} \leftarrow \{s \in \mathcal{S} \mid \text{dist}(v, s) \leq D \text{ and } s.det \cup v \text{ is legal}\}$;

17   **if** $ls^{(v)}$ *is* $\varnothing$ **then**

18   Remove c from $s.det$;

19   Put all $v \in lv$ back to $s.det$;

20   **return** fail;

21   **else**

22   Pick $s^* \in ls^{(v)}$ with the highest $\text{SCORE}(s^*.det \cup v, s^*) - \text{SCORE}(s^*.det, s^*)$;

23 $s^*.det \leftarrow s^*.det \cup v$;

24   **return** success;

---

main loop (line 1 to line 4), each illegal cell c is legalized with the maximum displacement constraint $D^{(c)}$ using the function $\texttt{Legalize}(c, D^{(c)})$. For each c, we set the initial displacement constraint as D (line 2), and incrementally relax it (line 4) if a legal solution cannot be found. To legalize an illegal cell c with displacement constraint D using $\texttt{Legalize}(c, D)$ (line 5 to line 11), we first collect the set of slices $ls^{(c)}$ that are within distance D w.r.t the location of c in FIP (line 6). Then we sort all slices in $ls^{(c)}$ by their $\text{SCORE}_{\text{ripup}}$ defined in Equation (2.13) in descending order (line 7). After that, we try to rip up $s \in ls^{(c)}$ one by one and legalize c using the function $\texttt{RipUpAndLegalize}(s, c, D)$ until the legalization successes (line 8 to line 10). If c cannot be legalized by breaking any slice in $ls^{(c)}$, $\texttt{Legalize}(c, D)$ will return *fail* (line 11) and the displacement constraint will be relaxed in the main loop (line 3 to line 4).

The details of the function $\texttt{RipUpAndLegalize}(s, c, D)$ are given in line 12 to line 24. The goal of this function is to break $s.det$ and legalize c as well as the cells in $s.det$ under the displacement constraint D. We first rip up $s.det$ and put c alone into it (line 14). Then, the cells that are originally in $s.det$ are legalized sequentially in the loop from line 15 to line 23. For each such a cell v, we first collect the set of slices $(ls^{(v)})$ that are within distance D (w.r.t the location of v in FIP) and have their det compatible with v (line 16). If no such slice exists, we discard all the changes that have been made in this function call (line 18 to line 19) and return *fail* (line 20). Otherwise, among all candidate slices $(ls^{(v)})$, we pick the one with the highest score gain $\text{SCORE}(s^*.det \cup v, s^*) - \text{SCORE}(s^*.det, s^*)$ and put v into it (line 22 to line

23). The function call successes only when all cells that are originally in $s.det$ can find legal positions within displacement D (line 24).

2.4.4 Experimental Results

Table 2.7: ISPD 2016 Contest Benchmarks Statistics

Design	#LUT	#FF	#RAM	#DSP	#Ctrl Set
FPGA-01	50K	55K	0	0	12
FPGA-02	100K	66K	100	100	121
FPGA-03	250K	170K	600	500	1281
FPGA-04	250K	172K	600	500	1281
FPGA-05	250K	174K	600	500	1281
FPGA-06	350K	352K	1000	600	2541
FPGA-07	350K	355K	1000	600	2541
FPGA-08	500K	216K	600	500	1281
FPGA-09	500K	366K	1000	600	2541
FPGA-10	350K	600K	1000	600	2541
FPGA-11	480K	363K	1000	400	2091
FPGA-12	500K	602K	600	500	1281
Resources	538K	1075K	1728	768	-

We implement UTPlaceF-DL in C++ based on UTPlaceF [40] and perform the experiments on a Linux machine running with Intel Core i9-7900X CPUs (3.30 GHz, 10 cores, and 13.75 MB L3 cache) and 128 GB RAM. OpenMP 4.0 [2] is used to support multi-threading. The benchmark suite released by Xilinx for the ISPD 2016 FPGA placement contest [79] is used to validate the effectiveness of the proposed approaches. All the routings are conducted by Xilinx Vivado v2015.4 [72]. The characteristics of the ISPD 2016 benchmark suite are listed in Table 2.7.

76

2.4.4.1 Effectiveness Validation of Proposed Techniques

Table 2.8 demonstrates the effectiveness of the proposed dynamic area adjustment (DAA) and direct legalization (DL) techniques. Here we compare four different placement methodologies, as listed in the four columns. The column "UTPlaceF" represents the original UTPlaceF [40] flow. The column "UTPlaceF + DAA" applies DAA on top of the original UTPlaceF. The column "UTPlaceF-DL" employs both DAA and the proposed DL by replacing the packing, CLB-level placement, and legalization subroutines in "UTPlaceF + DAA" flow with the proposed DL. To further demonstrate the effectiveness of the proposed DL, we also implement a greedy Tetris-based direct legalization (greedy DL) for comparison. This greedy DL adopts the same score function Equation (2.12) used by the proposed DL, and it legalizes one cell at a time to maximize the score improvement defined in Equation (2.11). The results of substituting the proposed DL in "UTPlaceF-DL" with this greedy DL are shown in the column "DAA + Greedy DL". Metrics "WL" and "RT" represent the routed wirelength and runtime, while "WLR" and "RTR" represent the wirelength and runtime ratios normalized to the "UTPlaceF-DL" column. Note that these four flows share the same underlying global and detailed placement engines, therefore noises from parts that are irrelevant to this work can be completely decoupled in this comparison.

It is worthwhile to mention that the proposed DL should not be applied without DAA since this can result in very suboptimal or even illegal solutions. In the proposed DL, all cells simultaneously seek to legalize themselves. With-

out DAA, however, the FIP solution can be greatly far away from a truly legal placement, and in this case, a significant portion of cells can fail to find nearby legal positions. Therefore, we always employ the proposed DL together with the DAA technique in our experiments.

In this experiment, we enable 16 threads for our DL algorithm due to its parallel nature, and all other parts (global/detailed placement) are single-threaded. While the UTPlaceF is universally executed with a single thread for the following two reasons: (1) the packing and legalization algorithms in UTPlaceF are inherently sequential and hard to be parallelized without quality degradation or extra threading communication overhead; (2) the packing and legalization in UTPlaceF only take about 15% of the total runtime on average, therefore, the overall performance gain would be still limited even if they have been carefully parallelized.

We first compare the columns "UTPlaceF" and "UTPlaceF + DAA", where the only difference is whether or not DAA is applied. On average, "UTPlaceF + DAA" achieves 1.4% better routed wirelength with only 5% runtime overhead compared with "UTPlaceF". The reason is that, with DAA applied, the FIP solution can be considerably closer to a truly legal solution due to its packing effect consideration. This can be further proved by the experimental results shown in Table 2.9, which we will discuss in details later in this section.

We then compare the columns "UTPlaceF + DAA" and "DAA + Greedy DL" with "UTPlaceF-DL" to demonstrate the effectiveness of the

proposed DL. These three methodologies share the same FIP solutions and only differ by their packing and legalization steps. As can be seen, on average, "UTPlaceF-DL" outperforms "UTPlaceF + DAA" and "DAA + Greedy" by 3.0% and 6.0%, respectively, in routed wirelength. It should be noted that "UTPlaceF-DL" outperforms "DAA + Greedy DL" on all twelve benchmarks in routed wirelength with only 7% longer runtime. Therefore, compared with the *Pack-Place-Legalize* methodology in "UTPlaceF + DAA" and the greedy DL, the proposed DL algorithm can effectively explore a larger solution space and achieves better solution quality.

By comparing the columns "UTPlaceF-DL" and "UTPlaceF", we can see that, with both DAA and DL employed, the proposed flow outperforms the original UTPlaceF by 4.4% in routed wirelength, while runs $1.24\times$ faster. It is worthwhile to mention that, among all the designs, `FPGA-10` contains the largest number of control sets and flip-flops, which make it severely difficult to pack and legalize. On this particular design, UTPlaceF-DL outperforms UTPlaceF by 29.5% in routed wirelength. Besides, UTPlaceF-DL also consistently excels on other control set-intensive designs, like `FPGA-06`, `FPGA-07`, and `FPGA-09`. Thus, our approach is especially effective for hard-to-pack designs.

Table 2.8: Routed Wirelength (WL in 10^3) and Runtime (RT in seconds) Comparison with UTPlaceF† [40]

Design	UTPlaceF† [40]				UTPlaceF† + DAA				DAA + Greedy DL				UTPlaceF-DL			
	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR
FPGA-01	346	70	1.016	1.15	342	70	1.005	1.15	371	55	1.092	0.90	340	61	1.000	1.00
FPGA-02	652	111	0.999	0.97	643	109	0.985	0.96	691	102	1.059	0.89	653	114	1.000	1.00
FPGA-03	3173	318	1.011	0.95	3107	331	0.990	0.99	3283	309	1.046	0.93	3139	333	1.000	1.00
FPGA-04	5440	399	1.020	1.01	5250	411	0.985	1.04	5479	362	1.028	0.92	5331	394	1.000	1.00
FPGA-05	9775	432	0.973	1.00	9687	449	0.964	1.03	10162	396	1.012	0.91	10045	434	1.000	1.00
FPGA-06	6505	718	1.121	1.24	6173	803	1.064	1.39	6185	541	1.066	0.94	5801	578	1.000	1.00
FPGA-07	9876	792	1.056	1.20	9718	849	1.039	1.28	9754	626	1.043	0.95	9356	662	1.000	1.00
FPGA-08	7735	641	0.932	0.92	7942	690	0.957	0.99	8451	650	1.018	0.94	8298	695	1.000	1.00
FPGA-09	12062	1409	1.037	1.59	11904	1385	1.023	1.56	12229	830	1.051	0.94	11633	887	1.000	1.00
FPGA-10	8178	1783	1.295	2.33	7935	1782	1.256	2.33	7399	730	1.171	0.95	6317	766	1.000	1.00
FPGA-11	9966	1001	0.951	1.12	10386	1023	0.991	1.15	10833	848	1.034	0.95	10476	890	1.000	1.00
FPGA-12	7635	1343	1.117	1.36	7557	1551	1.106	1.57	7534	914	1.102	0.93	6835	988	1.000	1.00
Norm.	-	-	1.044	1.24	-	-	1.030	1.29	-	-	1.060	0.93	-	-	1.000	1.00

† : This UTPlaceF version is fine-tuned for even better routed wirelength and runtime compared with the original publication [40].

Table 2.9: Displacement Comparison with UTPlaceF

Design	UTPlaceF [40]		UTPlaceF + DAA		DAA + Greedy DL		Proposed	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
FPGA-01	5.7	42.6	5.6	26.1	1.2	9.2	1.0	11.4
FPGA-02	5.2	305.0	5.9	191.1	1.1	11.3	1.0	11.8
FPGA-03	12.2	146.9	7.7	90.9	1.2	23.8	1.0	11.5
FPGA-04	20.3	124.3	7.8	128.0	1.3	64.3	1.1	11.5
FPGA-05	12.1	185.1	10.5	126.7	1.2	47.8	1.1	11.8
FPGA-06	60.4	187.8	17.0	209.0	1.7	52.7	1.3	11.8
FPGA-07	21.1	232.6	13.9	166.8	1.7	92.5	1.3	11.5
FPGA-08	11.8	95.4	6.3	107.9	1.1	13.3	1.0	10.9
FPGA-09	13.0	150.6	10.3	144.7	1.6	86.8	1.2	11.7
FPGA-10	25.5	164.7	26.2	127.4	2.6	166.7	1.4	11.7
FPGA-11	40.8	138.1	15.6	124.8	1.4	46.4	1.0	11.5
FPGA-12	28.7	177.6	13.2	176.6	1.5	74.1	1.1	11.5
Norm.	21.4	162.5	11.7	135.0	1.5	57.4	1.2	11.6

Another notable merit of UTPlaceF-DL is that the correlation between FIPs and post-legalization solutions can be significantly improved. This property is appealing in the sense that metrics, like wirelength, timing, and routability, optimized in FIPs can be greatly preserved in legal solutions. Table 2.9 shows the average and maximum cell displacements between the FIPs and the post-legalization/DL solutions by using the four different methodologies listed in Table 2.8. As can be seen, the original UTPlaceF introduces huge cell displacements with average and maximum values of 21.4 and 162.5, respectively. DAA technique alone can effectively reduce them down to 11.7 and 135.0 in "UTPlaceF + DAA". In "DAA + Greedy DL", by applying the greedy DL instead of the packing-based methodology, the average and maximum displacements can be further reduced to 1.5 and 57.4, respectively. For all twelve benchmarks, the proposed methodology with DAA and the pro-
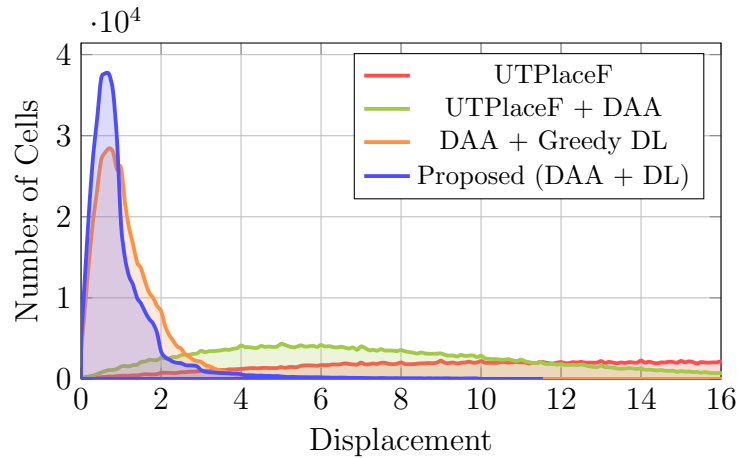
Figure 2.16: The cell displacement distributions of UTPlaceF, UTPlaceF + DAA, DAA + Greedy DL, and the proposed flow (DAA + DL) based on design `FPGA-03`. The displacement of a cell is defined as the Manhattan distance between its locations in the FIP and the post-legalization/DL placement. The average/maximum displacements are 12.2/146.9, 7.7/90.9, 1.2/23.8, and 1.0/11.5 in UTPlaceF, UTPlaceF + DAA, DAA + Greedy DL, and the proposed flow, respectively.

posed DL together can achieve maximum displacements that are less than 12.0, which is the predefined constraint in our DL algorithm. Meanwhile, the average displacements are all in the range from 1.0 to 1.5. As expected, our DAA and DL techniques can greatly help to preserve the FIP solution even after a legal solution is obtained.

Figure 2.16 visualizes the distributions of cell displacement between the FIPs and the post-legalization/DL solutions based on design `FPGA-03`. Compared with UTPlaceF, most cells in the proposed methodology are much closer to their original locations in the FIP. However, tremendously large displacements can be observed in UTPlaceF.

Table 2.10: HPWL and the Maximum LUT and FF Utilizations (Equation (2.7)) w/ and w/o DAA after FIP

Design	Norm. HPWL		Max. LUT & FF Util.	
	w/o DAA	w/ DAA	w/o DAA	w/ DAA
FPGA-01	1.093	1.000	0.87	1.00
FPGA-02	0.992	1.000	1.08	1.11
FPGA-03	0.982	1.000	1.46	1.12
FPGA-04	0.966	1.000	1.59	1.11
FPGA-05	0.945	1.000	1.61	1.09
FPGA-06	0.970	1.000	2.04	1.12
FPGA-07	0.932	1.000	2.10	1.12
FPGA-08	0.907	1.000	1.88	1.18
FPGA-09	0.990	1.000	1.96	1.28
FPGA-10	0.993	1.000	2.76	1.28
FPGA-11	0.941	1.000	1.99	1.27
FPGA-12	0.989	1.000	2.49	1.40
Norm.	0.975	1.000	1.82	1.17

Table 2.10 shows the impact of DAA on HPWL and the maximum LUT and FF resource utilization (Equation (2.7)) in FIP stage. On average, DAA reduces the maximum LUT/FF resource utilization from 1.82 to 1.17, while increases the wirelength by 2.5%. Note that, counter-intuitively, this wirelength increase is not a quality degradation, but an improvement in the sense that FIP solutions are getting closer to truly legal placements. This can be better illustrated by Figure 2.17. It shows the HPWL after the flat initial placement (FIP), legalization/DL (LG), and detailed placement (DP) stages in the four previously described flows. In spite of the larger HPWL after FIP, flows with DAA applied finally achieve better solutions with smoother wirelength convergence. Besides, we can also see that the proposed DL technique largely preserves the FIP solution in the LG stage and this is the main reason that the proposed flow can significantly outperform the other three methodologies.
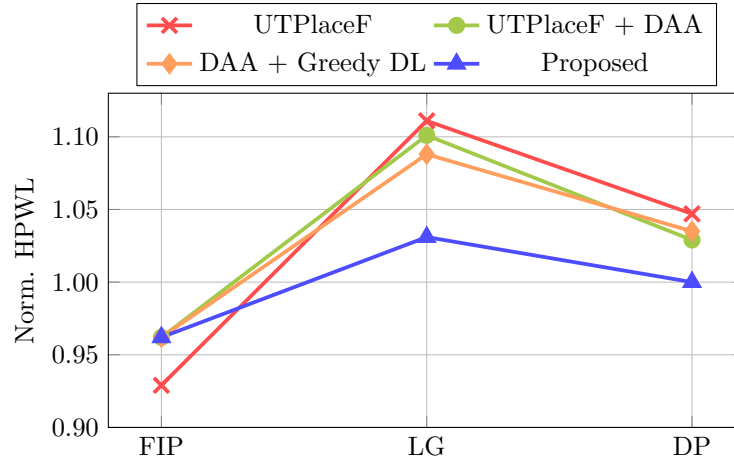
Figure 2.17: The normalized HPWL after the flat initial placement (FIP), legalization/DL (LG), and detailed placement (DP) in the four methodologies listed in Table 2.8. All HPWL values are normalized to the post-DP HPWL of the proposed flow.

2.4.4.2 Parameter Choosing in Dynamic Area Adjustment

A proper parameter setting in our dynamic area adjustment (DAA) algorithm is important to the overall solution quality. In this section, we discuss how several key parameters, including β_+, β_-, and $R_{\max}$ in Equation (2.8), should be set.

Figure 2.18 visualizes the normalized wirelength under different β_+ and β_- values based on design FPGA-01. $\beta_+ > 1$ and $\beta_- < 1$ control the rates of area inflation and shrinking, respectively, in DAA. The top-left ($\beta_+ \approx 1$ and $\beta_- \approx 1$) and bottom-right ($\beta_+ \gg 1$ and $\beta_- \ll 1$) regions in Figure 2.18 correspond to slow and fast area adjustment processes, respectively. As can be seen, the optimal wirelength is achieved at a relatively slow area adjustment rate, while the wirelength gets worse when cell areas are adjusted too fast.

84
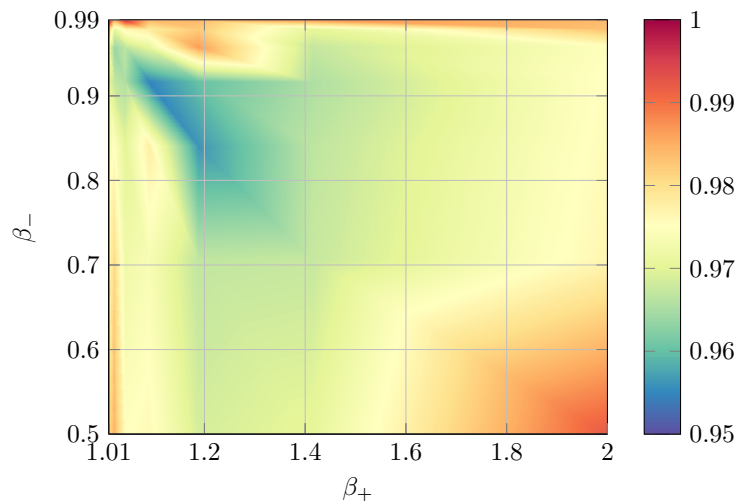
Figure 2.18: Normalized HPWL under different β_+ and β_- in Equation (2.8) based on design `FPGA-01`. All wirelengths are normalized to the solution without applying dynamic area adjustment.

This is because, for a very sharp area change, the placer typically needs several iterations to "heal" the wirelength. That is, a too fast area adjustment can be significantly harmful to the placement convergence. However, a too slow area adjustment should also be avoided in the sense that all the cell should reach their "target areas" reasonably earlier than the FIP stops to leave enough time to the placement for stabilizing. In our framework, we empirically set $\beta_+ = 1.1$ and $\beta_- = 0.95$ and we observe that the final solution quality is not very sensitive to settings around these two values.

Figure 2.19 shows the normalized HPWL and routed wirelength under different $R_{\max}$ values based on design `FPGA-05`. DAA only shrinks cells in regions with routing utilization less than $R_{\max}$ to prevent from overfilling routing congested regions. We can see that, as $R_{\max}$ increases, HPWL decreases
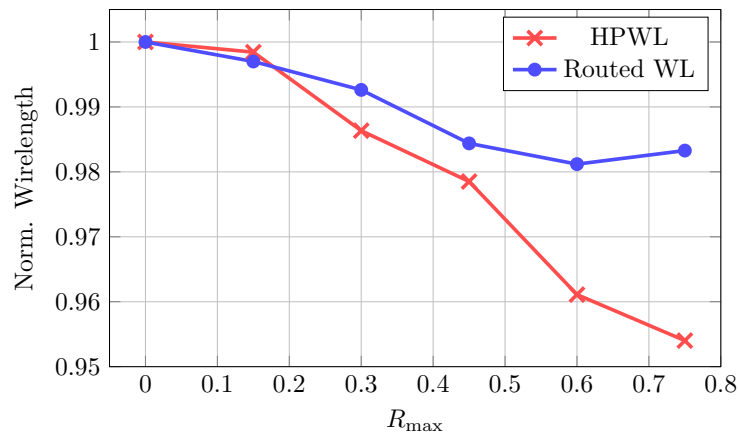
Figure 2.19: Normalized HPWL and routed wirelengths under different $R_{\max}$ in Equation (2.8) based on design `FPGA-05`. All wirelengths are normalized to the solution with $R_{\max} = 0$ (i.e., area shrinking is disallowed).

steadily due to the more aggressive cell shrinking. However, the routability get worse at the same time, and as a result, the routed wirelength reduction gradually saturates and starts to increase until an unroutable solution is reached (not shown in the figure). Therefore, a proper $R_{\max}$ is crucial to produce high-quality as well as routing-friendly solutions. In our framework, we empirically set $R_{max} = 0.65$.

2.4.4.3 Runtime Scaling of the Direct Legalization

Figure 2.20 shows the runtime scaling of our DL algorithm for different design sizes under 1, 2, 4, 8, and 16 threads. It can be seen that, with a fixed number of available threads, the algorithm scales linearly with respect to the design size. On the other hand, given a design, its runtime also decreases nearly linearly as the number of threads increases (except the 16-thread case with

86
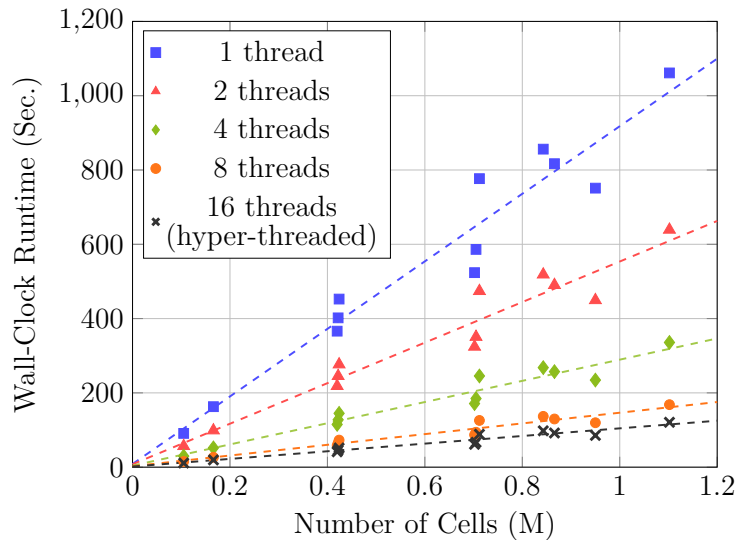
Figure 2.20: Runtime scaling of the direct legalization.

hyper-threading). On average, $1.65\times$, $3.15\times$, $6.19\times$, and $8.68\times$ speedups can be achieved with 2, 4, 8, and 16 threads, respectively, compared with the single-thread execution. Note that the scaling starts to saturate from 8 threads to 16 threads. This is because a CPU core can launch 2 hyper-threads that share the same execution resources and cannot be truly parallelized. Considering there are only 10 cores in our machine, at least 6 threads will not be running at their maximum speed in the case of 16 threads.

2.4.4.4 Comparison with Other State-of-the-Art Placers

To further demonstrate the effectiveness of UTPlaceF-DL, we also compare our result with other state-of-the-art academic placers, including RippleF-PGA [12], GPlace [62] as well as the top-3 winners of the ISPD 2016 contest, on the ISPD 2016 benchmark suite. The comparisons of routed wirelength

and runtime are presented in Table 2.11.

Like the experiment setup in Section 2.4.4.1, we enable 16 threads only for our DL algorithm and keep all other parts single-threaded. All other placers are executed using a single thread. Although other placers can also gain some performance by parallelizing their packing and legalization algorithms, the improvement might be limited. This is because most of their packing and legalization algorithms are not the runtime bottleneck, just like in UTPlaceF. For example, RippleFPGA only takes about 5% of the total runtime on packing and legalization [12].

Despite the different global/detailed placement engines and the execution machines, UTPlaceF-DL still shows the best overall routed wirelength. On average, UTPlaceF-DL outperforms the three contest winners, GPlace, and RippleFPGA by 8.0%, 14.0%, 44.4%, 25.4%, and 4.1%, respectively. Again, UTPlaceF-DL especially excels in control set-intensive designs, like `FPGA-06`, `FPGA-07`, `FPGA-09`, and `FPGA-10`. which further evidences the effectiveness of our approach on hard-to-pack designs. As for the runtime, UTPlaceF-DL is $3.96\times$, $4.90\times$, and $5.84\times$ faster than the three contest winners. While comparing with GPlace and RippleFPGA, UTPlaceF-DL runs $1.14\times$ and $1.56\times$ slower.

Table 2.11: Routed Wirelength (WL in 10^3) and Runtime (RT in seconds) Comparison with Other State-of-the-Art Academic Placers on the ISPD 2016 Benchmark Suite

Design	1st Place				2nd Place				3rd Place				GPlace [62]				RippleFPGA [12]				Proposed			
	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR
FPGA-01	*	*	-	-	380	118	1.117	1.93	582	97	1.711	1.59	494	30	1.451	0.49	353	31	1.037	0.51	340	61	1.000	1.00
FPGA-02	678	435	1.038	3.82	680	208	1.041	1.82	1047	191	1.603	1.68	903	61	1.383	0.54	645	57	0.989	0.50	653	114	1.000	1.00
FPGA-03	3223	1527	1.027	4.59	3661	1159	1.166	3.48	5029	862	1.602	2.59	3908	289	1.245	0.87	3262	201	1.039	0.60	3139	333	1.000	1.00
FPGA-04	5629	1257	1.056	3.19	6497	1449	1.219	3.68	7247	889	1.359	2.26	6278	280	1.178	0.71	5510	224	1.033	0.57	5331	394	1.000	1.00
FPGA-05	10265	1266	1.022	2.92	†	-	-	-	†	-	-	-	†	-	-	-	9969	270	0.992	0.62	10045	434	1.000	1.00
FPGA-06	6330	2920	1.091	5.05	7009	4166	1.208	7.21	6823	8613	1.176	14.90	7643	600	1.318	1.04	6180	424	1.065	0.73	5801	578	1.000	1.00
FPGA-07	10237	2703	1.094	4.08	10416	4572	1.113	6.91	10973	9196	1.173	13.89	11255	691	1.203	1.04	9640	493	1.030	0.74	9356	662	1.000	1.00
FPGA-08	8384	2645	1.010	3.81	8986	2942	1.083	4.23	12300	2741	1.482	3.94	9323	734	1.124	1.06	8157	425	0.983	0.61	8298	695	1.000	1.00
FPGA-09	†	-	-	-	13909	5833	1.196	6.58	†	-	-	-	14003	974	1.204	1.10	12305	589	1.058	0.66	11633	887	1.000	1.00
FPGA-10	*	*	-	-	*	*	-	-	†	-	-	-	†	-	-	-	7140	649	1.130	0.85	6317	766	1.000	1.00
FPGA-11	11091	3227	1.059	3.63	11713	7331	1.118	8.24	†	-	-	-	12368	923	1.181	1.04	11023	542	1.052	0.61	10476	890	1.000	1.00
FPGA-12	9022	4539	1.320	4.59	*	*	-	-	†	-	-	-	†	-	-	-	7363	650	1.077	0.66	6835	988	1.000	1.00
Norm.	-	-	1.080	3.96	-	-	1.140	4.90	-	-	1.444	5.84	-	-	1.254	0.88	-	-	1.041	0.64	-	-	1.000	1.00

*: Placement error. †: Unroutable placement.

2.4.4.5 Runtime Breakdown

The runtime breakdown of UTPlaceF-DL based on all twelve designs in the ISPD 2016 benchmark suite is shown in Figure 2.21. Again, we enable 16 threads only for DL and keep all other parts single-threaded. On average, 64.5% and 18.5% of the total runtime are taken by the quadratic placement (quadratic programming and rough legalization) and the detailed placement, respectively. While the proposed dynamic area adjustment, direct legalization, and post-DL exception handling techniques consume 2.2%, 12.5%, and 0.7% of the total runtime.
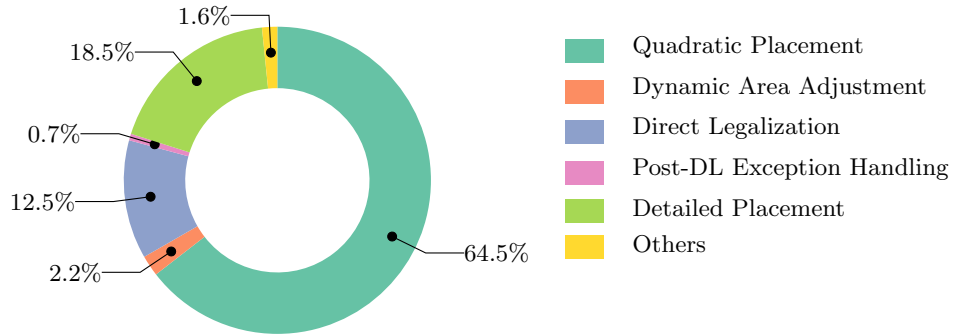
Figure 2.21: The runtime breakdown of UTPlaceF-DL based on designs in the ISPD 2016 benchmark suite.

2.4.5 Summary

In this section, we have proposed a new paradigm, UTPlaceF-DL, for FPGA placement without explicit packing. The proposed framework significantly improves the correlation between early placements and final legal solutions. To realize the proposed framework, a dynamic LUT and FF area

adjustment technique and a fully parallelizable direct legalization algorithm are proposed. Our experiments on the ISPD 2016 benchmark suite demonstrate the effectiveness of the proposed approach. As this is the first work to perform FPGA placement without explicit packing, we expect more research to be done to further improve the quality of results.

2.5   elfPlace: Electrostatics-based Placement for Large-Scale Heterogeneous FPGAs

There are various core FPGA placement algorithms have been proposed in the literature and analytical placers are among the most efficient and effective approaches. Analytical placers can be subdivided into quadratic placers and nonlinear placers. Quadratic approaches, like UTPlaceF and UTPlaceF-DL presented in Section 2.3 and Section 2.4 as well as works in [3, 12, 24, 25, 77, 78], approximate the placement objective using quadratic functions, while nonlinear approaches [14, 34, 50] use higher-order ones. Compared with quadratic approaches, nonlinear approaches often achieve better solution quality due to their even stronger expressive power.

In contrast to the enormous research endeavor spent on core placement algorithms, there are still very limited works coping with resource heterogeneity issue in FPGA. Most existing analytical placers only treat highly-discrete DSP and RAM blocks specially and eliminate the heterogeneity between LUTs and FFs by either spreading them together with adjusted areas [3,12,40] or simply packing them before placement [14]. These approaches are usually highly sensitive to the heuristics applied, which could hamper the solution quality and placement robustness. A more recent work [19] proposed a multi-commodity flow-based algorithm for quadratic placers to spread heterogeneous cells and it demonstrated significant improvement over previous spreading heuristics. However, due to the inherent limitation of the quadratic placement, their approach still simplifies spreading as a movement-minimization problem, which

cannot explicitly optimize wirelength nor preserve the relative order among cells of different resource types.

There are also works on optimizing other placement objectives to ease the downstream packing, legalization, and routing steps. Many works [3, 12] adopted the cell inflation technique to alleviate routing congestions. Our UTPlaceF-DL, presented in Section 2.4, further considered the impact of downstream packing/legalization and adjusted cell areas accordingly during placement to improve the overall solution quality. However, all these works were originally proposed for quadratic placers, studies on extending them to more powerful nonlinear placement are still lacking.

In this section, we present elfPlace, a general, flat, nonlinear placement algorithm for large-scale heterogeneous FPGAs. elfPlace adopts the idea of casting placement to electrostatic system initially proposed by ePlace family [16, 55] for application-specific integrated circuits (ASICs), and it is enhanced to tackle the FPGA heterogeneity issue in a unified and elegant way. Besides the conventional wirelength objective, elfPlace also performs routability, pin density, and packing-aware optimizations to achieve even higher-quality and smoother design closure. Our major contributions are summarized as follows.

- We enhance the original ePlace algorithm [16, 55] for ASICs to deal with heterogeneous resource types in FPGAs.

- We employ augmented Lagrangian method, instead of the multiplier

93

method used in ePlace, to formulate the nonlinear placement problem.

- We propose a preconditioning technique to improve the numerical convergence given the wide spectrum of cell sizes and net degrees in FPGA designs.

- We propose a normalized subgradient method to update density penalty multipliers, which control the spreading of different resource types in a self-adaptive manner.

- We improve the packing-aware area adjustment technique proposed in [44] and integrate it, together with routability and pin density optimizations, into elfPlace.

- We demonstrate more than 7% improvement in routed wirelength, on the ISPD 2016 benchmark suite [79], over four cutting-edge placers with very competitive runtime.

The rest of this section is organized as follows. Section 2.5.1 introduces the background knowledge. Section 2.5.2 sketches the overall flow of elfPlace. Section 2.5.3 describes the core placement algorithms and Section 2.5.4 details the routability, pin density, and packing-aware optimizations. Section 2.5.5 shows the experimental results, followed by the summary in Section 2.5.6.

2.5.1   Preliminaries

2.5.1.1   The ePlace Algorithm

ePlace [16, 55] is a leading-edge nonlinear global placement algorithm for ASICs. It approximates half-perimeter wirelength (HPWL),

$$W(\boldsymbol{x}, \boldsymbol{y}) = \sum_{e \in \mathcal{E}} W_e(\boldsymbol{x}, \boldsymbol{y}) = \sum_{e \in \mathcal{E}} \left(\max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j| \right), \qquad (2.14)$$

using the weighted-average (WA) model,

$$\widetilde{W}_{e_x}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\sum_{i \in e} x_i \exp(x_i/\gamma)}{\sum_{i \in e} \exp(x_i/\gamma)} - \frac{\sum_{i \in e} x_i \exp(-x_i/\gamma)}{\sum_{i \in e} \exp(-x_i/\gamma)}. \qquad (2.15)$$

Here $\boldsymbol{x}$ and $\boldsymbol{y}$ denote the cell locations, $\mathcal{E}$ denotes the set of nets in the design, and γ is a parameter to control the modeling smoothness and accuracy. Equation (2.15) only gives the x-directed WA model of a net and the total wirelength cost is defined as $\widetilde{W}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{e \in \mathcal{E}}(\widetilde{W}_{e_x}(\boldsymbol{x}, \boldsymbol{y}) + \widetilde{W}_{e_y}(\boldsymbol{x}, \boldsymbol{y}))$.

The key innovation of ePlace is that it casts the placement density cost to the potential energy of an electrostatic system. With this transformation, each cell i is modeled as a positive charge q_i with the quantity proportional to its area. Given the notations defined in Table 2.12, the electric force $\boldsymbol{F}_i = q_i \boldsymbol{\xi}_i = -q_i \nabla \psi_i$ will guide each charge i towards the direction of minimizing the total potential energy Φ,

$$\Phi(\boldsymbol{x}, \boldsymbol{y}) = \iint_R \rho(x, y) \psi(x, y), (x, y) \in R. \qquad (2.16)$$

Table 2.12: Notations used in the electrostatic system

R	A finite two-dimensional region
q_i	The electric charge quantity of charge i
$\rho(x, y)$	The electric charge density at $(x, y) \in R$
$\psi_i, \psi(x, y)$	The electric potential at charge i and $(x, y) \in R$
$\boldsymbol{\xi}_i, \boldsymbol{\xi}(x, y)$	The electric field at charge i and $(x, y) \in R$
$\Phi(\boldsymbol{x}, \boldsymbol{y})$	The total electric potential energy of placement $(\boldsymbol{x}, \boldsymbol{y})$

The unique solution of the electrostatic system is given by Equation (2.17).

$$\begin{cases} \nabla \cdot \nabla \psi(x, y) = -\nabla \cdot \boldsymbol{\xi}(x, y) = -\rho(x, y), (x, y) \in R, & (2.17a) \\[2mm] \hat{\boldsymbol{n}} \cdot \nabla \psi(x, y) = -\hat{\boldsymbol{n}} \cdot \boldsymbol{\xi}(x, y) = \boldsymbol{0}, (x, y) \in \partial R, & (2.17b) \\[2mm] \iint_R \rho(x, y) = \iint_R \psi(x, y) = 0, (x, y) \in R, & (2.17c) \end{cases}$$

where Equation (2.17a) is the Poisson's equation to relate electric potential, electric field, and charge density, Equation (2.17b) is Neumann boundary condition (i.e., zero electric fields on the boundary of R) to prevent charges from moving out of R, and Equation (2.17c) neutralizes the overall electric charge and potential to ensure the solution uniqueness of Equation (2.17). ePlace honors the placement density constraint by enforcing the electrostatic equilibrium state, where electric density is evenly distributed and $\Phi(\boldsymbol{x}, \boldsymbol{y}) = 0$.

ePlace computes the numerical solution of Equation (2.17) using spectral methods. It divides the placement region into a grid of $m \times m$ bins to construct the charge density map ρ, then the electric potential ψ and electric

field $\boldsymbol{\xi} = (\xi_x, \xi_y)$ can be obtained as follows.

$$a_{u,v} = \frac{1}{m^2} \sum_{x=0}^{m-1} \sum_{y=0}^{m-1} \rho(x,y) \cos(\omega_u x) \cos(\omega_v y), \qquad (2.18\text{a})$$

$$\psi(x,y) = \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} \frac{a_{u,v}}{\omega_u^2 + \omega_v^2} \cos(\omega_u x) \cos(\omega_v y), \qquad (2.18\text{b})$$

$$\xi_x(x,y) = \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} \frac{a_{u,v}\omega_u}{\omega_u^2 + \omega_v^2} \sin(\omega_u x) \cos(\omega_v y), \qquad (2.18\text{c})$$

$$\xi_y(x,y) = \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} \frac{a_{u,v}\omega_v}{\omega_u^2 + \omega_v^2} \cos(\omega_u x) \sin(\omega_v y). \qquad (2.18\text{d})$$

Here x and y are bin indexes, u and v denote frequency indexes from 0 to $m-1$, and $\omega_u = \frac{2\pi u}{m}$ and $\omega_v = \frac{2\pi v}{m}$ are the frequencies of $\sin/\cos$ wave functions. Equation (2.18) can be efficiently computed using discrete cosine transform (DCT) and its inverse (IDCT).

Finally, with both wirelength cost $\widetilde{W}(\boldsymbol{x}, \boldsymbol{y})$ and density penalty $\Phi(\boldsymbol{x}, \boldsymbol{y})$ well defined, ePlace then iteratively solves the following unconstrained nonlinear optimization problem using multiplier method,

$$\min_{\boldsymbol{x},\boldsymbol{y}} \ f(\boldsymbol{x}, \boldsymbol{y}) = \widetilde{W}(\boldsymbol{x}, \boldsymbol{y}) + \lambda \Phi(\boldsymbol{x}, \boldsymbol{y}), \qquad (2.19)$$

where λ is the density penalty multiplier to progressively enforce the density constraint.

2.5.2 elfPlace Overview

One major challenge of FPGA placement is the heterogeneity handling. elfPlace tackles this problem by maintaining separate electrostatic systems for different resource types, including LUT, FF, DSP, and RAM. The notations used in elfPlace are given in Table 2.13.

Table 2.13: Notations used in elfPlace

$\mathcal{S}$	The resource type set {LUT, FF, DSP, RAM}
$\mathcal{V}, \mathcal{V}_s$	The cell set and its subset with resource type s
$\mathcal{V}^{\mathrm{P}}, \mathcal{V}_s^{\mathrm{P}}$	The physical cell set and its subset of resource type s
$\mathcal{V}^{\mathrm{F}}, \mathcal{V}_s^{\mathrm{F}}$	The filler cell set and its subset of resource type s
A_i	The area of cell i
$\mathcal{B}_s$	The bin grid for resource type $s \in \mathcal{S}$
A_b^{P}	The physical cell area in bin b
C_b	The resource capacity in bin b
$\boldsymbol{\lambda}$	The density multiplier vector $(\lambda_{\mathrm{LUT}}, \lambda_{\mathrm{FF}}, \lambda_{\mathrm{DSP}}, \lambda_{\mathrm{RAM}})^T$
$\boldsymbol{\Phi}$	The potential energy vector $(\Phi_{\mathrm{LUT}}, \Phi_{\mathrm{FF}}, \Phi_{\mathrm{DSP}}, \Phi_{\mathrm{RAM}})^T$
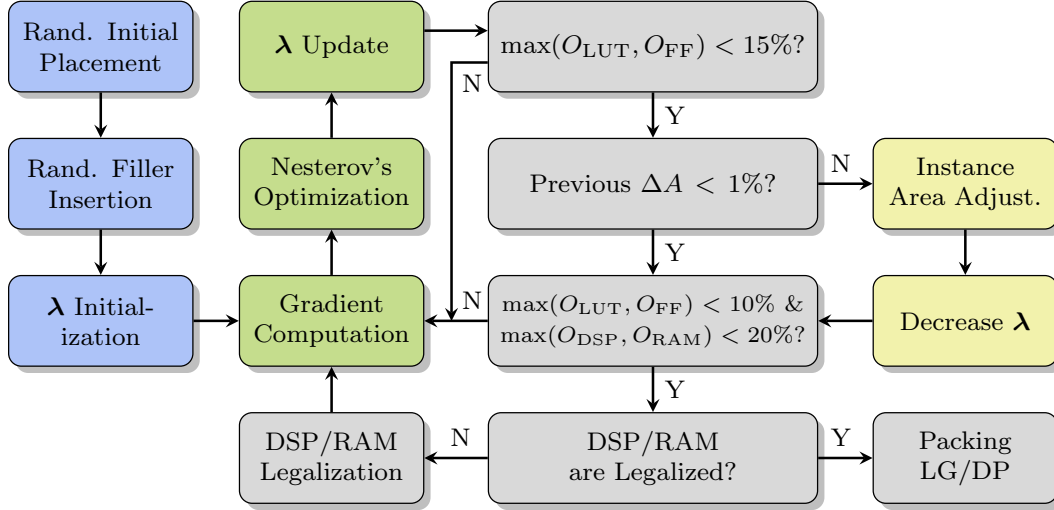
Figure 2.22: The overall flow of elfPlace.

Figure 2.22 illustrates the overall flow of elfPlace. Different from typical initial placement approaches that minimize wirelength by quadratic programming, elfPlace starts from a random initial placement, which has been observed to achieve nearly the same quality [51]. In the random initial placement, all movable cells are first placed at the centroid of fixed pins and an extra Gaussian noise perturbation is injected with standard deviation equal to 0.1% of the width and height of the placement region.

After the initial placement, filler cells are created and inserted independently for each resource type. Fillers are needed to pad whitespaces and produce compact placement solutions. For each resource type $s \in \mathcal{S}$ with the bin grid $\mathcal{B}_s$, its total filler area is computed as $\sum_{b \in \mathcal{B}_s} C_b - \sum_{i \in \mathcal{V}_s^{\mathrm{P}}} A_i$. In our experiments, LUT/FF fillers are set to be squares with 1/8 CLB area, and DSP and RAM fillers are set to be rectangles with dimensions 1.0×2.5 and 1.0×5.0 (CLB width), respectively, based on our target FPGA architecture. For each resource type s, fillers are randomly inserted based on the resource capacity distribution. More specifically, elfPlace first randomly distributes fillers of resource type s into bins based on the probabilities $C_b / \sum_{b \in \mathcal{B}_s} C_b$, and their final locations are then uniformly drawn within bins. By this insertion strategy, fillers can start with relatively low potential energies and it improves the convergence and stability of the later placement optimization.

Based on the initial placement and filler insertion, elfPlace initializes the density multiplier vector $\boldsymbol{\lambda} = (\lambda_{\mathrm{LUT}}, \lambda_{\mathrm{FF}}, \lambda_{\mathrm{DSP}}, \lambda_{\mathrm{RAM}})^T$ and then enters the core placement optimization phase. In each placement iteration, the gra-

dient of a wirelength-density co-optimization problem is computed and fed to a Nesterov's optimizer [55] to take a descent step. After that, $\boldsymbol{\lambda}$ is updated to balance the spreading efforts on different resource types and universally emphasize slightly more density penalties. When both LUT and FF overflows (O_{LUT} and O_{FF}) are reduced down to 15%, elfPlace adjusts cell areas with the consideration of routability, pin density, and downstream packing compatibility (i.e., LUT input pin constraint and FF control set constraint described in Section 2.2). After the cell area adjustment, the nearly-equilibrated electrostatic states are likely to be damaged, therefore, elfPlace reduces the density multipliers $\boldsymbol{\lambda}$, in this case, to recover the quality again. This area adjustment step is performed each time that LUT/FF converge to $\max(O_{\text{LUT}}, O_{\text{FF}}) < 15\%$ until the total area change is less than 1%. The overflow of each resource type s is given in Equation (2.20).

$$O_s = \frac{\sum_{b \in \mathcal{B}_s} \max(A_b^{\text{P}} - C_b, 0)}{\sum_{b \in \mathcal{B}_s} A_b^{\text{P}}}, \forall s \in \mathcal{S}. \tag{2.20}$$

Once the cell area converges and the overlaps are small enough for all resource types, i.e., $\max(O_{\text{LUT}}, O_{\text{FF}}) < 10\%$ and $\max(O_{\text{DSP}}, O_{\text{RAM}}) < 20\%$, elfPlace legalizes and fixes DSP and RAM blocks using the minimum-cost flow approach like in [12, 40]. Here we set a larger overflow target for DSP/RAM due to their much higher discreteness compared with LUT/FF. After that, LUT/FF placements are further optimized until they both meet the overflow target again ($\max(O_{\text{LUT}}, O_{\text{FF}}) < 10\%$). Finally, elfPlace adopts the packing, legalization, and detailed placement approaches in UTPlaceF-DL (Section 2.4)

to produce the final legal solution.

2.5.3   Core Placement Algorithms

2.5.3.1   The Augmented Lagrangian Formulation

With the density constraint for each resource type modeled as a separate electrostatic system, elfPlace solves the minimization problem defined as follows:

$$\min_{\boldsymbol{x},\boldsymbol{y}} \widetilde{W}(\boldsymbol{x},\boldsymbol{y}) \quad \text{s.t.} \ \Phi_s(\boldsymbol{x},\boldsymbol{y}) = 0, \forall s \in \mathcal{S}. \tag{2.21}$$

However, unlike ePlace solves the density constrained placement problem using the multiplier method given in Equation (2.19), elfPlace uses the augmented Lagrangian method (ALM), as shown in Equation (2.22), instead.

$$\min_{\boldsymbol{x},\boldsymbol{y}} \ f(\boldsymbol{x},\boldsymbol{y}) = \widetilde{W}(\boldsymbol{x},\boldsymbol{y}) + \sum_{s \in \mathcal{S}} \lambda_s \Big(\Phi_s(\boldsymbol{x},\boldsymbol{y}) + \frac{c_s}{2}\Phi_s(\boldsymbol{x},\boldsymbol{y})^2\Big). \tag{2.22}$$

Here λ_s and Φ_s are density multiplier and electric potential energy for each resource type $s \in \mathcal{S} = \{\text{LUT, FF, DSP, RAM}\}$, and c_s is a parameter to control the relative weight of the quadratic penalty term $\Phi_s(\boldsymbol{x},\boldsymbol{y})^2$. Slightly different from the typical ALM formulation where $\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y})^2$ has its weight independent to $\boldsymbol{\lambda}$, the magnitude of $\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y})^2$ is also determined by $\boldsymbol{\lambda}$ in Equation (2.22). This is to better control the overall effort on honoring the density constraints and make elfPlace less sensitive to the initial placement.

The ALM formulation in Equation (2.22) can be viewed as a mixture of the multiplier method and the penalty method. The motivation is that, when the resource type s has a high potential energy $\Phi_s(\boldsymbol{x},\boldsymbol{y})$, we want the

penalty term $\frac{c_s}{2}\Phi_s(\boldsymbol{x}, \boldsymbol{y})^2$ to dominate (i.e., $\frac{c_s}{2}\Phi_s(\boldsymbol{x}, \boldsymbol{y})^2 \gg \Phi_s(\boldsymbol{x}, \boldsymbol{y})$) and make Equation (2.22) become the penalty method as shown in Equation (2.23). Since in this case, the resource type s still has lots of overlaps, using the penalty method can enhance the convexity of the objective function and improve the convergence.

$$\min_{\boldsymbol{x}, \boldsymbol{y}} \ f_{\mathrm{PM}}(\boldsymbol{x}, \boldsymbol{y}) = \widetilde{W}(\boldsymbol{x}, \boldsymbol{y}) + \sum_{s \in \mathcal{S}} \lambda_s \frac{c_s}{2} \Phi_s(\boldsymbol{x}, \boldsymbol{y})^2. \tag{2.23}$$

On the other hand, when the resource type s converges to a relatively small potential energy $\Phi_s(\boldsymbol{x}, \boldsymbol{y})$, we want the $\Phi_s(\boldsymbol{x}, \boldsymbol{y})$ term to dominate (i.e., $\Phi_s(\boldsymbol{x}, \boldsymbol{y}) \gg \frac{c_s}{2}\Phi_s(\boldsymbol{x}, \boldsymbol{y})^2$) and make Equation (2.22) become the multiplier method as shown in Equation (2.24). In this case, the overlaps of the resource type s are already relatively small and using the multiplier method can continue the optimization without suffering the ill-conditioning problem associated with the penalty method.

$$\min_{\boldsymbol{x}, \boldsymbol{y}} \ f_{\mathrm{MM}}(\boldsymbol{x}, \boldsymbol{y}) = \widetilde{W}(\boldsymbol{x}, \boldsymbol{y}) + \sum_{s \in \mathcal{S}} \lambda_s \Phi_s(\boldsymbol{x}, \boldsymbol{y}). \tag{2.24}$$

The key of achieving this penalty method and multiplier method trade-off is to properly set the value of $c_s, \forall s \in \mathcal{S}$. We observe that, regardless of the design size, the final potential energy always converges to 10^{-5} to 10^{-7} of the initial one. Therefore, we define c_s as follows:

$$c_s = \frac{\beta}{\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})}, \forall s \in \mathcal{S}, \tag{2.25}$$

where β is set to 2×10^3 in our experiments and $\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})$ is the po-

tential energy of the random initial placement (described in Section 2.5.2). Under this setting, we will have $\frac{c_s}{2}\Phi_s(\boldsymbol{x}, \boldsymbol{y})^2 = \Phi_s(\boldsymbol{x}, \boldsymbol{y})$ when $\Phi_s(\boldsymbol{x}, \boldsymbol{y}) = 10^{-3}\,\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})$. The experimental result in Section 2.5.5.2 shows that our ALM formulation could improve the final routed wirelength by 1.2% compared with the original multiplier method adopted in ePlace.

2.5.3.2 Gradient Computation and Preconditioning

The x-directed gradient of our objective function defined in Equation (2.22) can be derived as shown in Equation (2.26). For brevity, only x-direction will be discussed in the rest of this section and similar conclusions are applicable to y-direction as well.

$$
\begin{aligned}
\frac{\partial f(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i} &= \frac{\partial \widetilde{W}(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i} + \lambda_s\Big(\frac{\partial \Phi_s(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i} + c_s\Phi_s(\boldsymbol{x}, \boldsymbol{y})\frac{\partial \Phi_s(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i}\Big) \\
&= \frac{\partial \widetilde{W}(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i} - \lambda_s q_i \xi_{x_i}\Big(1 + c_s\Phi_s(\boldsymbol{x}, \boldsymbol{y})\Big), \quad \forall i \in \mathcal{V}_s.
\end{aligned}
\tag{2.26}
$$

Although our ALM-based density penalty term is initially motivated by mathematics, there are still physical intuitions behind its gradient. By the nature of electrostatics, the electric force $q_i\boldsymbol{\xi}_i$ on each charge i will guide the charge towards a nearby low-potential well and this is reflected by the $\lambda_s q_i\boldsymbol{\xi}_{x_i}$ term in Equation (2.26). Besides, the extra $\lambda_s q_i\boldsymbol{\xi}_{x_i}c_s\Phi_s(\boldsymbol{x}, \boldsymbol{y})$ term further accelerates the charge movement for resource types with high potential energies, which often correspond to relatively large cell overlaps in the placement problem.

The gradient defined in Equation (2.26) will be preconditioned before being finally fed to the optimizer. Preconditioning can make the local curva-

103

ture of the objective function become nearly spherical, and hence, alleviate the ill-conditioning problem and improve the numerical convergence and stability. The most commonly used preconditioner is the inverse of the Hessian matrix $\boldsymbol{H}_f$ of the objective function f, and the preconditioned gradient $\boldsymbol{H}_f^{-1}\nabla f$, instead of the original ∇f, will be used as the (opposite of) descent direction. However, due to the scale of the placement problem and the complexity of our objective function, it is impractical to compute the exact Hessian. Instead, elfPlace adopts the much cheaper Jacobi preconditioner to approximate the actual Hessian.

The x-directed Jacobi preconditioner is a diagonal matrix with the i-th diagonal entry equal to $\frac{\partial^2 f}{\partial x_i^2}$. By Equation (2.26), we have

$$\frac{\partial^2 f(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i^2} = \frac{\partial^2 \widetilde{W}(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i^2} - \lambda_s q_i \Big(\frac{\partial \xi_{x_i}}{\partial x_i} \Big(1 + c_s \Phi(\boldsymbol{x}, \boldsymbol{y}) \Big) - c_s \xi_{x_i}^2 \Big). \qquad (2.27)$$

The closed-form expression of $\frac{\partial^2 \widetilde{W}(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i^2}$ is too expensive to compute in practice, therefore, we approximate it using

$$\frac{\partial^2 \widetilde{W}(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i^2} \sim \sum_{e \in \mathcal{E}_i} \frac{1}{|e| - 1}, \qquad (2.28)$$

where $\mathcal{E}_i$ denotes the set of nets incident to cell i and $|e|$ denotes the degree of the net e. The second-order derivative of the density term is even more complicated. Although the numerical solution of $\frac{\partial \xi_{x_i}}{\partial x_i}$ can be computed again through spectral methods based on Equation (2.18), we choose to only keep the $\lambda_s q_i$ term for the sake of efficiency.

Therefore, the overall x-directed second-order derivative of the objective is approximated as follows:

$$\frac{\partial^2 f(\boldsymbol{x}, \boldsymbol{y})}{\partial x_i^2} \sim h_{x_i} = \max\left(\sum_{e \in \mathcal{E}_i} \frac{1}{|e| - 1} + \lambda_s q_i,\ 1\right), \forall i \in \mathcal{V}_s, \qquad (2.29)$$

where the $\max(\cdot, 1)$ is to avoid extremely small h_{x_i} for filler cells, who do not have incident nets, when λ_s is very small. Finally, the preconditioned gradient,

$$H_f^{-1} \nabla f(\boldsymbol{x}, \boldsymbol{y}) = \left(\frac{1}{h_{x_1}} \frac{\partial f(\boldsymbol{x}, \boldsymbol{y})}{\partial x_1}, \frac{1}{h_{y_1}} \frac{\partial f(\boldsymbol{x}, \boldsymbol{y})}{\partial y_1}, \cdots\right)^T, \qquad (2.30)$$

will be fed to a Nesterov's optimizer [54] to iteratively update the placement solution. Since in FPGA designs, net degrees (e.g., local signal nets and global clock nets) and cell pin counts and sizes (e.g., small LUT/FF cells and large DSP/RAM blocks) can vary significantly, this wirelength preconditioner is essential to the numerical convergence of our optimization. The experimental result in Section 2.5.5.2 shows that elfPlace can barely converge without our preconditioning technique.

2.5.3.3   Density Multipliers Setting

One important thing we have not yet discussed is the setting of density multipliers $\boldsymbol{\lambda}$, which control the spreading efforts on different resource types. Since there is often heavy connectivity among different resource types, the spreading process must be capable of achieving target densities for all resource types while not ruining the natural physical clusters consisting of heterogeneous cells.

In elfPlace, we set the initial density multipliers $\boldsymbol{\lambda}^{(0)}$ as follows:

$$\boldsymbol{\lambda}^{(0)} = \eta \frac{\|\nabla \widetilde{W}(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})\|_1}{\sum_{i \in \mathcal{V}} q_i \|\boldsymbol{\xi}_i^{(0)}\|_1} \left(1, 1, \cdots, 1\right)^T, \tag{2.31}$$

where $(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})$ represent the initial placement, $\boldsymbol{\xi}_i^{(0)}$ denotes the initial electric field at cell i, η is a weighting parameter, and $\| \cdot \|_1$ denotes the L1-norm of a vector. In order to emphasize the wirelength optimization in early iterations, η is set to 10^{-4} in our experiments. Note that $\boldsymbol{\lambda}^{(0)}$ is an $|\mathcal{S}|$-dimensional vector, where $|\mathcal{S}|$ is the number of resource types, and by Equation (2.31), we start from spreading all resource types with the same weight.

Classical optimization approaches use the subgradient method to update $\boldsymbol{\lambda}$ [37]. According to Equation (2.22), the subgradient of $\boldsymbol{\lambda}$ is defined as

$$\nabla_{\text{sub}} \boldsymbol{\lambda} = \left(\cdots, \Phi_s(\boldsymbol{x}, \boldsymbol{y}) + \frac{c_s}{2} \Phi_s(\boldsymbol{x}, \boldsymbol{y})^2, \cdots\right)^T, s \in \mathcal{S}. \tag{2.32}$$

The reason why $\nabla_{\text{sub}} \boldsymbol{\lambda}$ is called subgradient instead of gradient is that the dual function, $l(\boldsymbol{\lambda}) = \max f(\boldsymbol{x}, \boldsymbol{y})|_{\boldsymbol{\lambda}}$, associated with Equation (2.22) is not smooth but piecewise linear [37].

However, in our placement problem, the potential energies of different resource types, Φ_s, can differ by order of magnitudes. The very sparse DSP/RAM blocks usually have significantly smaller total potential energies compared with LUT/FF cells. As a result, using the subgradient in Equation (2.32) to guide the $\boldsymbol{\lambda}$ updating can lead to severely ill-conditioned problems. To mitigate this issue, the normalized subgradient defined in Equa-

106

tion (2.33) is used instead in elfPlace.

$$\widehat{\nabla}_{\mathrm{sub}}\boldsymbol{\lambda} = \left(\cdots, \frac{1}{\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})}\left(\Phi_s(\boldsymbol{x},\boldsymbol{y}) + \frac{c_s}{2}\Phi_s(\boldsymbol{x},\boldsymbol{y})^2\right), \cdots\right)^T,$$

$$= \left(\cdots, \widehat{\Phi}_s(\boldsymbol{x},\boldsymbol{y}) + \frac{\beta}{2}\widehat{\Phi}_s(\boldsymbol{x},\boldsymbol{y})^2, \cdots\right)^T, s \in \mathcal{S}. \tag{2.33}$$

Here we use $\widehat{\Phi}_s(\boldsymbol{x},\boldsymbol{y}) = \Phi_s(\boldsymbol{x},\boldsymbol{y})/\Phi_s(\boldsymbol{x}^{(0)},\boldsymbol{y}^{(0)})$ to denote the potential energy normalized by the potential energy of the initial placement and c_s is replaced by its definition given in Equation (2.25). After this normalization, each $\widehat{\Phi}_s(\boldsymbol{x},\boldsymbol{y})$ is approximately upper bounded by 1, which can more accurately reflect the relative level of density violation for each resource type.

Given the $\boldsymbol{\lambda}^{(k)}$ and the step size $t^{(k)}$ at iteration k, we compute $\boldsymbol{\lambda}^{(k+1)}$ by Equation (2.34) and our step size updating scheme is further presented by Equation (2.35).

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + t^{(k)}\frac{\widehat{\nabla}_{\mathrm{sub}}\boldsymbol{\lambda}^{(k)}}{\|\widehat{\nabla}_{\mathrm{sub}}\boldsymbol{\lambda}^{(k)}\|_2}. \tag{2.34}$$

$$t^{(k)} = \begin{cases} \alpha_{\mathrm{H}} - 1, & \text{for } k = 0, \\ t^{(k-1)}\left(\frac{\log(\beta\|\widehat{\Phi}^{(k)}\|_2 + 1)}{1 + \log(\beta\|\widehat{\Phi}^{(k)}\|_2 + 1)}(\alpha_{\mathrm{H}} - \alpha_{\mathrm{L}}) + \alpha_{\mathrm{L}}\right), & \text{for } k > 0. \end{cases} \tag{2.35}$$

Here β is the same weighting parameter used in Equation (2.25) and the parameter pair $(\alpha_{\mathrm{L}}, \alpha_{\mathrm{H}})$ defines the range of increasing rate of the step size. The motivation of our step size updating scheme shown in Equation (2.35) is that the quadratic density penalty term often decays much faster than the linear penalty term in our objective Equation (2.22). Therefore, we incline to increase the step size faster when the quadratic penalty term dominates (i.e., $\beta\|\widehat{\Phi}^{(k)}\|_2 \gg 1$). As the cell overlaps become smaller, the linear penalty

107

term will start to take over and a slower increasing rate will be used in this case. In our experiments, we set (α_L, α_H) to $(1.05, 1.06)$. It should be noted that, although $1.05 \approx 1.06$, their high-order exponents can differ by order of magnitudes (e.g., $(1.06/1.05)^{500} > 100$).

(a)

(b)
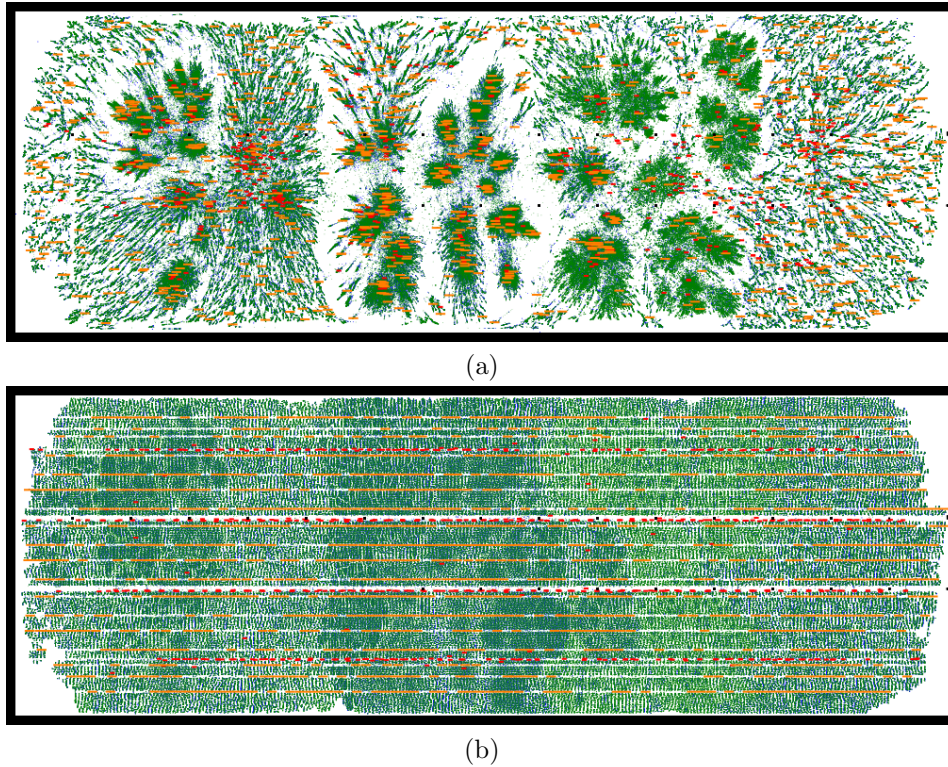
Figure 2.23: The distributions of physical LUT (green), FF (blue), DSP (red), and RAM (orange) cells in (a) an intermediate placement and (b) the placement right before DSP/RAM legalization based on `FPGA-10`. Both figures are rotated by 90 degrees.

Figure 2.23 illustrates the heterogeneous spreading process in elfPlace. As can be seen from Figure 2.23(a), our $\boldsymbol{\lambda}$ updating scheme can greatly preserve those natural physical clusters consisting of heterogeneous cells. The

placement right before DSP/RAM legalization shown in Figure 2.23(b) further demonstrates the capability of elfPlace to achieve nearly overlap-free solutions even for highly-discrete DSP/RAM blocks without explicit legalization.

2.5.4   Cell Area Adjustment

Besides minimizing wirelength, elfPlace is also capable of tackling other practical issues in real-world designs, such as routability, pin density, and packing compatibility. Routability and pin density optimizations have always been the fundamental requirements of placement to achieve routing-friendly solutions. While packing compatibility optimization, which was also discussed in UTPlaceF-DL (Section 2.4), is to further consider the effect of downstream packing early in the placement stage. It turns out that all these issues can be addressed by properly adjusting cell areas on top of our wirelength-driven placement. Therefore, in this section, we propose a unified cell area adjustment approach to simultaneously optimize all of them.

2.5.4.1   The Adjustment Scheme

Figure 2.24 sketches the algorithm flow of our cell area adjustment. In the beginning, for each physical cell i, we first compute three independent cell areas one each is optimized for routability (A_i^{ro}), pin density (A_i^{po}), and packing compatibility (A_i^{pko}). Let A_i denote the area of cell i before the adjustment, we define the target area increase of each physical cell i as follows:

$$\Delta A_i = \max(A_i^{\mathrm{ro}}, A_i^{\mathrm{po}}, A_i^{\mathrm{pko}}, A_i) - A_i, \forall i \in \mathcal{V}^{\mathrm{P}}. \tag{2.36}$$
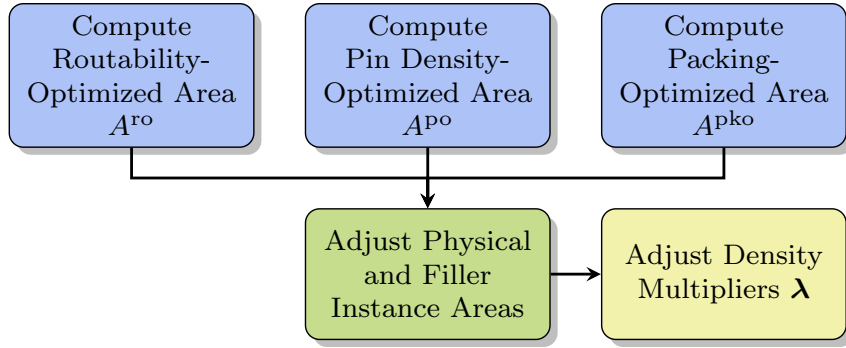
Figure 2.24: The area adjustment flow in elfPlace to simultaneously optimize routability, pin density, and downstream packing compatibility.

In order to prevent the total adjusted area from exceeding the total capacity of each resource type, all ΔA_i need to be further scaled by the following factor according to the resource type s of i.

$$\tau_s = \min\left(\frac{\sum_{i \in \mathcal{V}_s^{\mathrm{F}}} A_i}{\sum_{i \in \mathcal{V}_s^{\mathrm{P}}} \Delta A_i}, 1\right), \forall s \in \mathcal{S}, \tag{2.37}$$

where $\mathcal{V}_s^{\mathrm{F}}$ denotes the set of filler cells for resource type s and $\sum_{i \in \mathcal{V}_s^{\mathrm{F}}} A_i$ is the total filler area of resource type s before the adjustment. Basically, scaling all ΔA_i by Equation (2.37) guarantees that the total increased physical cell area is no greater than the total available filler area for each resource type. The final adjusted area A_i' of each physical cell i is then given by Equation (2.38).

$$A_i' = A_i + \tau_s \Delta A_i, \forall i \in \mathcal{V}_s^{\mathrm{P}}, \forall s \in \mathcal{S}. \tag{2.38}$$

Recall that elfPlace relies on electrostatic neutrality to meet the density constraints $\boldsymbol{\Phi} = \mathbf{0}$, therefore, we also need to downsize filler cells to maintain the total positive charge quantity unchanged. The final adjusted area A_i' of

each filler cell i is then defined as follows:

$$A_i' = \frac{\sum_{j \in \mathcal{V}_s} A_j - \sum_{j \in \mathcal{V}_s^{\mathrm{P}}} A_j'}{|\mathcal{V}_s^{\mathrm{F}}|}, \forall i \in \mathcal{V}_s^{\mathrm{F}}, \forall s \in \mathcal{S}. \tag{2.39}$$

Our area adjustment step is physically equivalent to redistributing charge density with the overall electrostatic neutrality preserved. After this redistribution, however, the previously nearly-equilibrated electrostatic states are likely to be ruined. In addition, the adjustment magnitudes and the potential energy increases can be highly uneven across different resource types (e.g., FFs can vary more than LUTs due to the control set rules). Therefore, we reset the density multipliers $\boldsymbol{\lambda}$ by Equation (2.40) to adapt and recover from this perturbation.

$$\boldsymbol{\lambda}' = \eta' \frac{\|\nabla \widetilde{W}\|_1}{\langle (\cdots, \sum_{i \in \mathcal{V}_s} q_i \|\boldsymbol{\xi}_i\|_1, \cdots)^T, \widehat{\nabla}_{\mathrm{sub}} \boldsymbol{\lambda} \rangle} \widehat{\nabla}_{\mathrm{sub}} \boldsymbol{\lambda}, \tag{2.40}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product of two vectors, $(\cdots, \sum_{i \in \mathcal{V}_s} q_i \|\boldsymbol{\xi}_i\|_1, \cdots)^T$ is an $|\mathcal{S}|$-dimensional vector that contains the L1-norm of the density gradient for each resource type $s \in \mathcal{S}$, $\widehat{\nabla}_{\mathrm{sub}} \boldsymbol{\lambda}$ denotes the normalized subgradient of $\boldsymbol{\lambda}$, as defined in Equation (2.33), after the area adjustment, and η' is a weighting parameter set to 0.1 in our experiments. Equation (2.40) essentially redirects $\boldsymbol{\lambda}$ to its current normalized subgradient $\widehat{\nabla}_{\mathrm{sub}} \boldsymbol{\lambda}$ with the scale determined by the gradient norm ratio between wirelength and density. In this way, both the direction and scale of the adjusted $\boldsymbol{\lambda}'$ can adapt the perturbed electrostatic system and help to better heal the placement quality. Besides, in order to smooth the placement convergence, the density multiplier step size is also adjusted by

111

Equation (2.41), where α_{H} is the same parameter as used in Equation (2.35).

$$t' = (\alpha_{\mathrm{H}} - 1) \, \|\boldsymbol{\lambda}'\|_2. \tag{2.41}$$
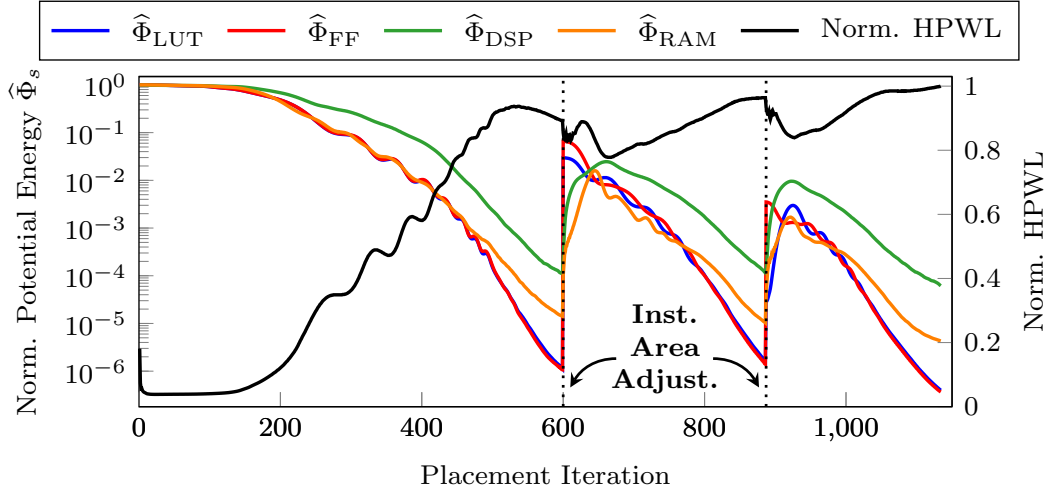
Figure 2.25: The normalized potential energy $\widehat{\boldsymbol{\Phi}}$ and HPWL at different placement iterations on `FPGA-10`.

Figure 2.25 illustrates the impact of cell area adjustment on the placement convergence process. In this example, the adjustment is performed twice at iteration 600 and 887, where the potential energies increase sharply. By using our adaptive density multiplier and step size resetting techniques, the wirelength can be gradually healed and smoothly converges to a nearly overlap-free solution.

2.5.4.2 The Optimized Area Computation

As we discussed in Section 2.5.4.1, for each physical cell i, elfPlace computes three independent areas that are optimized for routability (A_i^{ro}), pin

112

density (A_i^{po}), and packing compatibility (A_i^{pko}), respectively.

2.5.4.2.1 Routability-Optimized Area

In order to compute the routability-optimized areas, elfPlace first performs a RISA/RUDY-based [15, 69] routing congestion estimation. Let u_i^{h} and u_i^{v} denote the resulting horizontal and vertical routing utilizations at cell i, then we compute the routability-optimized area of each physical cell i as follows:

$$A_i^{\mathrm{ro}} = A_i \min \left(\max \left(u_i^{\mathrm{h}}, u_i^{\mathrm{v}} \right)^2, 2 \right), \forall i \in \mathcal{V}^{\mathrm{P}}. \tag{2.42}$$

2.5.4.2.2 Pin Density-Optimized Area

Similarly, elfPlace also estimates pin density by dividing the placement region into bins. Let c^{p} denote the unit-area pin capacity (determined by the FPGA architecture). For each cell i, if we denote its local pin density by u_i^{p} and denote its pin count as $|\mathcal{P}_i|$, then its pin density-optimized area is defined as follows:

$$A_i^{\mathrm{po}} = \frac{|\mathcal{P}_i|}{c^{\mathrm{p}}} \min \left(u_i^{\mathrm{p}}, 1.5 \right), \forall i \in \mathcal{V}^{\mathrm{P}}. \tag{2.43}$$

Different from the routability-optimized area A_i^{ro}, the pin density-optimized area A_i^{po} here is independent to the current cell area A_i. This is because, compared with routing utilization, local pin density is usually very noisy and sensitive to the placement. If A_i^{po} is self-accumulated as in Equation (2.42), it can be excessively over-inflated.

2.5.4.2.3 Packing Compatibility-Optimized Area

One special challenge of flat FPGA placement is that we can barely know the correct LUT and FF areas before the actual downstream packing solution is formed. Recall the CLB architecture described in Section 2.2, if a LUT/FF is incompatible with most of its physical neighbors (e.g., violating the pin count and control set rules), then it tends to occupy a significant portion of a CLB alone. For such a cell, we intuitively should assign it a larger area.

To estimate the cell areas in a feasible packing solution, we first assume the cell movement $(\Delta \boldsymbol{x}, \Delta \boldsymbol{y})$ during the downstream packing/legalization approximately follows Gaussian distribution. That is, we have $\Delta x_i \sim \mathcal{N}(0, \sigma)$ and $\Delta y_i \sim \mathcal{N}(0, \sigma)$, where σ is the assumed standard deviation of the movement. Then, we divide the placement region into square bins with bin length equal to σ, and for each LUT/FF cell i, we conduct the area estimation using the bin window $\mathcal{B}_i$, of size 5×5 bins, that is centered at (x_i, y_i). Let $(\mathcal{B}_i^{\mathrm{xl}}, \mathcal{B}_i^{\mathrm{yl}}, \mathcal{B}_i^{\mathrm{xh}}, \mathcal{B}_i^{\mathrm{yh}})$ denote the bounding box of the estimation window $\mathcal{B}_i$ of cell i, the expectation of any cell j falling into $\mathcal{B}_i$ then can be defined as

$$\mathbf{E}_{j \in \mathcal{B}_i} = P_\sigma(\mathcal{B}_i^{\mathrm{xl}} \leq x_j < \mathcal{B}_i^{\mathrm{xh}}) \, P_\sigma(\mathcal{B}_i^{\mathrm{yl}} \leq y_j < \mathcal{B}_i^{\mathrm{yh}}), \qquad (2.44)$$

where $P_\sigma(a \leq \mu < b)$ represents the total probability of the Gaussian distribution $\mathcal{N}(\mu, \sigma)$ in the range $[a, b)$.

For each LUT cell i, let $\mathcal{V}_i^{\mathrm{P}}$ and $\overline{\mathcal{V}_i^{\mathrm{P}}}$ denote the sets of LUTs that can and cannot be fitted into the same BLE with i (see Section 2.2), respectively,

then we define the packing compatibility-optimized area for LUT i as follows:

$$A_i^{\text{pko}} = \frac{1}{16} \frac{\sum_{j \in \mathcal{V}_i^{\text{P}}} \mathbf{E}_{j \in \mathcal{B}_i}}{\sum_{j \in \mathcal{V}_{\text{LUT}}^{\text{P}}} \mathbf{E}_{j \in \mathcal{B}_i}} + \frac{1}{8} \frac{\sum_{j \in \overline{\mathcal{V}_i^{\text{P}}}} \mathbf{E}_{j \in \mathcal{B}_i}}{\sum_{j \in \mathcal{V}_{\text{LUT}}^{\text{P}}} \mathbf{E}_{j \in \mathcal{B}_i}}, \forall i \in \mathcal{V}_{\text{LUT}}^{\text{P}}. \qquad (2.45)$$

Equation (2.45) essentially is a weighted average of the compatible and incompatible expectations for i within the window $\mathcal{B}_i$. Each $A_i^{\text{pko}}, \forall i \in \mathcal{V}_{\text{LUT}}^{\text{P}}$, is in the range $[1/16, 1/8]$ based on our target architecture.

The estimation for FFs are more subtle due to the complicated control set rules (see Section 2.2). For a FF cell i, let θ_i denote its control set $(\text{CK}, \text{SR}, \text{CE})$ and let Θ_i denote the set of control sets that have the same CK and SR with θ_i. If we use $n_{i,\theta}$ to denote the number of FFs in $\mathcal{B}_i$ with the control set θ, then the area of FF i in the tightest packing solution formed within $\mathcal{B}_i$ can be estimated by Equation (2.46), as given by Equation (2.10) in Section 2.4.3.2.3.

$$A_i^{\text{pko-disc}} = \frac{1}{2n_{i,\theta_i}} \frac{\lceil n_{i,\theta_i}/4 \rceil}{\sum_{\theta \in \Theta_i} \lceil n_{i,\theta}/4 \rceil} \left\lceil \frac{\sum_{\theta \in \Theta_i} \lceil n_{i,\theta}/4 \rceil}{2} \right\rceil, \forall i \in \mathcal{V}_{\text{FF}}^{\text{P}}. \qquad (2.46)$$

It can be seen that Equation (2.46) involves many ceiling operations ($\lceil \cdot \rceil$), which make $A_i^{\text{pko-disc}}$ discontinuous (disc) and very sensitive to the estimation window $\mathcal{B}_i$ and the placement solution.

In elfPlace, the much smoother Equation (2.47), instead of Equation (2.46), is used as the packing compatibility-optimized areas for FF cells.

$$A_i^{\text{pko}} = \frac{1}{2\mathbf{E}_{i,\theta_i}} \frac{\text{sdc}(\mathbf{E}_{i,\theta_i}, 4)}{\sum_{\theta \in \Theta_i} \text{sdc}(\mathbf{E}_{i,\theta}, 4)} \, \text{sdc}\Big(\sum_{\theta \in \Theta_i} \text{sdc}(\mathbf{E}_{i,\theta}, 4), \, 2 \Big), \forall i \in \mathcal{V}_{\text{FF}}^{\text{P}}. \quad (2.47)$$

It has two notable improvements over Equation (2.46): (1) it replaces each FF

115

count $n_{i,\theta}$ in Equation (2.46) with the smoother expectation $\mathbf{E}_{i,\theta}$, which denotes the expected number (nonintegral in general) of FFs in window $\mathcal{B}_i$ with the control set θ; (2) it replaces each division-ceiling operation in Equation (2.46) with the soft division-ceiling function $\mathtt{sdc}(x, d)$ defined in Equation (2.48).

$$\mathtt{sdc}(x, d) = \begin{cases} x + (1 - d)\lfloor x/d \rfloor, & \text{for } x/d - \lfloor x/d \rfloor < 1/d, \\ \lceil x/d \rceil, & \text{otherwise.} \end{cases} \tag{2.48}$$

The plots of $\mathtt{sdc}(x, d)$ function w.r.t. x/d are illustrated in Figure 2.26. It smoothes $\lceil x/d \rceil$ by linearizing the beginning $1/d$ of each sharp step. As d approaches to ∞, $\mathtt{sdc}(x, d)$ behaves more like $\lceil x/d \rceil$.
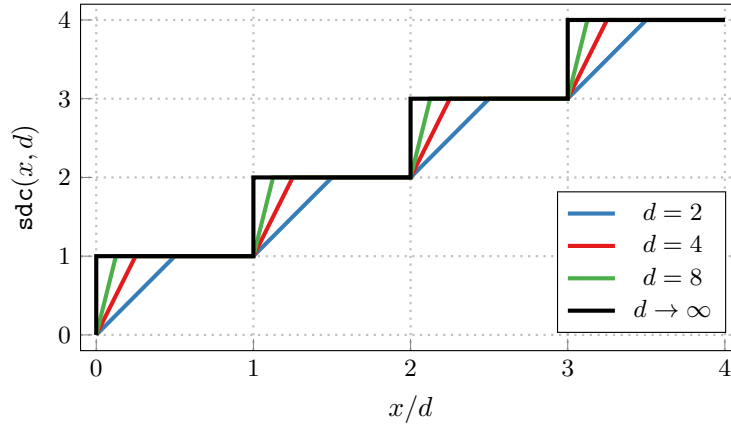
Figure 2.26: The plots of the soft division-ceiling function $\mathtt{sdc}(x, d)$ w.r.t. x/d.

2.5.5 Experimental Results

We implement elfPlace in C++ and perform experiments on a Linux machine running with Intel Core i9-7900 CPUs (3.30 GHz and 10 cores) and 128 GB RAM. Careful parallelization is applied throughout the whole frame-

116

work with the support of OpenMP 4.0 [2]. The ISPD 2016 FPGA placement contest benchmark suite [79] released by Xilinx is adopted to demonstrate the effectiveness and efficiency of elfPlace. Routed wirelength reported by Xilinx Vivado v2015.4 is used to evaluate the placement quality. The characteristics of the benchmarks are listed in Table 2.14.

Table 2.14: ISPD 2016 Contest Benchmarks Statistics

Design	#LUT	#FF	#RAM	#DSP	#Ctrl Set
FPGA-01	50K	55K	0	0	12
FPGA-02	100K	66K	100	100	121
FPGA-03	250K	170K	600	500	1281
FPGA-04	250K	172K	600	500	1281
FPGA-05	250K	174K	600	500	1281
FPGA-06	350K	352K	1000	600	2541
FPGA-07	350K	355K	1000	600	2541
FPGA-08	500K	216K	600	500	1281
FPGA-09	500K	366K	1000	600	2541
FPGA-10	350K	600K	1000	600	2541
FPGA-11	480K	363K	1000	400	2091
FPGA-12	500K	602K	600	500	1281
Resources	538K	1075K	1728	768	-

2.5.5.1 Comparison with State-of-the-Art Placers

We compare elfPlace with four state-of-the-art analytical FPGA placers, namely, UTPlaceF [40], RippleFPGA [12], GPlace3.0 [3], and UTPlaceF-DL [44]. The executables are obtained from their authors and executed on our machine. Since only UTPlaceF-DL and elfPlace support multi-threading, UTPlaceF, RippleFPGA, and GPlace3.0 are single-thread executed, while UTPlaceF-DL and elfPlace are executed with both a single thread and 10 threads.

Table 2.15 shows the comparison results. Metrics "WL" and "RT" represent the routed wirelength in thousands and runtime in seconds, while "WLR" and "RTR" represent the routed wirelength and runtime ratios normalized to the 10-threaded elfPlace. It can be seen that elfPlace achieves the best routed wirelength on eleven out of twelve designs and outperforms UT-PlaceF, RippleFPGA, GPlace3.0, and UTPlaceF-DL by, in average, 13.6%, 11.3%, 8.9%, and 7.1%, respectively. It is worthwhile to note that these wirelength improvements are fairly consistent from small designs to large ones. With only a single thread, elfPlace demonstrates similar runtime compared with UTPlaceF, GPlace3.0, and UTPlaceF-DL, while it runs more than $3\times$ slower than the fastest RippleFPGA. By exploiting 10 threads, elfPlace achieves $3.51\times$ speedup and shows similar runtime with RippleFPGA. Among all twelve designs, the 10-threaded elfPlace produces the fastest runtime on the seven largest designs, which evidences its good scalability.

Table 2.15: Routed Wirelength (WL in 10^3) and Placement Runtime (RT in seconds) Comparison with Other State-of-the-Art Placers

Design	UTPlaceF [40]				RippleFPGA [12]				GPlace3.0 [3]				UTPlaceF-DL [44]						elfPlace					
			1-thread				1-thread				1-thread				1-thread		10-thread				1-thread		10-thread	
	WL	WLR	RT	RTR	WL	WLR	RT	RTR	WL	WLR	RT	RTR	WL	WLR	RT	RTR	RT	RTR	WL	WLR	RT	RTR	RT	RTR
FPGA-01	357	1.128	144	2.25	353	1.115	**25**	**0.39**	356	1.125	70	1.09	340	1.076	154	2.41	50	0.78	**316**	**1.000**	226	3.53	64	1.00
FPGA-02	642	1.106	244	2.02	645	1.112	**47**	**0.39**	644	1.110	133	1.10	653	1.125	273	2.26	90	0.74	**580**	**1.000**	429	3.55	121	1.00
FPGA-03	3215	1.123	672	3.26	3262	1.140	**168**	**0.82**	3101	1.084	502	2.44	3139	1.097	700	3.40	248	1.20	**2862**	**1.000**	732	3.55	206	1.00
FPGA-04	5410	1.117	667	3.06	5510	1.137	**188**	**0.86**	5403	1.115	537	2.46	5331	1.101	802	3.68	278	1.28	**4844**	**1.000**	781	3.58	218	1.00
FPGA-05	9660	1.048	841	3.61	9969	1.082	**229**	**0.98**	10507	1.140	639	2.74	10045	1.090	889	3.82	302	1.30	**9215**	**1.000**	845	3.63	233	1.00
FPGA-06	6488	1.133	1387	3.91	6180	1.079	356	1.00	5820	1.016	1147	3.23	5801	1.013	1128	3.18	424	1.19	**5727**	**1.000**	1124	3.17	**355**	**1.00**
FPGA-07	10105	1.155	1438	4.27	9640	1.102	394	1.17	9509	1.087	1428	4.24	9356	1.069	1277	3.79	479	1.42	**8749**	**1.000**	1092	3.24	**337**	**1.00**
FPGA-08	7879	1.028	1419	4.62	8157	1.065	354	1.15	8126	1.061	1575	5.13	8298	1.083	1478	4.81	497	1.62	**7661**	**1.000**	1214	3.95	**307**	**1.00**
FPGA-09	12369	1.161	2043	5.20	12305	1.155	485	1.23	11711	1.099	1938	4.93	11633	1.092	1724	4.39	622	1.58	**10657**	**1.000**	1359	3.46	**393**	**1.00**
FPGA-10	8795	1.452	2526	5.54	7140	1.178	547	1.20	6836	1.128	1797	3.94	6317	1.043	1467	3.22	583	1.28	**6058**	**1.000**	1445	3.17	**456**	**1.00**
FPGA-11	**10196**	**0.978**	1719	4.70	11023	1.058	447	1.22	10260	0.985	1786	4.88	10476	1.005	1687	4.61	640	1.75	10421	1.000	1367	3.73	**366**	**1.00**
FPGA-12	7755	1.197	2455	5.18	7363	1.136	549	1.16	7224	1.115	2296	4.84	6835	1.055	1926	4.06	771	1.63	**6480**	**1.000**	1714	3.62	**474**	**1.00**
Norm.	-	1.136	-	3.97	-	1.113	-	**0.96**	-	1.089	-	3.42	-	1.071	-	3.63	-	1.31	-	**1.000**	-	3.51	-	1.00

119

2.5.5.2 Individual Technique Validation

Table 2.16 further validates the effectiveness of each proposed technique. The column "w/ MM in Equation (2.24)" shows the results of using the multiplier method (MM) in Equation (2.24), which is adopted by ePlace, instead of our proposed augmented Lagrangian method (ALM) in Equation (2.22). To make a fair comparison, we set the step size of the MM in a way that the MM and our ALM can converge within about the same amount of time. With this setup, our proposed ALM-based formulation can produce an average of 1.2% better routed wirelength compared with the MM-based formulation. The column "w/o Precond." shows the results without the preconditioning in Equation (2.29) and it can barely converge due to the wide spectrum of cell sizes and net degrees in FPGA designs. The column "w/ ePlace Precond." further gives the results of replacing our preconditioner in Equation (2.29) with the one proposed in ePlace [54]. Although the ePlace's preconditioning technique can achieve similar placement quality and efficiency, it fails to converge on two benchmarks in our experiments. Finally, the column "w/ A_i^{pko} in UTPlaceF-DL" presents the results of using the packing compatibility-optimized area proposed in UTPlaceF-DL instead of our Gaussian and `sdc`-smoothed Equation (2.45) and Equation (2.47). With our smoothing techniques, elfPlace can converge 15% faster while maintaining essentially the same solution quality.

Table 2.16: Normalized Routed Wirelength and Placement Runtime Comparison for Individual Technique Validation

Design	w/ MM in Equation (2.24)		w/o Precond.		w/ ePlace Precond.		w/ A_i^{pko} in UTPlaceF-DL		elfPlace	
	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR
FPGA-01	1.021	1.02	1.009	1.01	1.006	1.02	1.004	1.12	1.000	1.00
FPGA-02	1.010	0.97	*	-	*	-	1.001	1.16	1.000	1.00
FPGA-03	1.009	1.03	*	-	0.995	1.03	0.998	1.23	1.000	1.00
FPGA-04	1.046	0.94	*	-	*	-	1.002	1.16	1.000	1.00
FPGA-05	1.026	1.05	*	-	1.002	1.03	0.996	1.07	1.000	1.00
FPGA-06	1.008	1.01	*	-	1.030	0.98	1.004	1.25	1.000	1.00
FPGA-07	1.001	1.04	*	-	0.988	1.04	0.986	1.20	1.000	1.00
FPGA-08	0.991	1.01	*	-	0.996	1.01	0.999	1.14	1.000	1.00
FPGA-09	1.003	1.01	*	-	0.996	1.03	1.002	1.12	1.000	1.00
FPGA-10	1.006	1.01	*	-	1.000	1.01	0.996	1.03	1.000	1.00
FPGA-11	1.011	0.98	*	-	1.002	1.05	1.009	1.15	1.000	1.00
FPGA-12	1.017	1.02	*	-	0.995	1.04	1.003	1.14	1.000	1.00
Norm.	1.012	1.01	1.009	1.01	1.001	1.03	1.000	1.15	1.000	1.00

* Placement fails to converge.

2.5.5.3 Runtime Breakdown

Figure 2.27 shows the runtime breakdown of the 10-threaded elfPlace based on FPGA-12. The most time-consuming part is to compute the wirelength gradient $\nabla \widetilde{W}$, which takes 34.4% of the total runtime. The density gradient computation is relatively efficient and it consumes 7.5% of the total runtime on constructing the density maps $\boldsymbol{\rho}$ and 1.4% of the total runtime on computing the electric potential $\boldsymbol{\psi}$ and electric field $\boldsymbol{\xi}$ by Equation (2.18). The parameter updating, which involves the computation of wirelength, overflow, potential energy $\boldsymbol{\Phi}$, etc., takes 7.3% of the total runtime. The packing, legalization, and detailed placement algorithms adopted from UTPlaceF-DL consumes total 37.5% of the runtime. While the remaining 11.9% of the runtime is

121

spent on parsing, placement initialization, and the rest of runtime-insignificant tasks.
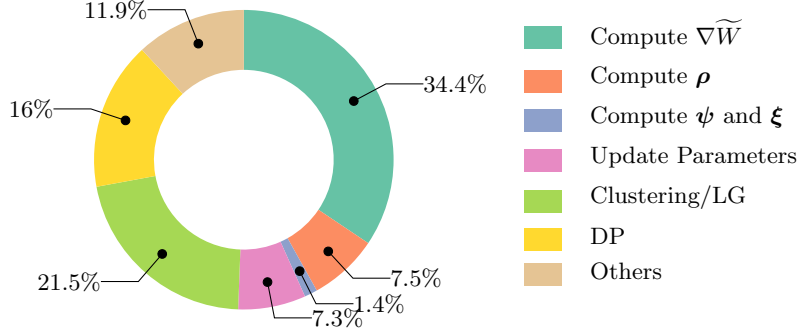
Figure 2.27: The runtime breakdown of the 10-threaded elfPlace based on `FPGA-12`.

2.5.6   Summary

In this section, we have presented elfPlace, a general, flat, nonlinear placement algorithm for large-scale heterogeneous FPGAs. elfPlace resolves the traditional FPGA heterogeneity issue by casting the density constraints of heterogeneous resource types to separate but unified electrostatic systems. An augmented Lagrangian formulation together with a preconditioning technique and a normalized subgradient-based multiplier updating scheme are proposed to achieve satisfiable solution quality with fast and robust numerical convergence. Besides pure-wirelength minimization, elfPlace is also capable of optimizing routability, pin density, and downstream packing compatibility based on a unified cell area adjustment scheme. Our experiments show that elfPlace significantly outperforms four state-of-the-art placers in routed wirelength with competitive runtime.

Chapter 3

Clock-Aware FPGA Placement Algorithms

3.1   Introduction

Placement is one of the most important and time-consuming optimization steps in FPGA implementation flow and it can significantly affect the efficiency and quality of mapped designs on FPGA devices. The placement problem has attracted great attention from both academia and industry and has been intensively studied during the last two decades. However, with the increasing complexity and scale of modern FPGA devices, today's FPGA architecture imposes various intricate constraints, which have not yet been paid enough attention to, during the placement stage. These constraints have great impacts on placement quality in terms of traditional design metrics such as wirelength, routability, power, and timing. Therefore, it is imperative to have

---

This chapter is based on the following publications.

1. Wuxi Li, Yibo Lin, Meng Li, Shounak Dhar, and David Z. Pan. "UTPlaceF 2.0: A high-performance clock-aware FPGA placement engine." ACM Transactions on Design Automation of Electronic Systems 23.4 (2018): 42.

2. Wuxi Li, Mehrdad E. Dehkordi, Stephen Yang, and David Z. Pan. "Simultaneous Placement and Clock Tree Construction for Modern FPGAs." In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 132-141. ACM, 2019.

I am the main contributor in charge of problem formulation, algorithm development, and experimental validations.

new placement techniques to honor these complicated architecture constraints.

Clock rules, among various FPGA architecture constraints, are of great importance, not only because of their significant impact on timing closure and power dissipation but their imposed layout constraints that affect how efficiently other logic can be mapped. With the consideration of clock rules, placement turns out to be much more difficult for FPGAs even to find a feasible solution. For modern FPGAs, clock network planning must be involved during or even before placement stage due to their pre-manufactured clock networks, which cannot be adjusted to different designs, as well as their limited clock routing resources. The interdependency between clock sink placement and clock network planning makes the clock-aware placement for FPGAs a challenging chicken-and-egg problem.

While many existing works have proposed fairly mature placement techniques for FPGAs to optimize conventional design metrics such as wirelength, routability, power, and timing [3, 7, 12, 14, 25, 40, 44, 50, 58, 66–68, 75, 77], there has been limited research effort on placement with the awareness of clock rules. The closest related previous work is [35], which proposes a cost function for cell swapping that penalizes high-clock-usage placement and integrates it into a conventional simulated annealing-based placement engine to produce clock-legal solutions. However, their approach suffers from slow annealing process and its quality heavily depends on the cost function tuning. Moreover, their approach can easily get stuck in some local swappings and hence unable to resolve global clock congestions.

124

In this chapter, we propose two clock-aware placement frameworks:

- **UTPlaceF 2.0**, a high-performance clock-aware placement framework that resolves clock routing congestions by an iterative minimum-cost flow-based cell assignment technique.

- **UTPlaceF 2.X**, a generalization of UTPlaceF 2.0 that explore a much larger clock routing solution space based on the branch-and-bound idea.

In the rest of this chapter, Section 3.2 introduces the target FPGA clocking architecture of the two proposed frameworks, Section 3.3 describes UTPlaceF 2.0, and Section 3.4 details UTPlaceF 2.X.

3.2 Target FPGA Clocking Architecture

Our target FPGA device is Xilinx *UltraScale VU095*, which was also adopted in both ISPD 2016 and ISPD 2017 FPGA placement contests [79,80]. Its clocking architecture is illustrated in Figure 3.1(a) – (c). The global clocking architecture, as shown in Figure 3.1(a), is physically a two-level network composed of a clock routing layer (R-layer) and a clock distribution layer (D-layer). To simplify the notations, in the rest of this chapter, we will denote the horizontal/vertical routing layer as HR/VR, and the horizontal/vertical distribution layer as HD/VD. In the target architecture, all of HR, VR, HD, and VD layers have 24 tracks running through each of the 5×8 clock regions. Figure 3.1(b) gives a closer look within a single clock region. The connection between HR and VR layers are bidirectional, while there are only unidirectional connections from HR/VR to VD and from VD to HD. Given this architecture, a clock tree needs to follow the pattern shown in Figure 3.1(c). Specifically, it consists of two parts: (1) a D-layer vertical trunk tree connecting all the clock regions containing clock loads; (2) an R-layer route connecting the clock source and the D-layer trunk tree. For the leaf clock networks, a clock region is further divided into multiple half-column regions of two-site width and half-clock-region height as shown in Figure 3.1(d). Figure 3.1(e) illustrates a detailed view of three adjacent half-column regions. Different half-column regions might contain distinct logic resources but they have similar leaf clock networks to directly drive clock sinks within them. In the target architecture, the leaf clock network in each half-column region has 12 routing tracks.

126
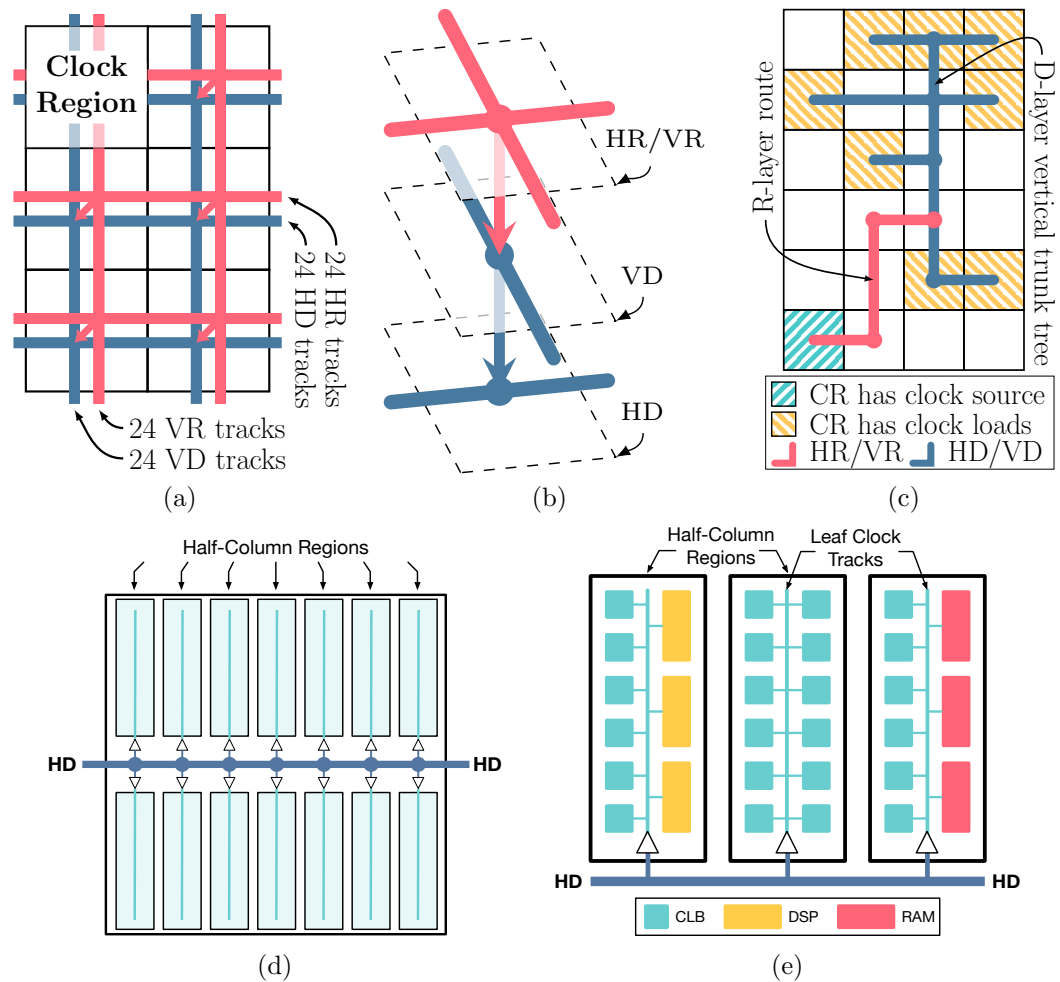
Figure 3.1: Illustration of the target clocking architecture. (a) A global view of 2×3 clock regions with R-layer (red) and D-layer (blue). (b) A detailed view of HR/VR/HD/VD within a single clock region. (c) The required routing pattern of a clock net. (d) A global view of half-column regions within a clock region. (e) A detailed view of three adjacent half-column regions with different logic resources.

3.3   UTPlaceF 2.0: A High-Performance Clock-Aware FPGA Placer

Modern FPGA devices contain complex clock architectures on top of configurable logic. The physical structure of clock networks in an FPGA is premanufactured and cannot be adjusted to different applications. Furthermore, clock routing resources are typically limited for high-utilization designs. Consequently, clock architectures impose extra clock constraints and further complicate physical implementation tasks such as placement. Traditional placement techniques only optimize conventional design metrics such as wirelength, routability, power, and timing without clock legality consideration. It is imperative to have new techniques to honor clock constraints during placement for FPGAs.

To address the clock legalization challenges in FPGA placement for large-scale state-of-the-art commercial FPGAs, we proposed a high-performance clock-aware placement engine, UTPlaceF 2.0, which simultaneously optimizes the conventional placement objectives of wirelength and routability and honors complicated global and detailed clock constraints. The major contributions of this work are highlighted as follows.

- An iterative minimum-cost-flow-based cell assignment technique producing clock-legal solutions with minimized perturbation to placements optimized for other design metrics (e.g., wirelength and routability).

- A clock-aware packing technique with probabilistic clock distribution

estimation for producing clock- and placement-friendly netlists.

- Won the first-place award of ISPD 2017 clock-aware FPGA placement contest [80] on industry-strength benchmarks released by Xilinx [29] and outperforms the second- and third-place winners by 3.3% and 9.7%, respectively, in routed wirelength with competitive runtime.

The rest of this section is organized as follows. Section 3.3.1 reviews the preliminaries. Section 3.3.2 presents our UTPlaceF 2.0 algorithms. Section 3.3.3 shows the experimental results, followed by the summary in Section 3.3.4.

3.3.1 Preliminaries and Overview

In this section, we will briefly introduce the clock constraints and the problem formulation for clock-aware placement.

3.3.1.1 Clock Constraints for Placement

Restricted by the clocking architecture, two major constraints, namely clock region constraint and half-column region constraint, are imposed in the placement stage.

The clock region constraint is introduced by the limited clock routing/distribution tracks and it requires the global clock demand in each clock region must be equal to or less than 24 in our target FPGA architecture. Exact global clock demand in each clock region requires detailed clock routing, which
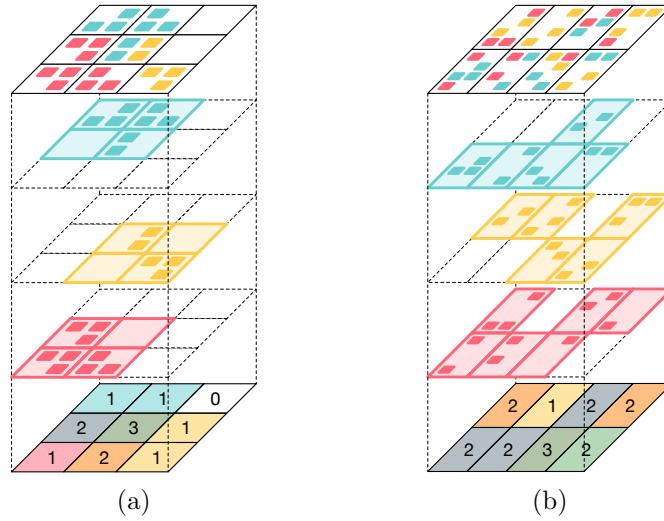
Figure 3.2: Illustration of the clock demand calculation for clock regions and half-column regions. Different colors represent different clocks. (a) Global clock demand calculation for clock regions. (b) Clock demand calculation for half-column regions within a clock region.

is often computationally expensive in practice. Therefore, we approximate it using the bounding box model for the sake of efficiency. More specifically, for a clock region, its global clock demand is defined as the total number of clock nets that have their bounding boxes intersected with it. Figure 3.2(a) illustrates how global clock demands are calculated for a simple placement with three clock nets. The top layer shows the sink distribution, the three middle layers represent the spanning clock regions of each clock and the layer in the bottom gives the final global clock demand in each clock region.

Similarly, imposed by the limited leaf clock tracks, the half-column region constraint requires that each half-column region can only contain at most 12 clocks. Figure 3.2(b) illustrates the clock demand calculation for

half-column regions within a clock region. Different from clock regions, the clock demand in a half-column region is only the number of clocks inside it, regardless of the clock net bounding boxes.

3.3.1.2 Problem Formulation

In modern FPGA placement, the optimization usually includes multiple objectives, such as wirelength, which is often measured by half-perimeter wire-length (HPWL), and routability. Wirelength is usually regarded as the major objective since it is a good first-order approximation of many other metrics, e.g., power and timing. However, pure wirelength-driven placement can result in routing quality degradation and even unroutable solutions. Therefore, in UTPlaceF 2.0, wirelength and routability are optimized simultaneously.

To produce legal placement solutions, apart from clock constraints, the packing rules described in Section 2.2 also need to be satisfied when multiple LUTs and FFs are placed into the same CLB site.

With all constraints and objectives defined, we now define our clock-aware placement problem as follows.

Problem 1 (Clock-Aware Placement) *Given a netlist of LUTs, FFs, DSPs, RAMs, and I/Os, produce a legal placement solution with minimized routed wirelength, meanwhile packing rules, clock region constraint, and half-column region constraint are satisfied.*

3.3.2   UTPlaceF 2.0 Algorithms

3.3.2.1   UTPlaceF 2.0 Overview
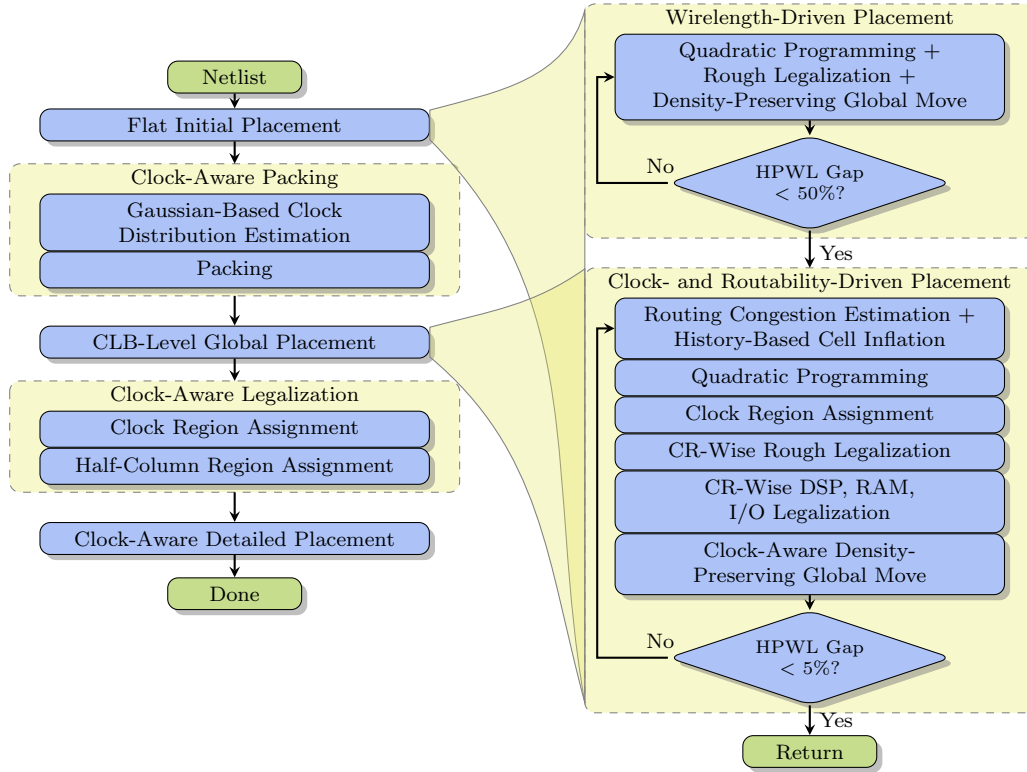
Figure 3.3: The overall flow of UTPlaceF 2.0.

The overall flow of UTPlaceF 2.0 is shown in Figure 3.3. UTPlaceF 2.0 is a natural extension of UTPlaceF (Section 2.3) and consists of five major steps: (1) flat initial placement (FIP); (2) packing; (3) CLB-level global placement; (4) legalization; (5) detailed placement. On top of conventional wirelength and routability optimizations performed in UTPlaceF, clock constraints are explicitly considered throughout the UTPlaceF 2.0 framework.

The FIP is responsible for producing the physical location and rout-

ing congestion information of each cell to better guide decision making in the packing stage. It consists of two phases: (1) pure wirelength-driven phase; (2) clock- and routability-driven phase. In each iteration of the first phase, a quadratic analytical placement utilizing hybrid [76] and bound-to-bound [70] net models is solved to minimize wirelength, followed by the rough legalization technique proposed in [48] for cell overlap removal. After that, a sequence of global moves is performed to further refine roughly-legalized placement while preserving cell density. In the second phase, besides the conventional routability optimization, an additional clock region assignment (see Section 3.3.2.2) step is called after the quadratic placement. It produces a cell-to-clock-region assignment solution that satisfies the clock region constraint. Then, the rough legalization and DSP/RAM block legalization are only performed within each clock region to honor the assignment result. UTPlaceF 2.0 stops FIP once the wirelength converges.

In the packing stage, a probability-based estimation is performed to predict the clock distribution in the final placement solution. Then, by incorporating the estimated clock information into the original packing algorithm in UTPlaceF, the packing in UTPlaceF 2.0 is enhanced to be clock-aware (see Section 3.3.2.3).

The CLB-level global placement is performed immediately after our packing to further optimize the placement for the post-packing netlist. It shares the same framework with the second phase of FIP and uses the final solution of FIP as the starting point to speed up the placement convergence.

In the legalization stage, while minimizing the conventional objective of pin movement, the clock region constraint and half-column region constraint are rigorously honored by our clock region assignment technique and half-column region assignment technique (see Section 3.3.2.4), respectively.

Finally, detailed placement techniques in UTPlaceF are extended to be clock-aware and maintain the clock legality throughout the wirelength and routability optimization process before the final solution is produced.

3.3.2.2   Clock Region Assignment

Clock region assignment is a key step in UTPlaceF 2.0 to satisfy the clock region constraint. It is intensively called throughout major steps, such as FIP, CLB-level global placement, and legalization, in UTPlaceF 2.0. Therefore, it has a significant impact on both solution quality and runtime. Here we formally define the clock region assignment problem as follows.

Problem 2 (Clock Region Assignment) *Given a roughly-legalized placement and logic resource capacity of each clock region, assign cells to clock regions to minimize total pin movement without logic resource or global clock overflow.*

Given the notations defined in Table 3.1, Problem 2 can be written as a binary minimization problem with linear and boolean logical constraints as shown in Formulation (3.1).

Table 3.1: Notations Used in Clock Region Assignment

$\mathcal{V}$	The set of cells.
$\mathcal{V}^s$	The set of cells of resource type $s \in \{\text{CLB, DSP, RAM}\}$.
P_i	The number of pins in cell i.
A_i^s	The cell i's demand for resource type $s \in \{\text{CLB, DSP, RAM}\}$.
$\mathcal{R}$	The set of clock regions.
C_j^s	The clock region j's capacity for resource type $s \in \{\text{CLB, DSP, RAM}\}$.
$D_{i,j}$	The physical distance between cell i and clock region j.
$\mathcal{E}$	The set of clock nets.
$Z_{i,e}$	A binary value represents whether cell i is in clock net e.
$\mathcal{L}_j^+$	The set of clock regions that are left to or in the same column of clock region j.
$\mathcal{R}_j^+$	The set of clock regions that are right to or in the same column of clock region j.
$\mathcal{B}_j^+$	The set of clock regions that are below or in the same row of clock region j.
$\mathcal{T}_j^+$	The set of clock regions that are above or in the same row of clock region j.

$$\min_{\boldsymbol{x}} \quad \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{R}} P_i\, D_{i,j}\, x_{i,j}, \tag{3.1a}$$

$$\text{s.t.} \quad x_{i,j} \in \{0,1\}, \forall i \in \mathcal{V}, \forall j \in \mathcal{R}, \tag{3.1b}$$

$$\sum_{j \in \mathcal{R}} x_{i,j} = 1, \forall i \in \mathcal{V}, \tag{3.1c}$$

$$\sum_{i \in \mathcal{V}} A_i^s\, x_{i,j} \le C_j^s, \forall j \in \mathcal{R}, \forall s \in \{\text{CLB, DSP, RAM}\} \tag{3.1d}$$

$$\sum_{e \in \mathcal{E}} \Big[\Big(\bigvee_{q \in \mathcal{L}_j^+} Z_{i,e}\, x_{i,q}\Big) \wedge \Big(\bigvee_{q \in \mathcal{R}_j^+} Z_{i,e}\, x_{i,q}\Big) \wedge$$
$$\Big(\bigvee_{q \in \mathcal{B}_j^+} Z_{i,e}\, x_{i,q}\Big) \wedge \Big(\bigvee_{q \in \mathcal{T}_j^+} Z_{i,e}\, x_{i,q}\Big)\Big] \le 24, \forall j \in \mathcal{R}. \tag{3.1e}$$

Formulation (3.1) optimizes over binary variables $x_{i,j}$ to minimize the objective (3.1a) of total pin movement. If cell $i \in \mathcal{V}$ is assigned to clock region $j \in \mathcal{R}$, then $x_{i,j} = 1$, otherwise $x_{i,j} = 0$. The constraint (3.1c) ensures that
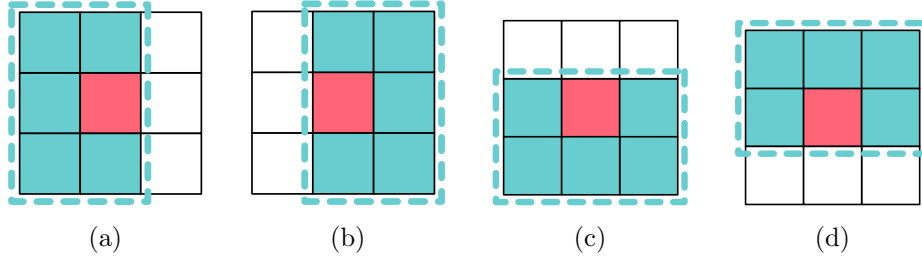
Figure 3.4: Illustration of the sets of clock regions (dashed regions) (a) $\mathcal{L}_j^+$, (b) $\mathcal{R}_j^+$, (c) $\mathcal{B}_j^+$, and (d) $\mathcal{T}_j^+$ of clock region j (red) defined in Table 3.1.

each cell is assigned to exactly one clock region. The constraint (3.1d) guarantees that the demands are no greater than the capacities for all logic resource types (e.g., CLB, DSP, and RAM) in each clock region. The boolean logical constraint (3.1e) ensures that the clock region constraint is satisfied. The sets of clock regions $\mathcal{L}_j^+$, $\mathcal{R}_j^+$, $\mathcal{B}_j^+$, and $\mathcal{T}_j^+$ of any clock region j are illustrated in Figure 3.4. Intuitively, for a given clock region j, if and only if a clock net has cells located in all of $\mathcal{L}_j^+$, $\mathcal{R}_j^+$, $\mathcal{B}_j^+$, and $\mathcal{T}_j^+$, its bounding box would intersect with j and occupy global clock resource in it. The constraint (3.1e) sums up all such clock nets for each clock region and restricts the clock usage no more than the clock capacity, which is 24 in our target FPGA architecture.

3.3.2.2.1 The Relaxation Algorithm

With a proper modeling, the boolean logical constraint (3.1e) can be transformed into a set of linear constraints and then Formulation (3.1) can be optimally solved utilizing integer linear programming techniques. However, integer linear programming is computationally expensive and suffers from unafford-

able runtime for our application. Therefore, here we relax Formulation (3.1) to an easier problem that can be efficiently solved, as shown in Formulation (3.2).

$$\min_{\boldsymbol{x}} \quad \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{R}} (P_i\, D_{i,j} + \lambda_{i,j})\, x_{i,j}, \tag{3.2a}$$

$$\text{s.t.} \quad x_{i,j} \in \{0,1\}, \forall i \in \mathcal{V}, \forall j \in \mathcal{R}, \tag{3.2b}$$

$$\sum_{j \in \mathcal{R}} x_{i,j} = 1, \forall i \in \mathcal{V}, \tag{3.2c}$$

$$\sum_{i \in \mathcal{V}} A_i\, x_{i,j} \leq C_j, \forall j \in \mathcal{R}. \tag{3.2d}$$

Compared with the original Formulation (3.1), Formulation (3.2) has two major differences: (1) instead of considering the capacity constraint (3.1d) simultaneously for all logic resource types, here we only consider the capacity constraint (3.2d) for one resource type at a time (i.e., we consider different resource types separately); (2) we remove the boolean logical constraint (3.1e) and add a penalty multiplier $\lambda_{i,j}$ for each potential cell $i \in \mathcal{V}$ to clock region $j \in \mathcal{R}$ assignment $x_{i,j}$ to the objective (3.2a).

The main idea of our relaxation can be explained as follows. We first ignore the clock region constraint (3.1e) and only honor each logic resource constraint separately. Each time after solving Formulation (3.2), we properly penalize assignments causing clock overflows by updating the corresponding penalty multipliers $\lambda_{i,j}$ in the objective (3.2a). By repeating the process of solving and updating Formulation (3.2), the assignment will progressively converge to a clock-feasible solution.

Algorithm 9: Clock Region Assignment with Relaxation

    Input : A roughly-legalized placement.

    Output: A movement-minimized assignment without logic
            resource or clock overflows.

1 Divide cells $\mathcal{V}$ into three groups, $\mathcal{V}^{\mathrm{CLB}}$, $\mathcal{V}^{\mathrm{DSP}}$, and $\mathcal{V}^{\mathrm{RAM}}$, based on
   their logic resource types.;

2 $\lambda_{i,j} \leftarrow 0, \forall i \in \mathcal{V}, \forall j \in R$;

3 **while** *clock overflow exists* **do**

4   **foreach** $\mathcal{U} \in \{\mathcal{V}^{CLB}, \mathcal{V}^{DSP}, \mathcal{V}^{RAM}\}$ **do**

5   Solve Formulation (3.2) for $\mathcal{U}$ (See Section 3.3.2.2.2);

6   **end**

7   Update penalty multiplier $\boldsymbol{\lambda}$ (See Section 3.3.2.2.3);

8 **end**

The proposed relaxation-based clock region assignment algorithm is summarized in Algorithm 9. Cells are first divided into three groups based on their resource types, including CLB, DSP, and RAM, at line 1. Then all penalty multipliers $\lambda_{i,j}, \forall i \in \mathcal{V}, \forall j \in \mathcal{R}$ are initialized to zero at line 2, which degenerates the Formulation (3.2) into a pure logic-resource-constrained pin-movement minimization problem without clock legality consideration. Within the loop from line 3 to line 8, we first solve Formulation (3.2) for each cell group under current penalty multiplier settings, and then at line 7, penalty multipliers $\lambda_{i,j}$ are updated according to the assignment solutions produced from line 4 to line 6. The penalty multipliers are incrementally updated to penalize assignments causing clock overflows, and new assignments produced by the Formulation (3.2) with these updated penalty multipliers will be progressively closer to clock-legal solutions after each iteration. The process of solving and updating Formulation (3.2) from line 3 to line 8 is repeated until

a clock-overflow-free assignment is reached.

3.3.2.2.2   Minimum-Cost Flow Transformation

One notable merit of our relaxation in Formulation (3.2) is that it can be transformed into a minimum-cost-flow problem, which can be solved by many mature and efficient algorithms [4].
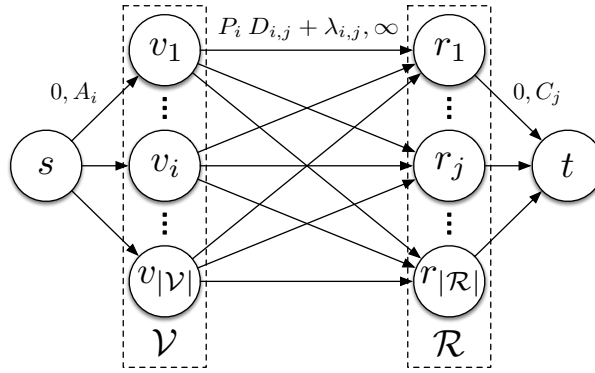
Figure 3.5: The minimum-cost flow representation of Formulation (3.2). Pair of numbers (e.g., $P_i\, D_{i,j} + \lambda_{i,j}, \infty$) on each edge represents cost and capacity, respectively, and ∞ means unlimited capacity.

Figure 3.5 illustrates the minimum-cost flow representation of Formulation (3.2). If all cells in $\mathcal{V}$ have unit resource demand (i.e., $A_i = 1, \forall i \in \mathcal{V}$), optimally solving Formulation (3.2) is equivalent to computing the minimum-cost flow of amount $\sum_{i \in \mathcal{V}} A_i$ on the graph. For cases with non-unit resource demands, however, the assignment solutions corresponding to their minimum-cost flow results may contain cells that are split into multiple clock regions, which require extra post-processing steps and slight relaxation on constraint (3.2d) to guarantee solution feasibility. In UTPlaceF 2.0, we always

139

apply unit cell resource demand ($A_i = 1$) to ensure the solutions produced by our minimum-cost flows transformation are feasible.

3.3.2.2.3 Penalty Multiplier Updating

Algorithm 9 is a generalized relaxation framework for solving the clock region assignment problem. One key step in this framework is to update the penalty multiplier $\boldsymbol{\lambda}$ after each assignment iteration. Different updating strategies can converge to different feasible solutions. In this section, we will only discuss the updating method that is applied in UTPlaceF 2.0. Its effectiveness is demonstrated by our experiments.

The guiding idea of our $\boldsymbol{\lambda}$ updating method is that we hope clock overflows can be mitigated by forbidding some clocks from occupying the overflowed clock regions. We observe that to block a clock net from taking clock resources in a given clock region, all the cells in this clock net must be strictly restricted in the region below, above, left to, or right to the clock region, as shown in Figure 3.6. Otherwise, the clock bounding box must intersect with the clock region. Thus, intuitively, clock congestion can be resolved by iteratively pushing clock nets to these four escaping regions corresponding to overflowed clock regions.

Inspired by this observation, we propose our penalty multiplier updating method summarized in Algorithm 10. We first choose the most overflowed clock region as our target clock region to resolve at line 1 and locate the four escaping regions illustrated in Figure 3.6 for the target clock region from line

Algorithm 10: Penalty Multiplier Updating for Clock Region Constraint

> **Input** : Penalty multiplier $\boldsymbol{\lambda}$. The maximum clock overflow descent step I_{max} for each clock region in each iteration.
>
> **Output:** An updated penalty multiplier $\boldsymbol{\lambda}$ that can produce less clock overflow.

1 Find the clock region r with the largest clock overflow O_{max};
2 $B_r \leftarrow$ the region below r;
3 $T_r \leftarrow$ the region above r;
4 $L_r \leftarrow$ the region left to r;
5 $R_r \leftarrow$ the region right to r;
6 $l \leftarrow \varnothing$;
7 **foreach** *clock $e \in \mathcal{E}$ occupying clock resource of r* **do**
8   **foreach** *region $b \in \{B_r, T_r, L_r, R_r\}$* **do**
9          $l \leftarrow l \cup \{\texttt{Cost}(e, b)\}$ (See Equation (3.3));
10 Sort candidates in l by ascending order of their cost;
11 $I \leftarrow \min(O_{max}, I_{max})$;
12 $i \leftarrow 0$;
13 $chosen[e] \leftarrow false, \forall e \in \mathcal{E}$;
14 **foreach** $\texttt{Cost}\ (e, b) \in$ *sorted l* **do**
15   **if** *chosen[e]* ***or*** $\neg\texttt{IsFeasible}(e, b, \boldsymbol{\lambda})$ **then continue**;
16   **foreach** *cell i in clock e* **do**
17   **for** *each clock region j that is not in region b* **do**
18              $\lambda_{i,j} \leftarrow \infty$;
19      $chosen[e] \leftarrow true$;
20      $i \leftarrow i + 1$;
21   **if** $i \geq I$ **then return**;
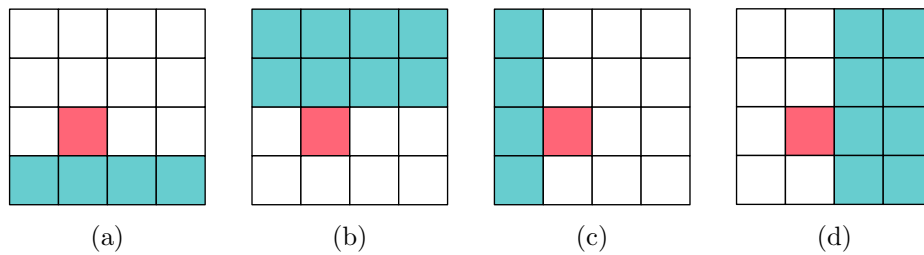
Figure 3.6: The four escaping regions (green) that are defined as the regions (a) below, (b) above, (c) left to, and (d) right to a given clock region (red), respectively. To avoid employing clock resources of the given clock region, all cells of a clock net must be simultaneously placed into one of these four escaping regions.

2 to line 5. Then from line 6 to line 9, for clock nets occupying clock resources in the target clock region, we compute their moving costs to each of these four escaping regions and store the calculation results as candidates for later overflow resolving. Within the loop from line 14 to line 21, these candidates are accessed in the ascending order of their costs. For each candidate, at line 15, we first ensure that no other candidate associated with the same clock net has been chosen and then call function `IsFeasible` to reject candidates leading to infeasible solutions. The feasibility checking and function `IsFeasible` will be further discussed later in this section. If a candidate is feasible, we block all assignments between cells in the candidate clock net and clock regions outside the candidate escaping region by setting the corresponding penalty multipliers to infinity from line 16 to line 18. This operation prevents the target clock region from being further employed by the candidate clock in the later assignment iterations. The loop from line 14 to line 21 is repeated until all overflows in the target clock are fully resolved or the number of resolved overflows reaches

the limit I_{max}, which is 2 in UTPlaceF 2.0 by default. Intuitively, I_{max} is the maximum descent step for clock overflow resolving. By using a smaller I_{max}, clock congestion can be resolved more smoothly and evenly among different clock regions with the cost of more assignment iterations.

Now we explain the function $\mathtt{Cost}(e, b)$ that computes the cost of pushing clock e to escaping region b. The objective of our assignment is to minimize the total pin movement. In addition, each cell movement may lead to logic resource overflow in the target escaping region. Thus the cost consists of two components: the pin movement cost and the logic resource cost, shown as follows:

$$Cost(e, b) = \sum_{i \in \mathcal{V}(e)} (P_i\, d_{i,b} + \alpha A_i^{\mathrm{CLB}} + \beta A_i^{\mathrm{DSP}} + \gamma A_i^{\mathrm{RAM}}), \qquad (3.3)$$

where $\mathcal{V}(e)$ denotes the set of cells in clock net e, and $d_{i,b}$ denotes the physical distance between cell i and box region b. The trade-off weights α, β, and γ are set to 5, 50, and 50, respectively, in our experiments.

The only thing left now is the feasibility checking function $\mathtt{IsFeasible}(e, b, \boldsymbol{\lambda})$. For an intermediate solution with the logic resource constraint satisfied, arbitrarily blocking a set of assignments cannot further guarantee the existence of feasible solutions. For example, if clock nets are restricted in small regions without enough logic resources to accommodate all the cells, the corresponding assignment problem will be infeasible. One straightforward method to avoid this issue is to solve the updated minimum-cost flow for the graph in Figure 3.5 to check if a feasible solution exists before

actually applying the new assignment blocking. However, this method is far too cumbersome to be applied in our iterative framework, which motivates us to propose a light-weight yet effective feasibility checking method.
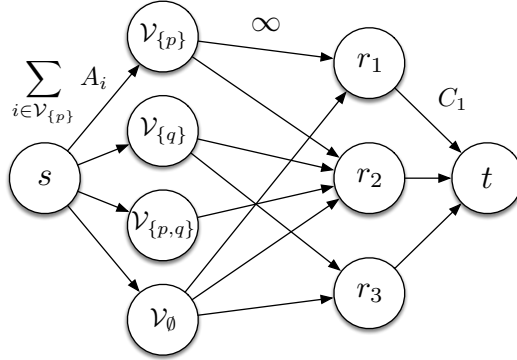
Figure 3.7: Illustration of our maximum-flow-based feasibility checking. Numbers on edges represent their capacities.

Instead of solving the minimum-cost flow for the graph in Figure 3.5, we construct a maximum-flow graph, as illustrated in Figure 3.7, to perform fast feasibility checking. Here we are trying to assign a netlist with two clock nets p and q to three clock regions r_1, r_2, and r_3. We divide all cells into four mutually exclusive groups $\mathcal{V}_{\{p\}}$, $\mathcal{V}_{\{q\}}$, $\mathcal{V}_{\{p,q\}}$, and $\mathcal{V}_{\varnothing}$, based on the clock nets they belong to, where $\mathcal{V}_p$ and $\mathcal{V}_q$ denote the groups of cells belonging only to p and q, respectively, $\mathcal{V}_{\{p,q\}}$ denotes the group of cells belonging to both p and q, and $\mathcal{V}_{\varnothing}$ is the set of cells without clock nets. In this graph construction, we only introduce edges for unblocked assignments ($\lambda_{i,j} \neq \infty$). In this example, clock p cannot be assigned to r_3 and clock q cannot be assigned to r_1. Then, with the proper edge capacity settings as shown in Figure 3.7, checking the assignment

feasibility of Formulation (3.2) is equivalent to computing the maximum flow of amount $\sum_{i \in \mathcal{V}} A_i$ in this graph. If the resulting maximum flow value is equal to $\sum_{i \in \mathcal{V}} A_i$, a feasible solution with the set of new assignment blocking applied must exist, otherwise, the assignment problem is infeasible. It should be noted that our maximum-flow-based checking are performed for different resource types (e.g., CLB, DSP, and RAM) separately each time and we only conclude the updated problem is feasible when all checkings are passed.

Compared with the minimum-cost flow graph in Figure 3.5, the problem size is dramatically reduced by the clock grouping in the graph construction in Figure 3.7. In addition, edge costs are removed in the maximum-flow graph since we solve it only for feasibility checking purpose. Therefore, the proposed maximum-flow-based technique enables a much faster yet reliable feasibility checking process.

3.3.2.2.4 Speed-up Techniques

The minimum-cost flow formulation shown in Figure 3.5 is optimal for a given λ and unit logic resource demand but it may suffer from long runtime for large designs. Instead of solving flat minimum-cost flow problems, we here propose a geometric clustering technique to reduce the problem size by grouping cells that are physically close and share the same set of clock nets together. Our proposed geometric clustering technique can significantly speed up the minimum-cost flow solving process while preserving good solution quality.

Here we use a simple example in Figure 3.8 to illustrate our clustering

145
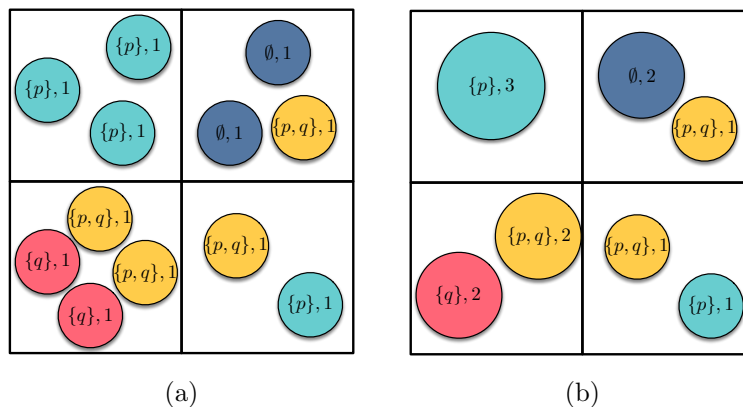
Figure 3.8: Illustration of our geometric clustering technique. (a) Twelve cells need to be assigned in the flat minimum-cost flow without the clustering technique. (b) With our geometric clustering applied, the number of objects needs to be assigned is reduced to seven. "$\mathcal{S}, A$" pair (e.g., $\{p, q\}$, 1) on each cell denotes the set of clock nets it belongs to and its logic resource demand, respectively. $\varnothing$ means the cell does not belong to any clock nets.

idea. In this example, there are twelve cells and two clock nets p and q. The "$\mathcal{S}, A$" pair (e.g., $\{p, q\}$, 1) on each cell denotes the set of clock nets it belongs to and its logic resource demand, respectively. If the flat minimum-cost-flow-based assignment in Figure 3.5 is directly applied, we need to assign twelve objects, as shown in Figure 3.8(a), in the problem. However, if we divide all twelve cells into four bins based on the two-by-two geometric partitioning shown in Figure 3.8 and cluster cells that have the same clock signatures within each partition, the number of objects to be assigned will reduce to seven, as

shown in Figure 3.8(b).

$$\min_{\boldsymbol{x}} \quad \sum_{k \in \mathcal{K}} \sum_{j \in \mathcal{R}} (\frac{P_k}{A_k} D_{k,j} + \lambda_{k,j}) x_{k,j}, \tag{3.4a}$$

$$\text{s.t.} \quad x_{k,j} \in \{0, 1, 2, \dots\}, \forall k \in \mathcal{K}, \forall j \in \mathcal{R}, \tag{3.4b}$$

$$\sum_{j \in \mathcal{R}} x_{k,j} = A_k, \forall k \in \mathcal{K}, \tag{3.4c}$$

$$\sum_{k \in \mathcal{K}} x_{k,j} \le C_j, \forall j \in \mathcal{R}. \tag{3.4d}$$

With our clustering technique applied, instead of solving Formulation (3.2), we solve a very similar Formulation (3.4), where $\mathcal{K}$ denotes the set of clusters, P_k and A_k denote the total pin count and total logic resource demand of cells in cluster k, and $D_{k,j}$ denotes the physical distance between the average location of cells in cluster k and clock region j. Besides, unit logic resource demand is assumed for all cells (not clusters).

Formulation (3.4) can still be solved utilizing the minimum-cost flow transformation illustrated in Figure 3.5. However, comparing to Formulation (3.2), several key differences need to be emphasized.

Firstly, since inaccuracy is injected to pin movement calculation by using averaged pin count $\frac{P_k}{A_k}$ and averaged cell locations for $D_{k,j}$ in the objective (3.4a), less optimal solutions will be obtained as the cluster size increases. On the other hand, a larger cluster size can dramatically reduce the problem size, which results in much faster runtime. Therefore, with different partition sizes, we can achieve different tradeoffs between quality and runtime. In UTPlaceF 2.0, the partition width and height are empirically set to 5 CLB

147

sites.

Secondly, in the minimum-cost flow graph with our clustering technique applied, each assignment object represents a cluster of cells and does not have unit logic resource demand anymore (i.e., $A_k \geq 1$). Thus, the final minimum-cost flow solution of Formulation (3.4) might assign non-zero flows to more than one edge of an assignment object. It is physically equivalent to splitting a cluster into subgroups and assigning them to multiple clock regions. For each of these split cases, to decide which cell goes to which clock region, one more level of minimum-cost flow corresponding to Formulation (3.2) is needed. Note that, this second-level minimum-cost flow is only for cells within the split cluster and the set of clock regions that this cluster has been assigned to. Since we assume unit resource demand for all cells, the split assignment would not happen again in these second-level minimum-cost flows and the solution feasibility can be guaranteed.

3.3.2.3 Clock-Aware Packing

Packing is responsible for clustering LUTs and FFs into CLBs that satisfy all packing rules, meanwhile, optimizing various design metrics such as power, timing, and channel width. In UTPlaceF (Section 2.3), packing is performed by pairwisely merging cells based on a set of attraction functions, which grant higher priorities to cell pairs that are physically closer and share more small nets. Although the packing algorithm in UTPlaceF is effective for wirelength and routability optimization, it is no longer able to guarantee

148

solution quality with the newly introduced clock region constraint.
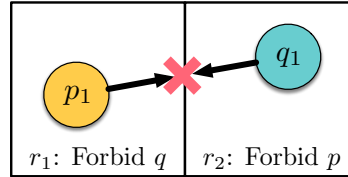
Figure 3.9: An example to show the necessity of clock-aware packing. p and q denote two clock nets. p_1 and q_1 are two cells belonging to clock p and q, respectively.

One example to illustrate the necessity of clock awareness for packing is shown in Figure 3.9. In this example, clock q is forbidden in clock region r_1 and clock p is forbidden in clock region r_2 according to the clock region assignment result in FIP. If we, unfortunately, cluster p_1 and q_1 together, the packing solution would become infeasible since no clock region can simultaneously contain clock p and q. Therefore, it is critical to making packing algorithms clock-aware for the solution feasibility.

3.3.2.3.1 Probability-Based Clock Distribution Estimation

One common idea to avoid the case shown in Figure 3.9 is to let packing algorithms honor the clock region assignment solution after FIP. However, this idea imposes another question: how to deal with clock regions with spare global clock resources. In general, not all the global clock resources are employed after clock region assignment, so if we can further allocate these spare resources to some clock nets, packing algorithms might be able to explore a larger solution space and produce even better packing solutions. Motivated

by this reason, we propose a probabilistic model to estimate the probability of each clock occupying each clock region in the final placement solution. The estimation results will be incorporated into our new packing algorithm (See Section. 3.3.2.3.2) to help us smartly make use of those spare clock resources.
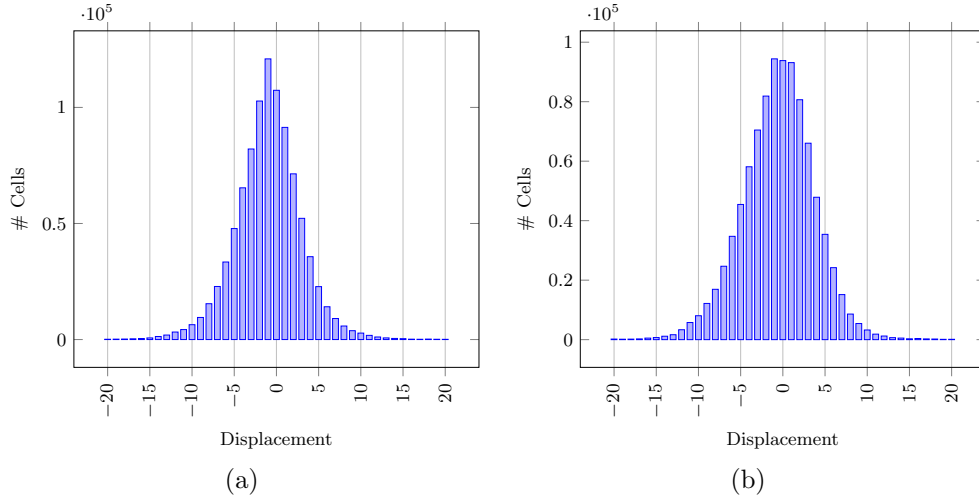
Figure 3.10: Distributions of cell displacement in the final placement w.r.t the FIP in (a) x-direction and (b) y-direction based on `CLK-FPGA05` in the ISPD 2017 contest benchmark suite. All placement solutions are generated by UTPlaceF without clock legality consideration.

The potential clock distribution mismatch between FIP and the final placement is introduced by the cell movement between them. We collect the displacement of all cells in the final placement solution w.r.t. their locations in the FIP for a representative benchmark and the result is shown in Figure 3.10. It can be seen that the cell displacement in both x- and y-directions have a bell-shaped distribution around zero. So it is reasonable to make the assumption that the cell displacement satisfies the Gaussian distribution with a mean value

150

of zero in both x- and y-directions, which can be written as follows:

$$X' - X \sim \mathcal{N}(0, \sigma_x^2), \tag{3.5a}$$

$$Y' - Y \sim \mathcal{N}(0, \sigma_y^2). \tag{3.5b}$$

where (X', Y') and (X, Y) denote (x, y) coordinates of cells in the final placement and FIP, respectively, $\mathcal{N}(\mu, \sigma^2)$ represents a Gaussian distribution with mean value μ and standard deviation σ, and σ_x and σ_y are estimated standard deviations for cell displacement in x- and y-directions.

Given the assumption in Equation (3.5), a cell i at location (x_i, y_i) in FIP, and any region r with bounding box (xl_r, yl_r, xh_r, yh_r), the probability of cell i being placed into the region r in the final placement, denoted by $h_{i,r}$, can be written as follows:

$$
\begin{aligned}
h_{i,r} =& \big(\text{CDF}(xh_r, x_i, \sigma_x) - \text{CDF}(xl_r, x_i, \sigma_x)\big) \\
& \big(\text{CDF}(yh_r, y_i, \sigma_y) - \text{CDF}(yl_r, y_i, \sigma_y)\big),
\end{aligned}
\tag{3.6}
$$

where $\text{CDF}(x, \mu, \sigma)$ is the cumulative distribution function of the Gaussian distribution $X \sim \mathcal{N}(\mu, \sigma^2)$ evaluated at x. It represents the probability that X takes value less than or equal to x. CDF is defined in Equation (3.7), where $\text{erf}(x)$ denotes the error function [5]. Figure 3.11(a) gives a visual illustration of the cell-to-region probability calculation process.

$$\text{CDF}(x, \mu, \sigma) = \frac{1}{2}\Big[1 + \text{erf}\Big(\frac{x - \mu}{\sigma\sqrt{2}}\Big)\Big]. \tag{3.7}$$

Now, we show the method to calculate the probability of clock region
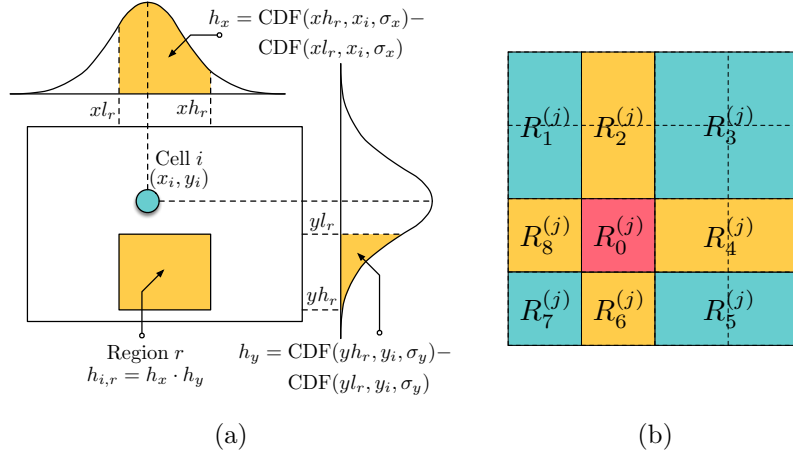
Figure 3.11: (a) A visual illustration for the cell-to-region probability calculation shown in Equation (3.6). (b) Nine sets of clock regions, $R_0^{(j)}, R_1^{(j)}, \ldots, R_8^{(j)}$, corresponding to clock region j (red).

j having clock demand from clock net k in the final placement. Given a clock region j, we divide all clock regions into nine sets, as shown in Figure 3.11(b), $\mathcal{R}_0^{(j)}, \mathcal{R}_1^{(j)}, \ldots, \mathcal{R}_8^{(j)}$, where $\mathcal{R}_0^{(j)}$ contains only clock region j. It is easy to see that the bounding box of any clock net k does not intersect with the clock region j if and only if all cells in the clock net k are placed in one of the four regions that are above, below, to the left of, and to the right of the clock region j ($\mathcal{R}_0^{(j)}$), i.e., $\mathcal{R}_1^{(j)} \cup \mathcal{R}_2^{(j)} \cup \mathcal{R}_3^{(j)}$, $\mathcal{R}_5^{(j)} \cup \mathcal{R}_6^{(j)} \cup \mathcal{R}_7^{(j)}$, $\mathcal{R}_7^{(j)} \cup \mathcal{R}_8^{(j)} \cup \mathcal{R}_1^{(j)}$, and $\mathcal{R}_3^{(j)} \cup \mathcal{R}_4^{(j)} \cup \mathcal{R}_5^{(j)}$. Therefore, the probability of clock region j not being occupied by clock net k, denoted by $\overline{H_{j,k}}$, can be defined using Equation (3.8).

$$\overline{H_{j,k}} = \sum_{\substack{(l,m,n)\in \\ \{(1,2,3),(3,4,5), \\ (5,6,7),(7,8,1)\}}} \prod_{i\in\mathcal{V}(k)} (h_{i,\mathcal{R}_l^{(j)}} + h_{i,\mathcal{R}_m^{(j)}} + h_{i,\mathcal{R}_n^{(j)}}) - \sum_{l\in\{1,3,5,7\}} \prod_{i\in\mathcal{V}(k)} h_{i,\mathcal{R}_l^{(j)}}. \quad (3.8)$$

Finally, the probability of clock region j having clock demand from clock net

152

k in the final placement, denoted by $H_{j,k}$, can be defined as follows:

$$H_{j,k} = 1 - \overline{H_{j,k}}. \tag{3.9}$$

3.3.2.3.2 Clock-Aware Packing Attraction Function

UTPlaceF 2.0 adopts the UTPlaceF packing framework, which consists of the maximum-weighted-matching-based and BestChoice-based [60] clustering algorithms. Both algorithms rely on attraction functions to evaluate the goodness of any pairwise clustering and iteratively merge high-attraction object pairs to form CLBs. To produce clock-friendly CLB-level netlists for placement, the packing attraction functions in UTPlaceF are enhanced to be clock-aware in UTPlaceF 2.0.

The original attraction functions in UTPlaceF for connected objects consist of two components. The first component is the distance score, which exponentially penalizes objects that are physically far away in FIP. The second component is the connectivity score, which grants higher priorities to objects that share more small nets [40]. The original packing attraction function for two connected objects i and j then can be generalized as follows:

$$\phi_{i,j} = \Big(1 - exp\big(\gamma(d_{i,j} - D^U)\big)\Big) \sum_{e \in \mathcal{E}_i \cap \mathcal{E}_j} \frac{k}{|e| - 1}. \tag{3.10}$$

where $d_{i,j}$ denotes the physical distance between i and j in FIP, $\mathcal{E}_i \cap \mathcal{E}_j$ denotes the set of nets that are shared by i and j, $|e|$ represents the number of pins in net e, k is 2 for two-pin nets and 1 for other nets, and γ and D^U are tuning

parameters determined experimentally.

The proposed new attraction function is the same as the original one, except that a new term is introduced to account for the clock legality. It is described by the following expression,

$$\phi_{i,j} = \left(1 - exp\big(\gamma(d_{i,j} - D^U)\big)\right) \sum_{e \in \mathcal{E}_i \cap \mathcal{E}_j} \frac{k}{|e| - 1} \prod_{k \in \mathcal{E}_i \cup \mathcal{E}_j} H_{t,k}. \qquad (3.11)$$

where $\mathcal{E}_i \cup \mathcal{E}_j$ denotes the set of clock nets that contain at least one of cell i and cell j, t represents the clock region that the center of gravity of i and j falls into, and $H_{t,k}$ is the probability defined in Equation (3.9). Here we use clock region t as the estimated target clock region to place the cluster of i and j. Intuitively, the new term represents the probability that a cluster can be placed into its estimated target clock region without any clock conflicts. So integrating this new term to the original attraction function helps to block out clustering cases that are likely to violate the clock region constraint.

The same clock penalty term is applied to all attraction functions in UTPlaceF to make the whole packing flow clock-aware. With our enhanced clock-aware attraction functions in UTPlaceF 2.0, better wirelength in the final solutions is demonstrated by our experiments.

3.3.2.4   Half-Column Region Assignment

As shown in Figure 3.3, the half-column region constraint is explicitly honored in the legalization stage by our half-column region assignment technique. Here we formally define the half-column region assignment problem as

follows.

Problem 3 (Half-Column Region Assignment) *Given a roughly-legalized placement, logic resource capacity of each half-column region, and a feasible clock region assignment solution, assign cells within each clock region to half-column regions to minimize total pin movement without logic resource or clock overflow.*

Problem 3 is very similar to the clock region assignment problem defined in Problem 2. It can also be formulated into Formulation (3.1) but with a simpler clock constraint (3.1e). So the same relaxation and minimum-cost flow framework described in Section 3.3.2.2.1 and Section 3.3.2.2.2 can be seamlessly applied to solve the half-column region assignment problem as well.

The only notable difference compared with the clock region assignment is the method to update penalty multipliers for clock overflow resolving. Our proposed penalty multiplier updating algorithm for half-column region constraint is summarized in Algorithm 11. It should be noted that, given a feasible clock region assignment solution, the half-column region assignment can be performed within each clock region independently without impacting clock legality. So the scope of Algorithm 11 is only limited in a single clock region.

Within each clock region, we first find the target half-column region with the largest clock overflow at line 1. Then, the cost of moving each clock out of the target half-column region is calculated and stored in a list from

155

Algorithm 11: Penalty Multiplier Updating for Half-Column
Region Constraint

    Input   : Penalty multiplier $\boldsymbol{\lambda}$.
    Output: An updated penalty multiplier $\boldsymbol{\lambda}$ that will result in less
             clock overflow.

1 Find the half-column region r with the largest clock overflow O_{max};
2 **foreach** *clock e in r* **do**
3 $l \leftarrow l \cup \{\texttt{Cost}(e, r)\}$ (See Equation (3.12));
4 **end**
5 Sort candidates in l by ascending order of their cost;
6 $i \leftarrow 0$;
7 **foreach** $\texttt{Cost}\ (e, r) \in$ *sorted* l **do**
8   **if** $\neg\texttt{IsFeasible}(e, r, \boldsymbol{\lambda})$ **then continue**;
9   **foreach** *cell i in clock e* **do**
10 $\lambda_{i,r} \leftarrow \infty$;
11   **end**
12 $i \leftarrow i + 1$;
13   **if** $i \geq O_{max}$ **then return**;
14 **end**

---

line 2 to line 4. The cost of moving clock e out of half-column r is defined as
follows:

$$Cost(e, r) = \sum_{i \in \mathcal{V}(e) \cap \mathcal{V}(r)} P_i, \tag{3.12}$$

where $\mathcal{V}(e) \cap \mathcal{V}(r)$ denotes the set of cells that are simultaneously in clock e and
half-column region r, and P_i represents the total number of pins associated
with cell i.

In the loop from line 7 to line 14, we iteratively pick the clock with
the smallest moving cost and block it out of the target half-column region
in the subsequent assignment iterations. A feasibility checking similar to the

one illustrated in Figure 3.7 is performed at line 8 to ensure the existence of feasible assignments after the edge blocking. This process is repeated until the number of iterations hits the overflow limit at line 13.

3.3.3 Experimental Results

UTPlaceF 2.0 is implemented in C++ and compiled by g++ 4.7.2. The official contest evaluation results conducted by Xilinx is used to demonstrate the effectiveness of UTPlaceF 2.0. All the experiments are performed on a Linux machine running with 3.40 GHz Intel Core i7 CPU and 32 GB RAM, using only one CPU core. The benchmark suite released by Xilinx for the ISPD 2017 clock-aware FPGA placement contest was used to evaluate the efficiency of UTPlaceF 2.0.

The characteristics of the ISPD 2017 benchmark suite are listed in Table 3.2. This benchmark suite consists of industry-strength designs with gate counts ranging from 0.45 million to about 1 million. Several designs in this suite have extremely high resource utilization and clock usage.

As UTPlaceF 2.0 won the first-place award of the ISPD 2017 clock-aware FPGA placement contest, we here compare our results with the second- and third-place contest winners as shown in Table 3.3. Metrics "WL" and "RT" represent the routed wirelength and runtime, while "WLR" and "RTR" represent the wirelength and runtime ratios normalized to UTPlaceF 2.0. It can be seen that UTPlaceF 2.0 achieves the best overall routed wirelength and outperforms by 3.3% and 9.7% in routed wirelength compared with the other

Table 3.2: ISPD 2017 Placement Contest Benchmarks Statistics

Designs	#LUT	#FF	#RAM	#DSP	#Clock
CLK-FPGA01	211K	324K	164	75	32
CLK-FPGA02	230K	280K	236	112	35
CLK-FPGA03	410K	481K	850	395	57
CLK-FPGA04	309K	372K	467	224	44
CLK-FPGA05	393K	469K	798	150	56
CLK-FPGA06	425K	511K	872	420	58
CLK-FPGA07	254K	309K	313	149	38
CLK-FPGA08	212K	257K	161	75	32
CLK-FPGA09	231K	358K	236	112	35
CLK-FPGA10	327K	506K	542	255	47
CLK-FPGA11	300K	468K	454	224	44
CLK-FPGA12	277K	430K	389	187	41
CLK-FPGA13	339K	405K	570	262	47
Resources	538K	1075K	1728	768	-

two contest winners, respectively. Running in a single thread, UTPlaceF 2.0 completes the largest benchmark `CLK-FPGA13` (0.96M cells) in 20 minutes.

We also report the runtime breakdown of UTPlaceF 2.0 in Figure 3.12(a). On average, the majority (68.4%) of the total runtime is taken by FIP, while the detailed placement, CLB-level global placement, packing, and legalization consume 20.0%, 5.3%, 3.3%, and 0.3% of the total runtime, respectively. We further divide the runtime of FIP into four components, as shown in Figure 3.12(b), where the preconditioned conjugate gradient for quadratic programming takes 88.8% of the FIP runtime, followed by 4.9% and 4.2% for rough legalization and density-preserving global move, respectively, and only 0.7% is taken by the clock region assignment.

158

Table 3.3: Routed Wirelength (WL in 10^3) and Runtime (RT in seconds) Comparison with the Contest Winners

Benchmark	2nd place				3rd place				UTPlaceF 2.0 (1st place)			
	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR
CLK-FPGA01	2209	3023	1.001	5.68	2269	354	1.027	0.67	2208	532	1.000	1.00
CLK-FPGA02	2274	3153	0.998	6.15	2504	333	1.099	0.65	2279	513	1.000	1.00
CLK-FPGA03	6229	4066	1.164	3.91	5803	666	1.084	0.64	5353	1039	1.000	1.00
CLK-FPGA04	3817	3077	1.032	4.33	4086	464	1.105	0.65	3698	711	1.000	1.00
CLK-FPGA05	4995	3631	1.065	3.87	5181	680	1.104	0.72	4692	939	1.000	1.00
CLK-FPGA06	5606	3836	1.003	3.60	6217	695	1.112	0.65	5589	1066	1.000	1.00
CLK-FPGA07	2505	3953	1.024	4.68	2676	410	1.095	0.49	2445	845	1.000	1.00
CLK-FPGA08	1990	4395	1.055	8.31	2057	277	1.091	0.52	1886	529	1.000	1.00
CLK-FPGA09	2583	5428	0.995	6.45	2814	414	1.084	0.49	2597	842	1.000	1.00
CLK-FPGA10	4770	3305	1.069	3.39	4840	516	1.084	0.53	4464	974	1.000	1.00
CLK-FPGA11	4208	4341	1.006	4.06	4777	548	1.142	0.51	4184	1068	1.000	1.00
CLK-FPGA12	3377	4949	1.002	6.39	3740	413	1.110	0.53	3369	774	1.000	1.00
CLK-FPGA13	3921	3748	1.019	3.20	4320	548	1.123	0.47	3848	1172	1.000	1.00
Norm.	-	-	1.033	4.92	-	-	1.097	0.58	-	-	-	-
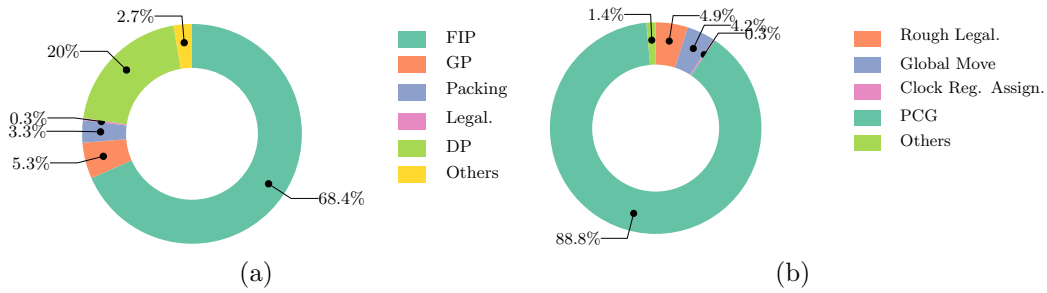
Figure 3.12: The runtime breakdown of (a) UTPlaceF 2.0 and (b) flat initial placement (FIP).

3.3.3.1 Efficiency Validation of Maximum-Flow-Based Feasibility Checking

We validate the efficiency of our maximum-flow-based feasibility checking proposed in Section 3.3.2.2.3 by experiments shown in Table 3.4. The column 2 - 5 separately list the number of solvings (# Solving) and total solving time (ST) of minimum-cost flow (MCF) and maximum flow (MF) for each benchmark. The column 6 gives the total solving time taken by MCF and MF.

Without the maximum-flow-based feasibility checking, an MCF instead of an MF solving is required to check if the updated penalty multipliers are feasible. In this case, our clock region assignment would become a pure MCF-based algorithm. We project the required solving time of it by Equation (3.13) and report the results in the column 7 of Table 3.4. The overall speedup of "MCF + MF" over "Pure MCF" for each benchmark is listed in the last column of Table 3.4.

$$\text{Proj. Pure MCF ST} = \frac{\text{MCF ST}}{\text{\# MCF Solving}}(\text{\# MCF Solving} + \text{\# MF Solving}) \tag{3.13}$$

Table 3.4: Runtime Speedup by Applying Maximum-Flow-Based Feasibility Checking

Benchmark	MCF		MF		MCF + MF	Proj. Pure MCF	Speedup
	#Solving	ST (sec)	#Solving	ST (sec)	ST (sec)	ST (sec)	
CLK-FPGA01	90	0.96	0	0.00	0.96	0.96	×1.00
CLK-FPGA02	120	1.19	60	0.01	1.20	1.79	×1.49
CLK-FPGA03	738	22.10	1227	0.34	22.44	58.84	×2.62
CLK-FPGA04	219	4.25	221	0.05	4.30	8.54	×1.99
CLK-FPGA05	171	4.57	168	0.05	4.62	9.06	×1.96
CLK-FPGA06	282	9.58	342	0.11	9.69	21.20	×2.19
CLK-FPGA07	84	0.91	0	0.00	0.91	0.91	×1.00
CLK-FPGA08	96	0.77	0	0.00	0.77	0.77	×1.00
CLK-FPGA09	87	0.95	0	0.00	0.95	0.95	×1.00
CLK-FPGA10	222	4.15	159	0.03	4.18	7.12	×1.70
CLK-FPGA11	216	4.79	153	0.03	4.82	8.18	×1.70
CLK-FPGA12	123	1.88	42	0.01	1.89	2.52	×1.33
CLK-FPGA13	93	1.61	12	0.01	1.62	1.82	×1.12

It can be observed that MF is about one to two orders of magnitude faster than MCF in our experiments. By applying the maximum-flow-based feasibility checking, we can achieve up to ×2.6 overall speedup. Note that the runtime of MCF is dependent to the partition size mentioned in Section 3.3.2.2.4 but the MF-based checking is not. Therefore, even more speedup can be achieved if smaller partitions are used in the clock region assignment.

3.3.3.2 Trade-off of Different Partition Sizes
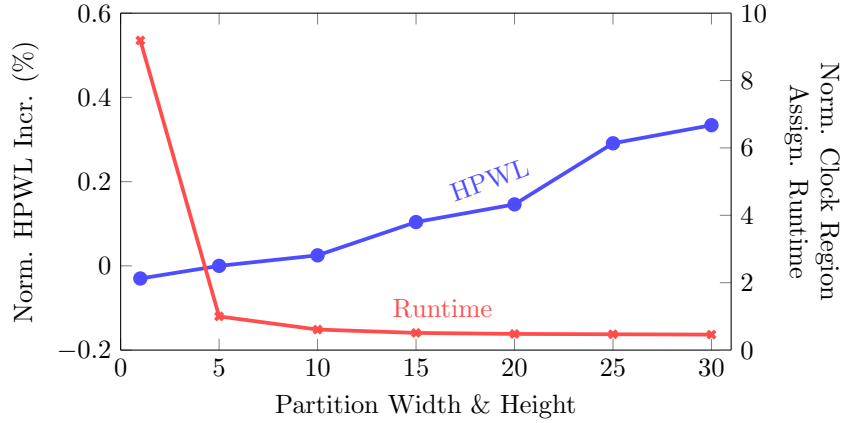
Figure 3.13: Trend of HPWL and clock region assignment runtime with different partition sizes for clustering (Section 3.3.2.2.4) on benchmark `CLK-FPGA05`. All HPWL and runtime are normalized to the result of partition width and height = 5.

Figure 3.13 gives the trade-off between HPWL and clock region assignment runtime under different partition sizes. We perform experiments on a representative benchmark, `CLK-FPGA05`, with partition width and height set to 1, 5, 10, 15, 25, and 30, respectively. All the HPWL and runtime are normalized to the result with height and width of 5. As can be seen, with larger partition sizes, the runtime drops quickly and saturates at around 0.4 (normalized runtime) while the wirelength increases steadily but slowly. Considering the wirelength is not very sensitive to the partition size, we experimentally choose partition width and height = 5 to achieve fast runtime and reasonably good solution quality.

3.3.3.3   Effectiveness Validation of Clock-Aware Packing

Figure 3.14 shows the normalized HPWL increases (less is better) of placements with and without clock-aware packing compared with the lower-bound placements that ignore all clock constraints. As can be seen, clock-aware packing, for most benchmarks, delivers better wirelength over original non-clock-aware packing. Another key observation is that, with both proposed clock region assignment and clock-aware packing techniques applied, UTPlaceF 2.0 can satisfy all clock constraints by only paying a little wirelength overhead ($< 1\%$).
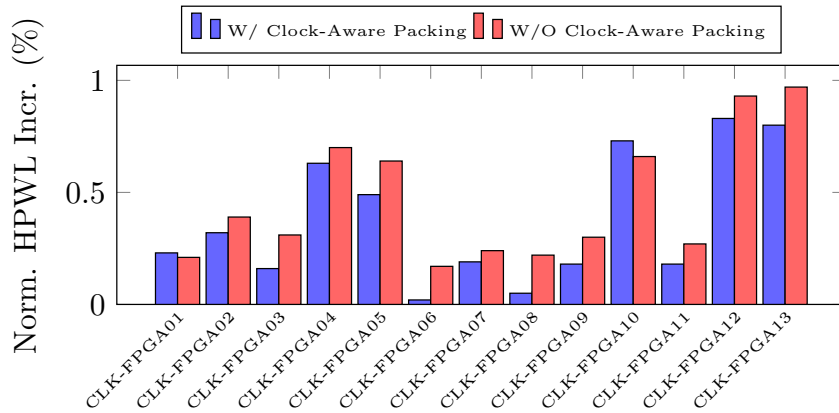
Figure 3.14: Normalized HPWL increases (%) of placements w/ and w/o clock-aware packing compared with placements without any clock constraint considerations.

3.3.4   Summary

With the increasing complicated FPGA clocking architecture, honoring clock rules is becoming a fundamental issue in modern FPGA implementation

162

flow. In this section, we have proposed a high-performance clock-aware FPGA placement engine, UTPlaceF 2.0, which is capable of satisfying clock rules for state-of-the-art FPGA devices while still maintaining good wirelength and routability. An iterative minimum-cost-flow-based clock region assignment framework, a probability-based clock distribution estimation method, and a clock-aware packing technique are proposed for better honoring clock legality throughout the whole placement and packing process. As the first-place winner of ISPD 2017 clock-aware FPGA placement contest, UTPlaceF 2.0 can efficiently achieve legal and high-quality placement solutions, which outperforms all other contest placers in routed wirelength with competitive runtime.

3.4 UTPlaceF 2.X: Simultaneous FPGA Placement and Clock Tree Construction

In Section 3.3, we have proposed an efficient clock-aware placement engine UTPlaceF 2.0. In UTPlaceF 2.0 and several recent works [14,64], clock routing of clock networks is approximated by the bounding boxes of their clock loads to simplify the clock legalization problem. Figure 3.15(a) gives an example of this approximated modeling. It shows the distribution of all the clock loads, including CLBs, DSPs, and RAMs, in a clock network. By using the bounding box modeling method, this clock network consumes clock routing resources in all the clock regions overlapped with its bounding box (shaded regions). However, we observe that this modeling method often overestimates the actual clock routing demands. This can be illustrated by Figure 3.15(b). It shows the same clock load distribution as that in Figure 3.15(a), but here, a clock tree (the bold black lines) is constructed to reveal the actual clock routing demands. Compared with the bounding box estimation in Figure 3.15(a), which consumes 9 clock regions, the same clock network only spans 6 clock regions with tree construction in Figure 3.15(b).

Apart from the clock routing modeling inaccuracy, most previous works resolve clock routing congestions in a greedy and iterative manner. Specifically, they repeatedly push clock loads away from overflowed clock regions while greedily minimizing the placement disturbance in each step. Such a method, however, can only explore a very narrow solution space and always follows the decisions made previously, which can lead to very suboptimal or even infeasible
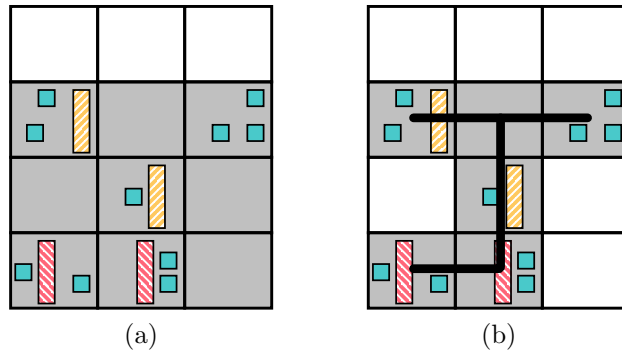
Figure 3.15: Illustration of the clock routing demand calculation using (a) the bounding box of clock loads and (b) the actual clock tree. Both figures show the same clock network with the same load distribution. Shaded areas denote the occupied clock regions. By using bounding box modeling in (a), the clock network occupies 9 clock regions, while the actual clock tree only spans 6 clock regions in (b).

solutions.

To remedy the aforementioned deficiencies in previous works, this section presents a generic framework, UTPlaceF 2.X, that simultaneously optimizes placement and ensures clock feasibility by explicit clock tree construction. Inspired by the branch-and-bound idea [36], we generalize the clock legalization as a tree-space exploration process. By doing so, UTPlaceF 2.X can explore a larger solution space and potentially produce better solutions compared with conventional greedy approaches like UTPlaceF 2.0. Our major contributions are highlighted as follows.

- Inspired by the branch-and-bound method, we interpret the solution space of clock routing as a tree and then generalize the clock legalization as a tree exploration process of finding legal solutions.

- We propose a novel Lagrangian relaxation-based clock tree construction technique to accurately model the clock routing demands during FPGA placement.

- We tentatively study different ways of constructing and exploring the solution-space tree and evaluate their impact on the overall quality of results and efficiency.

- We integrate the proposed techniques in a placer called UTPlaceF 2.X. Compared with other state-of-the-art methods on the ISPD 2017 clock-aware placement contest benchmarks [80], the proposed UTPlaceF 2.X achieves the best overall routed wirelength with competitive runtime.

The rest of this section is organized as follows. Section 3.4.1 gives the problem definition. Section 3.4.2 overviews UTPlaceF 2.X flow and details the proposed algorithms. Section 3.4.3 shows the experimental results, followed by the summary in Section 3.4.4.

3.4.1 Preliminaries
3.4.1.1 Problem Definition

In placement problem, routed wirelength is treated as one of the most important quality metrics since it is a good first-order approximation of overall performance (frequency) and power. Therefore, in this work, our objective is to minimize the routed wirelength. Given the clocking architecture (Section 3.2)

and the optimization objective, we now define our *simultaneous FPGA placement and clock tree construction* problem as follows.

Problem 4 (Simultaneous FPGA Placement and Clock Tree Construction) *Given an FPGA netlist, produce a placement with minimized routed wirelength and a corresponding clock routing solution that satisfies the target clocking architecture.*

3.4.2   UTPlaceF 2.X Algorithms
3.4.2.1   Overview of the Proposed Flow

The overall flow of UTPlaceF 2.X is shown in Figure 3.16. UTPlaceF 2.X is built on top of our UTPlaceF-DL described in Section 2.4 and it consists of three major phases: (1) wirelength-driven placement; (2) clock-driven placement; (3) legalization and detailed placement.

In each iteration of the wirelength-driven placement phase, a quadratic program is solved to minimize the wirelength and the rough legalization [33] technique is conducted to eliminate cell overlaps. This loop is repeated until the lower-bound wirelength and the upper-bound wirelength ratio [33] (LB/UB WL Ratio) converges to 0.9. In the clock-driven placement phase, an extra *clock network planning* step is performed right after the conventional quadratic placement. It seeks to construct a legal clock routing solution with minimized placement perturbation. After that, we assign cells to their feasible clock regions induced from the resulting clock routing and conduct rough legalization only within each clock region to preserve the clock legality. This clock region-

167
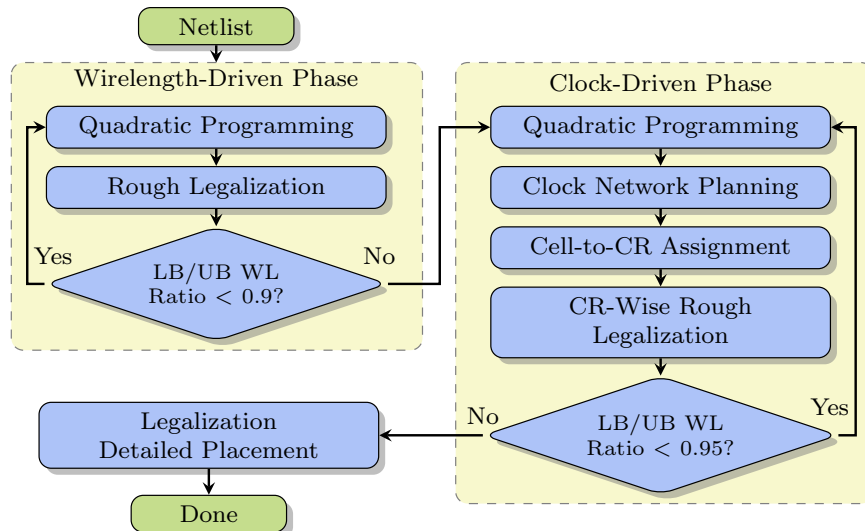
Figure 3.16: The overall flow of UTPlaceF 2.X.

wise rough legalization updates the anchor forces [33] of cells to point to their feasible locations found in the current placement iteration and pull them to form a more clock-feasible solution in the next iteration. The clock-driven placement phase stops when the wirelength fully converges. Finally, legalization and detailed placement are performed to further optimize the placement result while honoring the previously achieved clock routing.

As the centerpiece of UTPlaceF 2.X (Figure 3.16), the *clock network planning* step will be elaborated later in this section. We first define the *clock network planning* problem and give its mathematical formulation in Section 3.4.2.2. Then, in Section 3.4.2.3, we give an intuitive explanation of a general mathematical method, the branch-and-bound method, to solve a class of problems like this. We will show that our proposed algorithms share a similar underlying idea with the branch-and-bound method. The details of them

168

are given in Section 3.4.2.4 – 3.4.2.8.

3.4.2.2   The Clock Network Planning Problem

A well-optimized placement (in terms of conventional metrics, like wire-length, power, and timing) can often fail the clock routing. For such a case, the goal of our *clock network planning* is to find a clock-feasible solution that greatly preserves the given optimized placement. Therefore, our objective here is to minimize the total cell movement. Meanwhile, the following two constraints also need to be satisfied: (1) there should exist a legal clock routing solution; (2) there should not exist any logic resource overflows, that is, we should be able to legalize all the cells with relatively small displacement. Given the objective and constraints, we formally define the *clock network planning problem* as follows.

Problem 5 (Clock Network Planning) *Given an optimized FPGA placement, find a movement-minimized cell-to-clock region assignment without logic resource overflow and a corresponding clock routing solution satisfying the target clocking architecture.*

Given the notations defined in Table 3.5, Problem 5 can be written as a binary optimization problem shown in Formulation (3.14). It optimizes over binary variables $x_{v,r}$ to minimize the objective (3.14a) of total cell movement. If cell v is assigned to clock region r, then $x_{v,r} = 1$, otherwise, $x_{v,r} = 0$. Constraint (3.14c) guarantees that each cell is assigned to exactly one clock region.

Table 3.5: Notations Used in Clock Network Planning

$\mathcal{V}$	The set of cells.
$\mathcal{S}$	The set of resource types, e.g., {LUT, FF, DSP, RAM}.
$\mathcal{V}^{(s)}$	The set of cells of resource type $s \in \mathcal{S}$.
$A_v^{(s)}$	The cell v's demand for resource type $s \in \mathcal{S}$.
$\mathcal{R}$	The set of clock regions.
$C_r^{(s)}$	The clock region r's capacity for resource type $s \in \mathcal{S}$.
$D_{v,r}$	The physical distance between cell v and clock region r.
$\mathcal{E}$	The set of clock nets.

Constraint (3.14d) ensures that the total demand of each resource type is no more than the corresponding capacity in each clock region. Constraint (3.14e) requires the existence of legal clock routing solutions with respect to the assignment $\boldsymbol{x}$. Here we do not list the closed-form expression of this constraint since it can be extremely complicated and impossible to be tackled in practice.

$$\min_{\boldsymbol{x}} \quad \sum_{v \in \mathcal{V}} \sum_{r \in \mathcal{R}} D_{v,r}\, x_{v,r}, \tag{3.14a}$$

$$\text{s.t.} \quad x_{v,r} \in \{0,1\}, \forall v \in \mathcal{V}, \forall r \in \mathcal{R}, \tag{3.14b}$$

$$\sum_{r \in \mathcal{R}} x_{v,r} = 1, \forall v \in \mathcal{V}, \tag{3.14c}$$

$$\sum_{v \in \mathcal{V}} A_v^{(s)}\, x_{v,r} \le C_r^{(s)}, \forall r \in \mathcal{R}, \forall s \in \mathcal{S}, \tag{3.14d}$$

$$\text{Exist a legal clock routing w.r.t } \boldsymbol{x}. \tag{3.14e}$$

3.4.2.3 Branch-and-Bound Method

A general algorithm to solve binary optimization problems, like Formulation (3.14), is the *branch-and-bound method* [36]. It is a tree traversal-based heuristic to search the very large solution space of possible variable assignments. Its basic idea can be illustrated by Figure 3.17. In this example, we are trying to find the optimal solution of a constrained minimization problem over a discrete (e.g., binary and integral) space. Each circle here denotes a solution and the color intensity indicates its optimality (in terms of minimizing the cost without considering feasibility). Three feasible solutions are denoted by stroked circles.

A general branch-and-bound algorithm starts with solving the unconstrained problem P_0 and its optimal solution typically is not feasible, as in this example. In this case, by imposing different constraints, a branching procedure divides the solution space into several sub-spaces (P_1 and P_2) and we continue to find the unconstrained optimal solution in each of these sub-spaces. The same branching procedure is progressively performed to each sub-problem until a feasible solution is found (e.g., the solution of P_3). Once a feasible solution is reached, we can treat its cost as an upper-bound objective value of the target minimization problem, and then, all the branches with lower-bound objective values larger than it can be safely pruned in the later exploration. Using Figure 3.17 as an example, if we found the first feasible solution in P_3 with the objective value of Φ, then: (1) there is no need to further branch P_3 since the optimal solution of P_3 is guaranteed to be no worse than any sub-problem of

171
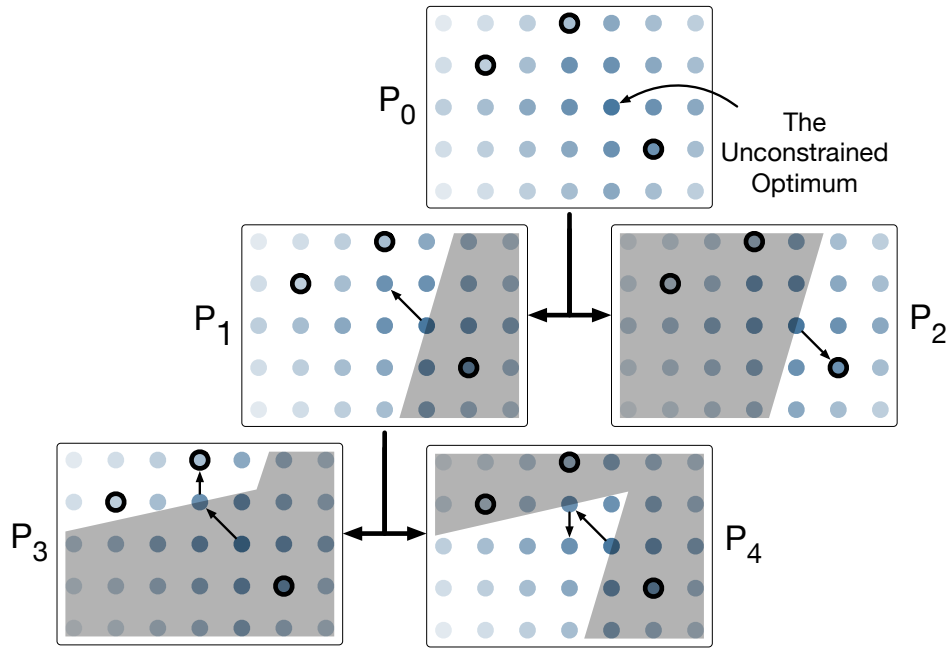
Figure 3.17: Illustration of the branch-and-bound method. Each circle denotes a solution and color intensity indicates its optimality. Feasible solutions are denoted by stroked circles.

P_3; (2) there is no need to explore branches (sub-spaces) with lower-bound objective values larger than Φ since solutions in them are guaranteed to be sub-optimal.

The branch-and-bound method can explore a sufficiently large solution space and find near-optimal solutions for various optimization problems within a limited amount of time. Our proposed clock network planning algorithm precisely borrows this idea. In the rest of this section, we will frequently link our proposed techniques to the concepts introduced in this section for better explanation.

3.4.2.4   The Clock Network Planning Algorithm

Algorithm 12 gives our proposed clock network planning algorithm to solve Formulation (3.14). Besides the inputs/outputs and notations described in Problem 5 and Table 3.5, an extra input parameter N is required for controlling the maximum number of feasible solutions to explore. We set N to 10 in our framework.

From line 1 to line 2, we initialize the best cell-to-clock region assignment $(\boldsymbol{x}^*)$, its cost (cost^*), and the corresponding clock routing solution (γ^*) as invalid and reset the number of feasible solutions n to 0. At line 3, we construct an initial clock-assignment constraint $\boldsymbol{\kappa}^{(0)}$. For a clock-assignment constraint $\boldsymbol{\kappa}$, each $\kappa_{e,r}$ is a binary value that indicates whether cells in clock net e can be assigned to clock region r. Similar to the branch-and-bound method starting with an unconstrained problem (Section 3.4.2.3), we do not consider clock feasibility at the beginning and allow any cell-to-clock region assignment. Therefore, all the entries in the initial clock-assignment constraint $\boldsymbol{\kappa}^{(0)}$ are set to 1 (line 3). To perform a tree traversal-based exploration like the branch-and-bound method, we maintain a stack to search the solution space in the depth-first search (DFS) order. The DFS starts with the constraint $\boldsymbol{\kappa}^{(0)}$ (line 4) and repeated from line 5 to line 21 until the stack becomes empty or enough number of feasible solutions are found ($\mathsf{n} = \mathsf{N}$). During the DFS, various clock-assignment constraints $\boldsymbol{\kappa}$ are branched from the constraint tree rooted at $\boldsymbol{\kappa}^{(0)}$, just like the branching procedure illustrated in Figure 3.17. The best solution found during this DFS exploration is returned at line 22 as

173

Algorithm 12: Clock Network Planning

    Input   : A placement with notations defined in Table 3.5. The maximum number of legal solutions N.

    Output: A movement-minimized cell-to-clock region assignment without logic resource overflow and a corresponding legal clock routing solution.

1 $(\mathsf{cost}^*, \boldsymbol{x}^*, \gamma^*) \leftarrow (+\infty, \mathsf{none}, \mathsf{none})$;

2 $\mathsf{n} \leftarrow 0$;

3 $\kappa_{e,r}^{(0)} \leftarrow 1, \forall e \in \mathcal{E}, \forall r \in \mathcal{R}$;

4 $\mathsf{stack.push}(\boldsymbol{\kappa}^{(0)})$;

5 **while** stack *is not empty* **and** $\mathsf{n} < \mathsf{N}$ **do**

6 $\boldsymbol{\kappa} \leftarrow \mathsf{stack.pop}()$;

7   Get the optimal cell-to-clock region assignment $\boldsymbol{x}^{(\boldsymbol{\kappa})}$ and its cost $\mathsf{cost}^{(\boldsymbol{\kappa})}$ under constraint $\boldsymbol{\kappa}$ (Section 3.4.2.5);

8   **if** *no feasible* $\boldsymbol{x}^{(\boldsymbol{\kappa})}$ *exists* **then continue**;

9   Get the clock routing solution $\gamma^{(\boldsymbol{\kappa})}$ corresponding to $\boldsymbol{x}^{(\boldsymbol{\kappa})}$ (Section 3.4.2.6);

10   **if** $\gamma^{(\boldsymbol{\kappa})}$ *is overflow-free* **then**

11 $\mathsf{n} \leftarrow \mathsf{n} + 1$;

12   **if** $\mathsf{cost}^{(\boldsymbol{\kappa})} < \mathsf{cost}^*$ **then**

13 $(\mathsf{cost}^*, \boldsymbol{x}^*, \gamma^*) \leftarrow (\mathsf{cost}^{(\boldsymbol{\kappa})}, \boldsymbol{x}^{(\boldsymbol{\kappa})}, \gamma^{(\boldsymbol{\kappa})})$;

14   **end**

15   **end**

16   **else if** $\gamma^{(\boldsymbol{\kappa})}$ *has routing overflow* **then**

17   Derive a set of more strict constraints K' from $\boldsymbol{\kappa}$ (Section 3.4.2.7);

18   Remove $\boldsymbol{\kappa}' \in K'$ that has lower-bound cost larger than cost^* (Section 3.4.2.8);

19   Push $\boldsymbol{\kappa}' \in K'$ into stack by their lower-bound costs from high to low;

20   **end**

21 **end**

22 **return** $(\mathsf{cost}^*, \boldsymbol{x}^*, \gamma^*)$;

the final result.

In each execution of line 5 to line 21, we first fetch the clock-assignment constraint $\boldsymbol{\kappa}$ on the top of the stack (line 6), then get the movement-minimized cell-to-clock region assignment constrained by logic resources and $\boldsymbol{\kappa}$ (line 7). This step can be interpreted as, within the sub-space $\boldsymbol{\kappa}$, finding the optimal solution $\boldsymbol{x}^{(\boldsymbol{\kappa})}$ of Formulation (3.14) without considering the clock constraint (3.14e). If no such $\boldsymbol{x}^{(\boldsymbol{\kappa})}$ exists, this branch will be discarded (line 8). Otherwise, we continue to evaluate the clock feasibility of $\boldsymbol{x}^{(\boldsymbol{\kappa})}$ by constructing a clock routing solution $\gamma^{(\boldsymbol{\kappa})}$ (line 9). If $\gamma^{(\boldsymbol{\kappa})}$ is routing overflow-free, $(\mathsf{cost}^{(\boldsymbol{\kappa})}, \boldsymbol{x}^{(\boldsymbol{\kappa})}, \gamma^{(\boldsymbol{\kappa})})$ then forms a feasible solution and we will update the best solution $(\mathsf{cost}^*, \boldsymbol{x}^*, \gamma^*)$ if needed (line 10 to line 15). If $\gamma^{(\boldsymbol{\kappa})}$ still has routing overflows, we will branch new clock-assignment constraints from $\boldsymbol{\kappa}$ to encourage more clock-friendly solutions (line 17). These new constraints $\boldsymbol{\kappa}' \in K'$ can be interpreted as sub-spaces of $\boldsymbol{\kappa}$ and some previously allowed clock assignments in $\boldsymbol{\kappa}$ can be blocked in $\boldsymbol{\kappa}' \in K'$. Among these newly derived constraints, we prune those that can only lead to sub-optimal solutions (line 18) and push the remaining into the stack in the descending order of their lower-bound costs (line 19). By doing so, we always first explore the branch with the minimum lower-bound cost at each constraint tree node.

The details of each core building block in Algorithm 12 will be further elaborated in the later sections. Section 3.4.2.5 describes the cell-to-clock region assignment at line 7. Section 3.4.2.6 presents the clock routing at line 9. The clock-assignment constraint derivation at line 17 and the lower-

175

bound cost calculation from line 18 to line 19 are detailed in Section 3.4.2.7 and Section 3.4.2.8, respectively.

3.4.2.5   Minimum-Cost Flow-Based Cell-to-Clock Region Assignment

The cell-to-clock region assignment (line 7 in Algorithm 12) essentially is solving the clock-unconstrained version of Formulation (3.14) within the sub-space of a given clock-assignment constraint $\boldsymbol{\kappa}$. It can be written as a binary optimization problem shown in Formulation (3.15), where $\mathcal{E}(v)$ denotes the set of clocks in cell v, binary value $\kappa_{e,r}$ indicates whether cells in clock net e can be assigned to clock region r, and other notations are inherited from Table 3.5. Note that Formulation (3.14) and Formulation (3.15) only differ by the clock constraints (3.14e) and (3.15e).

$$\min_{\boldsymbol{x}} \quad \sum_{v \in \mathcal{V}} \sum_{r \in \mathcal{R}} D_{v,r}\, x_{v,r}, \tag{3.15a}$$

$$\text{s.t.} \quad x_{v,r} \in \{0,1\}, \forall v \in \mathcal{V}, \forall r \in \mathcal{R}, \tag{3.15b}$$

$$\sum_{r \in \mathcal{R}} x_{v,r} = 1, \forall v \in \mathcal{V}, \tag{3.15c}$$

$$\sum_{v \in \mathcal{V}} A_v^{(s)}\, x_{v,r} \leq C_r^{(s)}, \forall r \in \mathcal{R}, \forall s \in \mathcal{S}, \tag{3.15d}$$

$$x_{v,r} = 0, \forall (v,r) \in \{v \in \mathcal{V}, r \in \mathcal{R} \mid \exists e \in \mathcal{E}(v) \text{ s.t. } \kappa_{e,r} = 0\}. \tag{3.15e}$$

The motivation of formulating Formulation (3.15) in this way is that it can be approximately transformed into a set of minimum-cost flow problems, each of which corresponds to a resource type (e.g., LUT, FF, DSP, and RAM). Since the minimum-cost flow is a well-studied problem, it can be efficiently solved by

176

many mature algorithms [4]. Figure 3.18 gives a graph representation of the minimum-cost flow corresponding to Formulation (3.15) with a single resource type. It is a bipartite graph (regardless of the super source S and the super target T) with vertices for cells $(v_1, v_2, \ldots, v_{|\mathcal{V}|})$ on the left and vertices for clock regions $(r_1, r_2, \ldots, r_{|\mathcal{V}|})$ on the right. We introduce an edge between each pair of cell and clock region, but set its capacity to 0 if the assignment is forbidden by the given constraint κ. With the edge cost and capacity settings shown in Figure 3.18, computing the minimum-cost flow of amount $\Sigma_{v \in \mathcal{V}} A_v^{(s)}$ on the graph can approximate the optimal solution of Formulation (3.15).
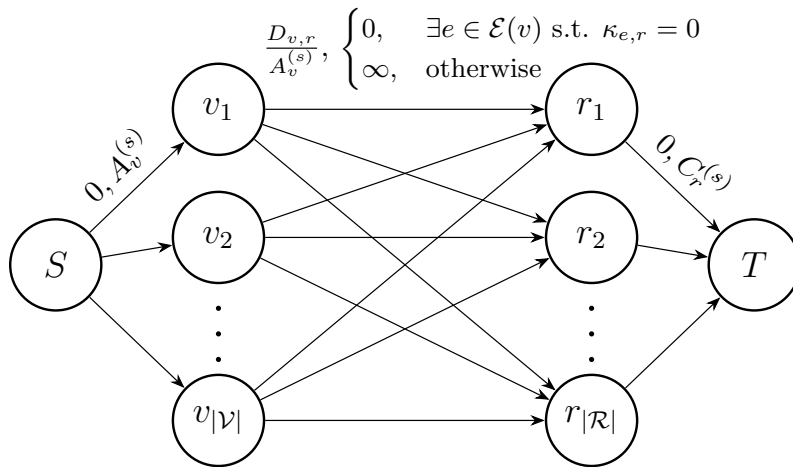
Figure 3.18: A graph representation of the minimum-cost flow for Formulation (3.15) with a single resource type. The pair of numbers on each edge denotes the unit flow cost and the flow capacity, respectively. For example, the edge between S and v_1 has a unit flow cost of 0 and a flow capacity of $A_{v_1}^{(s)}$.

The sub-optimality comes from the fact that, in a minimum-cost flow solution, a cell can be split and assigned to multiple clock regions. In such a case, we move all the "fragments" to the clock region containing the largest one

177

among them to realize an actual cell-to-clock region assignment. In practice, the splitting only occurs in a negligibly small portion of cells, thus the global optimality can still be largely retained[1]. It is worthwhile to mention that, if the logic resource demands of all cells for a given resource type s are the same (i.e., $A_i^{(s)} = A_j^{(s)}, \forall i, j \in \mathcal{V}$), the solution given by the minimum-cost flow is also optimal for Formulation (3.15). This case is applicable to resource types that only have one single cell type (e.g., DSP and CLB).

A minimum-cost flow solution, however, cannot always be realized as a complete cell-to-clock region assignment even without cell splitting. If the resulting flow amount is less than the amount of flow being pushed ($\Sigma_{v \in \mathcal{V}} A_v^{(s)}$), then not all the cells can be assigned without logic resource overflow. This can happen in scenarios where clock nets are over-constrained in too-small regions. In such a case, it is guaranteed that no feasible solutions exist in the sub-space defined by the given clock-assignment constraint $\boldsymbol{\kappa}$, and thus we can safely prune this branch as described at line 8 of Algorithm 12.

3.4.2.6 Clock Tree Construction

In this section, we will present the algorithm to construct a clock tree solution for a given cell-to-clock region assignment. As introduced in Section 3.2, a clock tree consists of a D-layer vertical trunk tree that connects all clock loads and an R-layer route that connects the D-layer trunk tree to the

---

[1]This post step might produce some negligible logic resource overflows. If the logic resource constraint needs to be rigorously honored, slightly tighter logic resource capacities can be applied to leave some margin for it.

clock source. Since the routing patterns on these two layers are very different, the routings on these two layers are conducted separately in our framework. Since R-layer routing relies on the D-layer trunk location, we perform D-layer routing first, then followed by R-layer routing.

3.4.2.6.1 Lagrangian Relaxation-Based D-Layer Clock Tree Construction

(a)   (b)   (c)
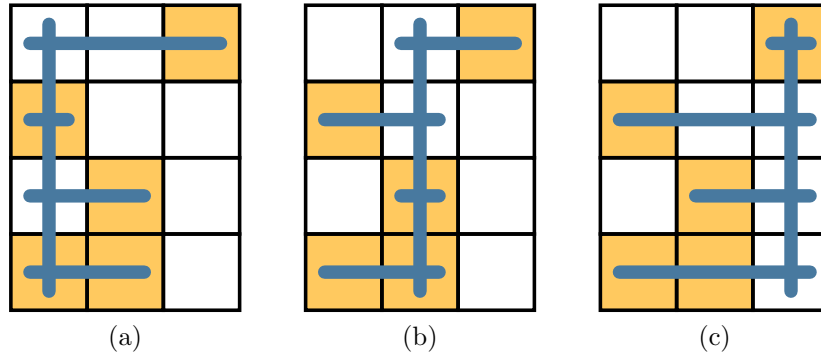
Figure 3.19: Three different D-layer clock tree topologies of the same clock load distribution on a 3×4 clock region grid. Each of them is a vertical trunk tree with horizontal branches connecting all the clock loads. Yellow shaded regions denote clock regions containing clock loads of the given clock net.

As shown in Figure 3.19, given a cell-to-clock region assignment on a clock region grid with m columns ($m = 3$ in Figure 3.19), we can generate m D-layer clock tree topologies for each clock by placing the vertical trunk in different columns. Our goal here is to select exactly one clock tree topology from the m candidates for each clock such that there is no VD/HD overflow. Meanwhile, a topology-dependent objective (e.g., resource usage, clock skew, insertion delay, etc.) also needs to be optimized.

179

If we denote the set of m clock tree candidates of clock e by $\mathcal{T}(e)$, denote the set of all clock tree candidates by $\mathcal{T}$ (i.e., $\mathcal{T} = \bigcup_{e \in \mathcal{E}} \mathcal{T}(e)$), denote the topology cost of clock tree candidate t by ϕ_t, and use binary values $H_{t,r}/V_{t,r}$ to represent whether clock tree candidate t occupies an HD/VD track in clock region r, then the D-layer clock tree construction problem can be mathematically written as a binary optimization problem shown in Formulation (3.16).

$$\min_{\boldsymbol{x}} \quad \sum_{t \in \mathcal{T}} \phi_t \, \boldsymbol{z}_t, \tag{3.16a}$$

$$\text{s.t.} \quad \boldsymbol{z}_t \in \{0, 1\}, \forall t \in \mathcal{T}, \tag{3.16b}$$

$$\sum_{t \in \mathcal{T}(e)} \boldsymbol{z}_t = 1, \forall e \in \mathcal{E}, \tag{3.16c}$$

$$\sum_{t \in \mathcal{T}} H_{t,r} \, \boldsymbol{z}_t \leq 24, \forall r \in \mathcal{R}, \tag{3.16d}$$

$$\sum_{t \in \mathcal{T}} V_{t,r} \, \boldsymbol{z}_t \leq 24, \forall r \in \mathcal{R}. \tag{3.16e}$$

Formulation (3.16) optimizes over binary variables $\boldsymbol{z}_t$ to minimize the objective of topology cost (3.16a). If the clock tree candidate t is selected in the routing solution, then $\boldsymbol{z}_t = 1$, otherwise, $\boldsymbol{z}_t = 0$. Constraint (3.16c) ensures that exactly one candidate is selected for each clock net. Constraints (3.16d) and (3.16e) bound the HD/VD clock routing usage in each clock region (24 is the number of available HD/VD tracks in each clock region of our target device as described in Section 3.2). In this work, since feasibility is the only consideration for clock networks, we simply set the topology cost ϕ_t as the total HD and VD demand of t. However, other metrics (e.g., clock skew) can

180

also be integrated in practice.

Although Formulation (3.16) can be optimally solved using integer linear programming techniques, they are too computationally expensive and unaffordable in our application. Therefore, we relax Formulation (3.16) to a much easier problem, as shown in Formulation (3.17). Here, we remove the two clock resource constraints (3.16d) and (3.16e) and add a set of Lagrangian multipliers [22] λ_t in the objective (3.17a). Each λ_t can be interpreted as the routing-overflow penalty applied to the clock tree candidate t and we assign a larger value to it if t is likely to run through congested regions. Then, by properly updating these λ_t and iteratively solving Formulation (3.17), overflow-free or overflow-minimized clock routing solutions can be achieved.

$$\min_{\boldsymbol{x}} \qquad \sum_{t \in \mathcal{T}} (\phi_t + \lambda_t)\, \boldsymbol{z}_t, \qquad (3.17a)$$

$$\text{s.t.} \qquad \boldsymbol{z}_t \in \{0, 1\}, \forall t \in \mathcal{T}, \qquad (3.17b)$$

$$\sum_{t \in \mathcal{T}(e)} \boldsymbol{z}_t = 1, \forall e \in \mathcal{E}. \qquad (3.17c)$$

Algorithm 13 summarizes our Lagrangian relaxation-based D-layer clock tree construction. From line 1 to line 2, we create all the clock tree candidates $\mathcal{T}$ and initialize their penalties $\boldsymbol{\lambda}^{(0)}$ to 0. In each Lagrangian iteration (line 4 to line 8), we first get the optimal solution $\boldsymbol{z}^{(i)}$ of Formulation (3.17) with $\boldsymbol{\lambda}^{(i)}$ (line 5). Then, $\boldsymbol{\lambda}^{(i+1)}$ can be derived from $\boldsymbol{\lambda}^{(i)}$ by penalizing clock tree candidates that run through overflowed clock regions in the routing solution

Algorithm 13: Lagrangian Relaxation-Based D-layer Clock Tree Construction

> **Input** : A cell-to-clock region assignment $\boldsymbol{x}$. The maximum number of Lagrangian iterations $\mathsf{I}_{\max}$ (default is 20).
>
> **Output :** A D-layer routing solution with minimized routing overflow and topology cost.

1 Create all clock tree candidates $\mathcal{T}$ for $\boldsymbol{x}$ (Figure 3.19);

2 $\lambda_t^{(0)} \leftarrow 0, \forall t \in \mathcal{T}$;

3 $\mathsf{i} \leftarrow 0$;

4 do

5 $\quad$ $\boldsymbol{z}^{(i)} \leftarrow \texttt{solveLR}(\boldsymbol{\lambda}^{(i)})$ // solve Formulation (3.17)

6 $\quad$ $\boldsymbol{\lambda}^{(i+1)} \leftarrow \texttt{updateLR}(\boldsymbol{z}^{(i)}, \boldsymbol{\lambda}^{(i)})$ // update $\boldsymbol{\lambda}$

7 $\quad$ $\mathsf{i} \leftarrow \mathsf{i} + 1$;

8 while $\boldsymbol{z}^{(i)}$ *has overflow* **and** $\boldsymbol{\lambda}^{(i)} \neq \boldsymbol{\lambda}^{(i-1)}$ **and** $\mathsf{i} < \mathsf{I}_{\max}$;

9 $\boldsymbol{z}^* \leftarrow$ the $\boldsymbol{z}^{(i)}$ with the minimum overflow;

10 return $\{t \in \mathcal{T} \mid \boldsymbol{z}_t^* = 1\}$;

11 Function $\texttt{solveLR}(\boldsymbol{\lambda})$:

12 $\quad$ $\boldsymbol{z}_t \leftarrow 0, \forall t \in \mathcal{T}$;

13 $\quad$ **foreach** $e \in \mathcal{E}$ **do**

14 $\quad\quad$ $t^* \leftarrow$ the $t \in \mathcal{T}(e)$ with the minimum $\phi_t + \lambda_t$;

15 $\quad\quad$ $z_{t^*} \leftarrow 1$;

16 $\quad$ **return** $\boldsymbol{z}$;

17 Function $\texttt{updateLR}(\boldsymbol{z}^{(i)}, \boldsymbol{\lambda}^{(i)})$:

18 $\quad$ $\Delta\lambda_t \leftarrow 0, \forall t \in \mathcal{T}$;

19 $\quad$ **foreach** $r \in \mathcal{R}$ *with HD/VD overflows of* O_H/O_V **do**

20 $\quad\quad$ $\mathcal{T}_H(r) \leftarrow \{t \in \mathcal{T} \mid H_{t,r} = 1\}$;

21 $\quad\quad$ $\mathcal{T}_V(r) \leftarrow \{t \in \mathcal{T} \mid V_{t,r} = 1\}$;

22 $\quad\quad$ **foreach** $t \in \mathcal{T}_H(r)$ **do** $\Delta\lambda_t \leftarrow \Delta\lambda_t + \frac{O_H}{|\mathcal{T}_H(r)|}$;

23 $\quad\quad$ **foreach** $t \in \mathcal{T}_V(r)$ **do** $\Delta\lambda_t \leftarrow \Delta\lambda_t + \frac{O_V}{|\mathcal{T}_V(r)|}$;

24 $\quad$ $\alpha \leftarrow \infty$;

25 $\quad$ **foreach** $e \in \mathcal{E}$ **do**

26 $\quad\quad$ $t^* \leftarrow$ the $t \in \mathcal{T}(e)$ being selected in iteration i;

27 $\quad\quad$ **foreach** $t \in \mathcal{T}(e)$ *that has* $\Delta\lambda_t < \Delta\lambda_{t^*}$ **do**

28 $\quad\quad\quad$ $\alpha \leftarrow \texttt{min}(\alpha, \frac{(\phi_t + \lambda_t) - (\phi_{t^*} + \lambda_{t^*})}{\Delta\lambda_{t^*} - \Delta\lambda_t})$;

29 $\quad$ **if** $\alpha = \infty$ **then return** $\boldsymbol{\lambda}^{(i)}$;

30 $\quad$ $\lambda_t^{(i+1)} \leftarrow \lambda_t^{(i)} + (1 + \delta)\,\alpha\,\Delta\lambda_t, \forall t \in \mathcal{T}$;

31 $\quad$ **return** $\boldsymbol{\lambda}^{(i+1)}$;

given by $\boldsymbol{z}^{(\mathrm{i})}$ (line 6). This iteration is repeated until one of the following holds: (1) a routing overflow-free solution is found; (2) $\boldsymbol{\lambda}$ does not change anymore; (3) the maximum iteration count $\mathsf{I}_{\mathrm{max}}$ is reached; Finally, from line 9 to line 10, we backtrace all the explored solutions and return the one with the minimum routing overflows as the final result.

The optimal solution of Formulation (3.17), as given in the function `solveLR` (line 11 to line 16), can be efficiently obtained by picking the candidate with the minimum $\phi_t + \lambda_t$ from each candidate pool $\mathcal{T}(e)$. The function `updateLR` (line 17 to line 31) presents our $\boldsymbol{\lambda}$ updating scheme. From line 18 to line 23, we first calculate the base penalty $\Delta\lambda_t$ for each candidate t. As shown at line 22 and line 23, for an overflowed clock region, we treat its overflow value (O_H/O_V) as the total amount of penalty and evenly distribute the penalty to all the candidates running through it. After that, from line 24 to line 28, we calculate the minimum scaling factor α that can change the optimal solution of Formulation (3.17) with $\alpha\,\Delta\boldsymbol{\lambda}$ being added to current $\boldsymbol{\lambda}$. If such an α does not exist, $\boldsymbol{\lambda}$ are kept unchanged (line 29). Otherwise, we add the extra penalty $(1+\delta)\,\alpha\,\Delta\boldsymbol{\lambda}$ to $\boldsymbol{\lambda}^{(\mathrm{i})}$ ($\delta \ll 1$ for tie-breaking) and return the result as $\boldsymbol{\lambda}^{(\mathrm{i}+1)}$ (line 30 to line 31).

3.4.2.6.2 A* Search-Based R-Layer Clock Tree Routing

The R-layer routing is responsible for connecting the clock source to the D-layer trunk tree. Given a D-layer clock routing solution, the R-layer routing is very similar to the conventional 2-pin net global routing problem. The

only difference is that, in each of these 2-pin nets, one of the two "pins" is a vertical trunk (Section 3.2) instead of a single terminal. Therefore, we extend the conventional A* search [26]-based routing algorithm to treat all the clock regions occupied by the D-layer trunk as legal endpoints. Besides, a rip-up and reroute technique similar to [53] is also applied to iteratively resolve routing overflows.

3.4.2.7 Clock-Assignment Constraint Derivation

Recall that, at line 17 of Algorithm 12, for a given clock-assignment constraint κ, if an overflow-free clock routing solution cannot be found, we will derive a set of new constraints from κ to encourage more clock-friendly solutions. In this section, we will detail this clock-assignment constraint derivation process. Since, in practice, R-layer routing is much less congested than D-layer routing and rarely fails, we will only discuss constraint derivation methods for resolving D-layer congestions. However, similar ideas are also applicable to R-layer routing.

3.4.2.7.1 Constraint Derivation for VD Overflows

Algorithm 14 summarizes our constraint deviation scheme for resolving VD overflows. We first get the clock region r with the most VD overflow (line 1). Then, for each clock that occupies VD resources in r, we generate placement blockages in four directions, as shown in Fig 3.20, that can potentially alleviate the congestion in r. Finally, we impose each of these blockages on top of the

184
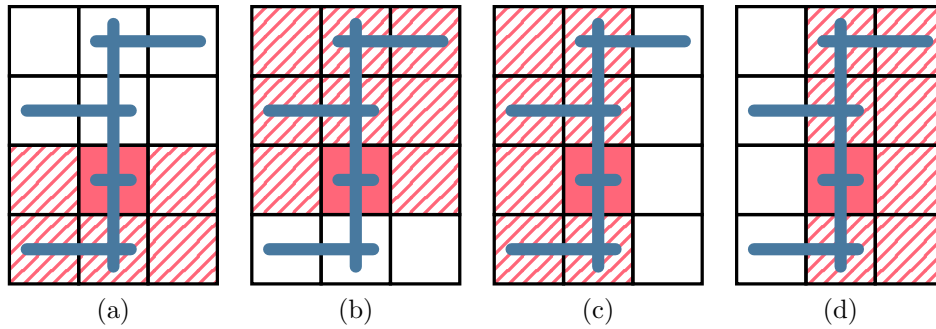
Figure 3.20: Four half plane-based clock-assignment blockages (hatched/solid red regions) that are in the (a) south, (b) north, (c) west, and (d) east of a VD overflowed clock region (solid red region).

current clock-assignment constraint κ to form a set of new constraints K' (line 3 to line 9). If there are q clock nets occupying VD resources in r, there will be $4q$ new constraints in K' and each $\kappa' \in K'$ represent a sub-space of κ as described in Section 3.4.2.4.

3.4.2.7.2 Constraint Derivation for HD Overflows

Our constraint derivation for HD overflow is similar to that for VD, as described in Section 3.4.2.7.1 and Algorithm 14. However, given the fact that HD branches affect the tree topology much more locally than VD trunks, blockages of granularities finer than Figure 3.20 might be able to achieve even better results. For example, the corner-based (Figure 3.21) and the row-based (Figure 3.22) blockages can potentially resolve the overflow with less cell movement compared with the blockages shown in Figure 3.20. Surprisingly, as will be shown in Section 3.4.3.3, the blockage schemes in Figure 3.21 and Figure 3.22 cannot outperform that in Figure 3.20 within a limited amount of

185

Algorithm 14: CLOCK-ASSIGNMENT CONSTRAINT DERIVA-
TION FOR VD OVERFLOWS

Input : A clock-assignment constraint $\boldsymbol{\kappa}$ and its clock routing
solution γ.

Output: A set of new clock-assignment constraints K' derived
from $\boldsymbol{\kappa}$ that can potentially alleviate the VD overflow.

1 $r \leftarrow$ the clock region with the most VD overflow in γ;
2 $K' \leftarrow \varnothing$;
3 **foreach** $e \in \mathcal{E}$ *that occupies VD resources in* r **do**
4   **foreach** *blockage B in Figure 3.20* **do**
5         $\boldsymbol{\kappa}' \leftarrow \boldsymbol{\kappa}$;
6         $\boldsymbol{\kappa}'_{e,b} \leftarrow 0, \forall b \in B$;
7         $K' \leftarrow K' \cup \boldsymbol{\kappa}'$;
8   **end**
9 **end**
10 **return** K';

time in our experiments. This might be because the blockages in Figure 3.21
and Figure 3.22 tend to cut the placeable region of each clock into non-convex
and unconnected fragments, which significantly slow down the convergence of
Algorithm 12. While using the blockages in Figure 3.20, the placeable region
of each clock is guaranteed to be a rectangle.

It is still an open problem to find the best constraint derivation scheme,
but we can see that our framework is generic and any other constraint deriva-
tion methods can also be easily integrated.

3.4.2.8 Lower-Bound Cost Calculation

In this section, we introduce a method to calculate the lower-bound
cost of Formulation (3.15) for a given clock-assignment constraint. This lower-

186
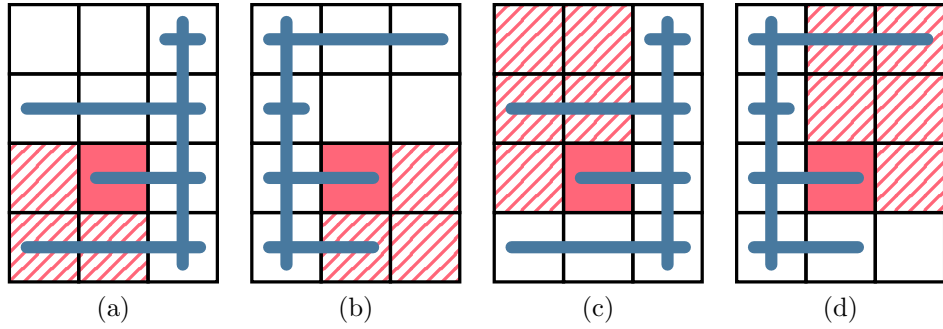
Figure 3.21: Corner-based clock-assignment blockages for an HD overflowed clock region.
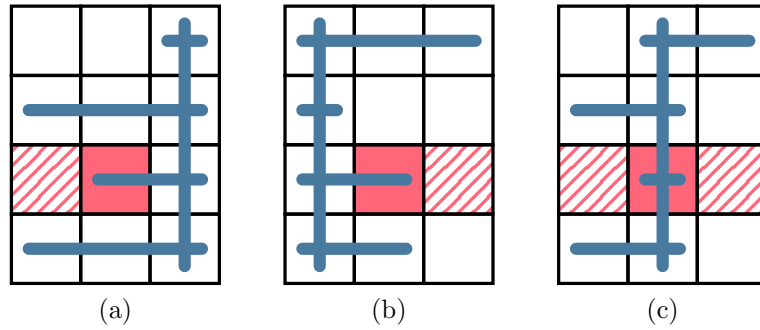
Figure 3.22: Row-based clock-assignment blockages for an HD overflowed clock region.

bound cost is essential for: (1) the solution pruning at line 18 of Algorithm 12; (2) the constraint sorting at line 19 of Algorithm 12. A good lower-bound cost should be: (1) as tight as possible to reflect the actual cost; (2) cheap to calculate, as it will be evaluated much more frequently than the actual cost calculation (solving Formulation (3.15)). Given these requirements, we propose the following lower-bound cost for a given clock-assignment constraint κ:

$$\text{cost}_{\text{LB}}(\kappa) = \sum_{v \in \mathcal{V}} \min_{\{r \in \mathcal{R} | \kappa_{e,r}=1, \forall e \in \mathcal{E}(v)\}} D_{v,r}, \tag{3.18}$$

187

where $\mathcal{E}(v)$ denotes the set of clock nets incident to cell v. Equation (3.18) can be proved as a lower-bound cost of Formulation (3.15) since it is the optimal solution of Formulation (3.15) without considering the logic resource constraint (3.15d). Moreover, its time complexity is only linear to the number of cells in the design, which is very computationally efficient.

3.4.3 Experimental Results

We implement UTPlaceF 2.X in C++ based on our UTPlaceF-DL framework described in Section 2.4 and perform the experiments on a Linux machine running with Intel Core i9-7900X CPUs (3.30 GHz and 10 cores) and 128 GB RAM. The ISPD 2017 clock-aware FPGA placement contest benchmark suite [80] released by Xilinx is used to demonstrate the effectiveness of the proposed approach. Routed wirelength reported by Xilinx Vivado v2016.4 [72] is used to evaluate the placement quality.

Table 3.6 lists the characteristics of the ISPD 2017 benchmark suite. To further demonstrate the effectiveness of UTPlaceF 2.X, we also perform experiments under more strict clock constraints. Specifically, besides using all of the 24 clock routing tracks, we also conduct experiments that only utilize up to 12, 8, 7, 6, and 5 clock routing tracks in each clock region. In the rest of this section, they will be denoted as "Clock Capacity (CC) = 24/12/8/7/6/5".

The wirelength optimization kernels (global/detailed placement and legalization) of our placer are carefully parallelized. Therefore, we enabled 10 threads for them in our experiments, but the techniques proposed in UT-

Table 3.6: ISPD 2017 Contest Benchmarks Statistics

Designs	#LUT	#FF	#RAM	#DSP	#Clock
CLK-FPGA01	211K	324K	164	75	32
CLK-FPGA02	230K	280K	236	112	35
CLK-FPGA03	410K	481K	850	395	57
CLK-FPGA04	309K	372K	467	224	44
CLK-FPGA05	393K	469K	798	150	56
CLK-FPGA06	425K	511K	872	420	58
CLK-FPGA07	254K	309K	313	149	38
CLK-FPGA08	212K	257K	161	75	32
CLK-FPGA09	231K	358K	236	112	35
CLK-FPGA10	327K	506K	542	255	47
CLK-FPGA11	300K	468K	454	224	44
CLK-FPGA12	277K	430K	389	187	41
CLK-FPGA13	339K	405K	570	262	47
Resources	538K	1075K	1728	768	-

PlaceF 2.X are all executed with only a single thread.

3.4.3.1 Comparison with Other State-of-the-Art Placers

Table 3.7 compares our results with other state-of-the-art academic clock-aware placers, including UTPlaceF 2.0 [42], NTUfplace [34], RippleF-PGA [64], and the clock-aware version of UTPlaceF-DL [44]. Among them, UTPlaceF 2.0, NTUfplace, and RippleFPGA are extensions of the top-3 winners of the ISPD 2017 clock-aware placement contest [80]. Metrics "WL" and "RT" represent the routed wirelength and runtime, while "WLR" and "RTR" represent the wirelength and runtime ratios normalized to UTPlaceF 2.X.

All the results of other placers are from their original publications. Since we are not able to get all their results under different CC values, this comparison is only based on the default clock capacity CC = 24. In this

comparison, UTPlaceF 2.0, NTUfplace, and RippleFPGA are single-threaded, while UTPlaceF-DL and our placer are executed with 16 and 10 threads, respectively. The runtime result of NTUfplace in Table 3.7 is the total runtime of placement and routing since the original publication did not report the placement runtime alone.

Due to the differences in machines, experiment setups, and baseline placement algorithms, Table 3.7 is not an apple-to-apple comparison from the clock legalization perspective. However, we still can see that UTPlaceF 2.X achieves the best overall routed wirelength with very competitive efficiency.

3.4.3.2 Comparison with a State-of-the-Art Method

To further demonstrate the effectiveness of UTPlaceF 2.X in a fair way, we implement the clock-related parts in our UTPlaceF 2.0 described in Section 3.3 (it is also the 1st-place winner of the ISPD 2017 clock-aware placement contest) and replaced the counterparts in UTPlaceF 2.X with them. This new placer is denoted as UTPlaceF 2.0-Impl. Since UTPlaceF 2.0-Impl and UTPlaceF 2.X only differ by the clock legalization approaches, the noises from parts that are irrelevant to this work (e.g., global/detailed placement) can be completely decoupled.

Table 3.7: Routed Wirelength (WL in 10^3) and Runtime (RT in seconds) Comparison with Other State-of-the-Art Placers (CC = 24)

Designs	UTPlaceF 2.0 [42]				NTUfplace [34]				RippleFPGA [64]				UTPlaceF-DL [44]				UTPlaceF 2.X			
	WL	RT	WLR	RTR	WL	RT†	WLR	RTR†	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR
CLK-FPGA01	2208	422	1.056	2.34	2098	3524	1.003	19.58	2011	288	0.962	1.60	2101	476	1.004	2.64	2092	180	1.000	1.00
CLK-FPGA02	2279	407	1.039	2.27	2173	3351	0.991	18.72	2168	266	0.988	1.49	2263	454	1.032	2.54	2194	179	1.000	1.00
CLK-FPGA03	5353	824	1.048	2.40	5049	6722	0.988	19.60	5265	583	1.031	1.70	5181	930	1.014	2.71	5109	343	1.000	1.00
CLK-FPGA04	3698	564	1.027	2.33	3710	5101	1.030	21.08	3607	380	1.002	1.57	3654	656	1.015	2.71	3600	242	1.000	1.00
CLK-FPGA05	4692	744	1.030	2.30	4523	6336	0.993	19.62	4660	569	1.023	1.76	4589	846	1.007	2.62	4556	323	1.000	1.00
CLK-FPGA06	5589	845	1.029	2.44	5169	7932	0.952	22.93	5737	591	1.056	1.71	5375	963	0.989	2.78	5432	346	1.000	1.00
CLK-FPGA07	2445	670	1.052	3.33	2380	4071	1.024	20.25	2326	304	1.001	1.51	2448	515	1.053	2.56	2324	201	1.000	1.00
CLK-FPGA08	1886	419	1.044	2.48	1843	3109	1.020	18.40	1778	247	0.984	1.46	1829	436	1.012	2.58	1807	169	1.000	1.00
CLK-FPGA09	2597	668	1.036	3.39	2499	4423	0.997	22.45	2530	327	1.009	1.66	2556	523	1.019	2.66	2507	197	1.000	1.00
CLK-FPGA10	4464	772	1.056	2.70	4294	6569	1.015	22.97	4496	512	1.063	1.79	4255	801	1.006	2.80	4229	286	1.000	1.00
CLK-FPGA11	4184	847	1.063	3.20	4031	6538	1.024	24.67	4190	455	1.064	1.72	4014	679	1.020	2.56	3936	265	1.000	1.00
CLK-FPGA12	3369	614	1.041	2.49	3244	5300	1.002	21.46	3388	409	1.047	1.66	3253	647	1.005	2.62	3236	247	1.000	1.00
CLK-FPGA13	3848	929	1.033	3.44	3818	5639	1.025	20.89	3833	441	1.029	1.63	3731	743	1.002	2.75	3723	270	1.000	1.00
Norm.	-	-	1.043	2.70	-	-	1.005	20.97	-	-	1.020	1.64	-	-	1.014	2.66	-	-	1.000	1.00

†: [34] only reports the total runtime of placement and routing, so the RT and RTR here are just for reference.

Table 3.8 compares UTPlaceF 2.X with UTPlaceF 2.0-Impl. It can be seen that UTPlaceF 2.X achieves similar results compared with UTPlaceF 2.0-Impl under relatively loose clock constraints (CC = 24/12). As the clock capacity reduces, UTPlaceF 2.X starts to outperform UTPlaceF 2.0-Impl in both routed wirelength and runtime. On average, with CC = 8/7/6/5, UTPlaceF 2.0-Impl suffers 1.1%/2.3%/13.1%/30.4% wirelength degradation, while runs 1.21×/1.40×/1.67×/2.19× slower compared with the baseline (UTPlaceF 2.X with CC = 24). However, with UTPlaceF 2.X, the routed wirelength only degrades by 0.6%/1.2%/2.3%/17.6% with only 7%/10%/17%/24% runtime increase. Furthermore, in the extremely challenging case of CC = 5, UTPlaceF 2.X can find feasible solutions for 9 out of 13 designs within the runtime limit of 1800 seconds, while only 6 designs can be successfully placed using UTPlaceF 2.0-Impl. Therefore, UTPlaceF 2.X is especially effective for cases with high clock utilization.

Table 3.8: Normalized Wirelength and Runtime Comparison between UTPlaceF 2.0-Impl (2.0-Impl) and UT-PlaceF 2.X (2.X) Under Different Clock Capacities (CC)

Designs	CC = 24 2.0-Impl WLR	RTR	2.X WLR	RTR	CC = 12 2.0-Impl WLR	RTR	2.X WLR	RTR	CC = 8 2.0-Impl WLR	RTR	2.X WLR	RTR	CC = 7 2.0-Impl WLR	RTR	2.X WLR	RTR	CC = 6 2.0-Impl WLR	RTR	2.X WLR	RTR	CC = 5 2.0-Impl WLR	RTR	2.X WLR	RTR
CLK-FPGA01	1.002	0.99	1.000	1.00	1.003	1.01	1.003	1.03	1.013	1.28	1.011	1.09	1.011	1.40	1.019	1.13	1.024	1.45	1.029	1.19	1.066	1.67	1.031	1.18
CLK-FPGA02	1.000	0.99	1.000	1.00	1.000	1.01	1.000	0.99	1.005	1.05	1.004	1.03	1.015	1.26	1.012	1.06	1.211	1.49	1.024	1.13	1.299	2.25	1.061	1.14
CLK-FPGA03	1.001	1.00	1.000	1.00	1.002	1.06	1.001	1.04	1.022	1.18	1.008	1.12	1.043	1.59	1.013	1.10	1.444	2.22	1.023	1.23	*	-	*	-
CLK-FPGA04	0.999	0.99	1.000	1.00	1.001	1.02	1.000	1.03	1.013	1.15	1.005	1.07	1.014	1.17	1.009	1.12	1.056	1.74	1.013	1.13	1.563	3.33	1.175	1.24
CLK-FPGA05	1.000	0.99	1.000	1.00	1.000	1.05	1.001	1.04	1.007	1.33	1.004	1.10	1.021	1.52	1.009	1.13	1.059	1.70	1.032	1.24	*	-	*	-
CLK-FPGA06	1.000	1.02	1.000	1.00	1.004	1.09	1.000	1.07	1.026	1.62	1.009	1.15	1.031	1.67	1.016	1.17	1.181	2.02	1.024	1.27	*	-	*	-
CLK-FPGA07	1.001	1.00	1.000	1.00	0.999	1.03	0.999	1.04	1.014	1.28	1.009	1.07	1.058	1.43	1.015	1.14	1.354	1.70	1.046	1.20	1.417	2.56	*	-
CLK-FPGA08	1.000	0.96	1.000	1.00	1.001	0.98	1.001	1.02	1.016	1.17	1.010	1.05	1.024	1.33	1.017	1.07	1.086	1.50	1.021	1.13	1.121	1.69	1.052	1.14
CLK-FPGA09	0.998	1.00	1.000	1.00	1.000	1.03	1.000	1.01	1.002	1.07	1.003	1.02	1.011	1.24	1.008	1.06	1.013	1.42	1.011	1.11	1.358	1.62	1.045	1.19
CLK-FPGA10	0.999	1.00	1.000	1.00	1.000	1.05	0.999	1.04	1.012	1.42	1.005	1.05	1.015	1.58	1.009	1.11	1.040	1.46	1.019	1.17	*	-	1.323	1.25
CLK-FPGA11	0.999	0.99	1.000	1.00	0.999	1.02	1.000	1.03	1.006	1.05	1.003	1.05	1.017	1.46	1.011	1.08	1.030	1.73	1.017	1.13	*	-	1.511	1.56
CLK-FPGA12	0.999	0.99	1.000	1.00	0.999	1.00	1.000	1.01	1.007	1.07	1.003	1.02	1.014	1.14	1.009	1.08	1.019	1.68	1.017	1.11	*	-	1.321	1.19
CLK-FPGA13	1.000	0.99	1.000	1.00	1.000	1.00	1.001	1.02	1.006	1.12	1.004	1.02	1.028	1.37	1.013	1.05	1.192	1.65	1.026	1.12	*	-	1.070	1.20
Norm.	1.000	0.99	1.000	1.00	1.001	1.03	1.000	1.03	1.011	1.21	1.006	1.07	1.023	1.40	1.012	1.10	1.131	1.67	1.023	1.17	1.304	2.19	1.176	1.24

*: Fail to find feasible placement solutions within 1800 seconds.

193

3.4.3.3 Comparison of Different Clock-Assignment Blockage Schemes

Table 3.9 compares the three clock-assignment blockage schemes described in Section 3.4.2.7: (1) half plane-based blockages (Figure 3.20); (2) corner-based blockages (Figure 3.21); (3) row-based blockages (Figure 3.22). Surprisingly, more fine-grained blockage schemes lead to worse quality and runtime in our experiments. In the very challenging case of $CC = 6$, the half plane-based scheme outperforms the corner-based and the row-based schemes by 9.0% and 24.4%, respectively, in routed wirelength. Meanwhile, it also runs $1.08\times$ and $1.33\times$ faster than them. Moreover, the corner-based and the row-based schemes failed to find feasible solutions for 1 and 4 designs, respectively, within the runtime limit of 1800 seconds, while the half plane-based scheme can achieve feasible solutions for all 13 designs.

We observe that the two fine-grained schemes often split the feasible region of each clock into separated and non-convex fragments, which can significantly worsen the convergence of the algorithm and result in poor solution quality or even infeasible solutions within a limited amount of time. It is still an open problem to find blockage schemes better than the half-plane one (Figure 3.20), but preserving the continuity and convexity of feasible regions should play a very important role here.

Table 3.9: Normalized Wirelength and Runtime Comparison of Different Clock-Assignment Blockage Schemes

Designs	CC = 24						CC = 12						CC = 8						CC = 6					
	Half Plane		Corner		Row		Half Plane		Corner		Row		Half Plane		Corner		Row		Half Plane		Corner		Row	
	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR
CLK-FPGA01	1.000	1.00	1.000	0.99	1.000	1.00	1.003	1.03	1.002	1.07	1.003	1.05	1.011	1.09	1.009	1.14	1.019	1.18	1.029	1.19	1.020	1.18	1.205	1.31
CLK-FPGA02	1.000	1.00	1.000	1.01	1.000	1.00	1.000	0.99	1.001	1.01	1.001	1.00	1.004	1.03	1.002	1.05	1.002	1.04	1.024	1.13	1.261	1.28	1.162	1.26
CLK-FPGA03	1.000	1.00	1.000	1.03	1.000	1.01	1.001	1.04	1.001	1.14	1.000	1.20	1.008	1.12	1.004	1.18	1.007	1.30	1.023	1.23	1.090	1.41	*	-
CLK-FPGA04	1.000	1.00	1.000	1.00	1.000	1.00	1.000	1.03	1.000	1.06	1.000	1.07	1.005	1.07	1.004	1.10	1.006	1.13	1.013	1.13	1.047	1.15	1.271	1.52
CLK-FPGA05	1.000	1.00	1.000	1.00	1.000	1.00	1.001	1.04	1.000	1.08	1.000	1.14	1.004	1.10	1.003	1.15	1.006	1.18	1.032	1.24	1.134	1.34	*	-
CLK-FPGA06	1.000	1.00	1.000	1.02	1.000	1.01	1.000	1.07	1.000	1.12	1.001	1.29	1.009	1.15	1.006	1.20	1.033	1.50	1.024	1.27	*	-	*	-
CLK-FPGA07	1.000	1.00	1.000	1.00	1.000	1.00	0.999	1.04	0.999	1.03	1.000	1.03	1.009	1.07	1.018	1.14	1.019	1.16	1.046	1.20	1.260	1.24	1.380	1.70
CLK-FPGA08	1.000	1.00	1.000	1.00	1.000	1.00	1.001	1.02	1.001	0.99	1.000	1.03	1.010	1.05	1.014	1.05	1.025	1.10	1.021	1.13	1.028	1.18	1.101	1.27
CLK-FPGA09	1.000	1.00	1.000	1.00	1.000	1.01	1.000	1.01	1.000	1.03	1.000	1.03	1.003	1.02	1.000	1.06	1.002	1.08	1.011	1.11	1.044	1.20	1.066	1.26
CLK-FPGA10	1.000	1.00	1.000	1.01	1.000	1.00	0.999	1.04	1.002	1.08	1.000	1.09	1.005	1.05	1.025	1.23	1.038	1.35	1.019	1.17	1.080	1.40	1.749	2.68
CLK-FPGA11	1.000	1.00	1.000	1.00	1.000	1.00	1.000	1.03	1.000	1.06	1.000	1.07	1.003	1.05	1.001	1.10	1.001	1.13	1.017	1.13	1.041	1.21	1.333	1.77
CLK-FPGA12	1.000	1.00	1.000	0.99	1.000	1.00	1.000	1.01	1.001	1.00	1.000	1.03	1.003	1.02	1.013	1.14	1.014	1.14	1.017	1.11	1.213	1.28	1.138	1.28
CLK-FPGA13	1.000	1.00	1.000	1.00	1.000	0.99	1.001	1.02	1.000	1.03	1.000	1.04	1.004	1.02	1.013	1.11	1.013	1.07	1.026	1.12	1.146	1.28	*	-
Norm.	1.000	1.00	1.000	1.00	1.000	1.00	1.000	1.03	1.000	1.05	1.000	1.08	1.006	1.07	1.009	1.13	1.014	1.18	1.023	1.17	1.113	1.26	1.267	1.56

∗: Fail to find feasible placement solutions within 1800 seconds.

195

3.4.3.4 Branch-and-Bound Tree Exploration

Figure 3.23 visualizes the lower-bound costs (Equation (3.18)) and the actual costs (Formulation (3.15)) of the first 30 feasible solutions found in a branch-and-bound tree exploration of `CLK-FPGA01` with CC = 6. We can observe that our lower-bound cost estimation is highly correlated with the actual cost. Therefore, even if we greedily pick the branch with the minimum lower-bound cost at each step, a relatively good solution can still be obtained, which is the first feasible solution shown in Figure 3.23. However, due to the non-convexity of the clock network planning problem, the first solution is generally not the optimum. In this example, there are 7 solutions (in the first 30 feasible solutions) having less actual costs than the first one (the solution obtained by the greedy approach). The best among them is achieved at number 27, which is about 4% better than the first solution.

3.4.4 Summary

In this section, we have proposed a generic FPGA placement framework, UTPlaceF 2.X, that simultaneously optimizes placement quality and ensures clock feasibility by explicit clock tree construction. The proposed UTPlaceF 2.X significantly reduces the placement quality degradation while honoring the clock feasibility for designs with high clock utilization. To realize UTPlaceF 2.X, a branch-and-bound-inspired clock network planning algorithm and a Lagrangian relaxation-based clock tree construction technique are proposed. Our experiments on the ISPD 2017 benchmark suite demonstrate
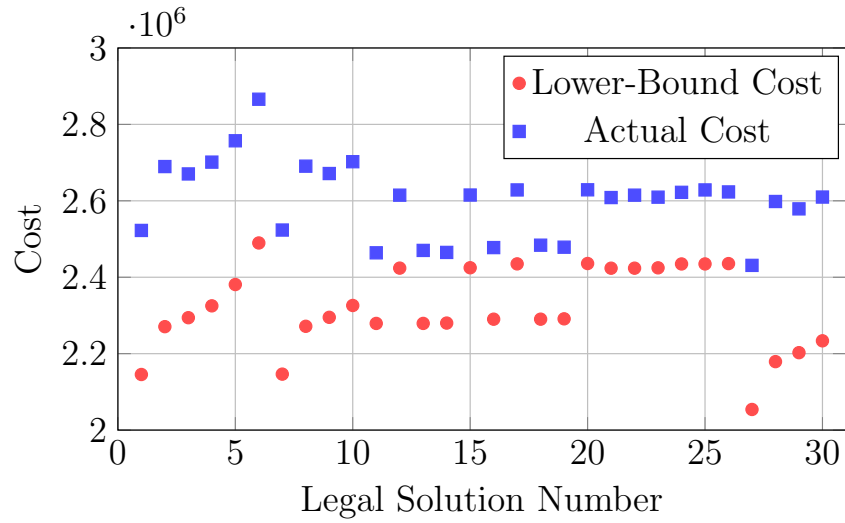
Figure 3.23: The lower-bound and actual costs of the first 30 feasible solutions found in a branch-and-bound procedure of `CLK-FPGA01` with CC = 6.

that UTPlaceF 2.X outperforms other state-of-the-art approaches in routed wirelength with competitive runtime.

Chapter 4

Parallel FPGA Placement Algorithms

4.1 Introduction

Placement is one of the most time-consuming optimization steps in modern FPGA physical implementation flow. In the past several decades, most of the research attention was focused on improving placement solution quality. However, ultra-fast and efficient placement core engines are now in great demand. Firstly, with the increasing capacity and complexity of modern FPGA devices, the gate count of state-of-the-art commercial FPGAs has reached the scale of millions [18,29]. Moreover, as a type of hardware accelerators, FPGAs need to be frequently reconfigured to adapt rapid and continuous changes from users in many scenarios, like datacenter-based cloud computing. Therefore, it is particularly desirable to develop a high-performance placement engine to reduce the FPGA implementation time. Besides, placement also has significant impacts on the quality of design mapping, hence, good

---

This chapter is based on the following publication.

1. Wuxi Li, Meng Li, Jiajun Wang, and David Z. Pan. "UTPlaceF 3.0: A parallelization framework for modern FPGA global placement." In Proceedings of the 36th International Conference on Computer-Aided Design, pp. 922-928. IEEE Press, 2017.

I am the main contributor in charge of problem formulation, algorithm development, and experimental validations.

performance-boosting techniques should still maintain competitive placement solution quality.

The scalability issue has constantly pushed the evolution of placement algorithms in the past few decades. In the early age of placement, simulated annealing-based placers, such as [7, 10, 11], dominated industry and academic research. Although the annealing technique worked well for small designs, as the capacity of FPGAs kept growing, it was no longer scalable for larger FPGA devices. Industry and academia then turned to minimum-cut-partitioning-based placers, e.g. [56], which performed well for designs with tens of thousands of gates. When the gate count of FPGAs arrived at the scale of millions, analytical placers, e.g., [14, 24, 25, 39, 50, 62, 63, 77, 78], started to steadily outperform annealing and minimum-cut approaches in both runtime and quality.

Despite the effectiveness and efficiency of analytical placers, their runtime still increases rapidly as the complexity and scale of FPGA devices has not stopped growing. One promising solution is leveraging today's powerful multi-core CPUs to achieve efficient placement parallelism. For quadratic placers, wirelength is approximated as a quadratic objective, which can be minimized by solving symmetric and positive-definite (SPD) linear systems. The solution can be quickly approached by various Krylov-subspace methods, among which the preconditioned conjugate gradient (PCG) method is the most efficient known algorithm for placement problem. As solving SPD linear systems dominates the total runtime of placement, some previous effort has been made on parallelizing PCG from various angles. For example, SimPL [33], a

quadratic placer for application-specific integrated circuits (ASICs), achieved 1.89× speedup for PCG by solving x- and y-directed wirelength simultaneously with 2 threads. Further speedup is possible by parallelizing matrix-vector operations in PCG. However, the work in [49] showed that this technique cannot even achieve 2× speedup by using 16 threads in their experiments.

Table 4.1: Comparison of Recent FPGA and ASIC Placement Contest Benchmarks

Benchmark Suite		Avg. #Pin Per Movable Cell	Avg. #Pin Per Net
FPGA	ISPD 2016	4.99	4.95
	ISPD 2017	4.16	4.15
ASIC	ICCAD 2014	3.07	3.06
	ICCAD 2015	3.13	2.99

A more recent work, POLAR 3.0 [49], presented a quadratic placement parallelization framework for ASICs based on geometric partitioning. In particular, they divide cells into partitions based on their physical locations, then perform PCG and rough legalization [33] for each partition separately in parallel. In order to reduce solution quality loss, a full-netlist placement is performed periodically to allow inter-partition cell moving. Although their approach achieved promising results (4× speedup with 1.2% wirelength degradation by using 16 threads) on ASIC benchmarks, similar performance gain might not be reproducible to FPGA designs for the following two reasons: (1) placeable cells and nets in FPGA designs typically have more pins compared with ASIC designs (as shown in Table 4.1), which can lead to stronger inter-partition connections and larger quality degradation; (2) various FPGA

200

architecture constraints (e.g., clock legalization rules [17]) impose difficulties to the parallelization of rough legalization.

In this section, we propose a highly-parallelized quadratic placement framework for FPGAs, UTPlaceF 3.0, which takes full advantages of modern multi-core CPUs and delivers an appealing performance boost. Besides, the placement quality loss is also taken care of with only very little runtime overhead. The major contributions of this work are highlighted as follows.

- We propose a placement-driven block-Jacobi preconditioning technique that transforms a placement problem into a set of independent sub-problems, which can be solved in parallel.

- We propose a parallelized incremental placement correction technique to reduce placement quality degradation.

- The ISPD 2016 benchmark suite demonstrates the effectiveness of our framework. On average, UTPlaceF 3.0 achieves $5\times$ speedup by using 16 threads with only 3.0% wirelength degradation.

The rest of this section is organized as follows. Section 4.1.1 reviews the preliminaries of quadratic placement. Section 4.1.2 systematically studies the performance bottlenecks of state-of-the-art quadratic placers. Section 4.1.3 gives the details of UTPlaceF 3.0 algorithms. Section 4.1.4 shows the experimental results, followed by the summary in Section 4.1.5.

4.1.1 Preliminaries

4.1.1.1 Quadratic Placement

An FPGA netlist can be represented as a hypergraph $H = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$ is the set of cells and $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$ is the set of nets. Let $\boldsymbol{x} = \{x_1, x_2, ..., x_{|\mathcal{V}|}\}$ and $\boldsymbol{y} = \{y_1, y_2, ..., y_{|\mathcal{V}|}\}$ be the x and y coordinates of all cells. The wirelength-driven global placement problem is to determine cell position vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ such that the total wirelength is minimized. Wirelength is measured by the half-perimeter wirelength (HPWL) defined as follows:

$$\text{HPWL}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{e \in \mathcal{E}} \left(\max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j| \right). \tag{4.1}$$

As HPWL is not differentiable everywhere, quadratic placers approximate it by squared Euclidean distance between cells using various net models, such as hybrid net model [76] and bound-to-bound (B2B) net model [70]. Therefore, the wirelength cost function in quadratic placers is defined as

$$W(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}\boldsymbol{x}^T Q_x \boldsymbol{x} + \boldsymbol{c}_x^T \boldsymbol{x} + \frac{1}{2}\boldsymbol{y}^T Q_y \boldsymbol{y} + \boldsymbol{c}_y^T \boldsymbol{y} + const. \tag{4.2}$$

Since both Hessian matrices Q_x and Q_y in Equation (4.2) are SPD, minimizing $W(\boldsymbol{x}, \boldsymbol{y})$ is equivalent to solving the following two linear systems.

$$Q_x \boldsymbol{x} = -\boldsymbol{c}_x, \tag{4.3a}$$

$$Q_y \boldsymbol{y} = -\boldsymbol{c}_y. \tag{4.3b}$$

To eliminate cell overlaps, most state-of-the-art quadratic placers [39, 62, 63] adopted a cell-spreading technique, called rough legalization [33], which adds extra spreading force pointing to cells' anchor locations (i.e., locations that satisfy cell density constraint). After each rough legalization execution, linear systems (4.3a) and (4.3b) are updated accordingly and solved again. The loop of solving linear systems and performing rough legalization is repeated until cells are fully spread out.

4.1.2 Empirical Runtime Study

In this section, we will empirically analyze the runtime bottlenecks of a state-of-the-art quadratic placer and evidence that achieving a highly scalable parallel placement is indeed a challenging problem.
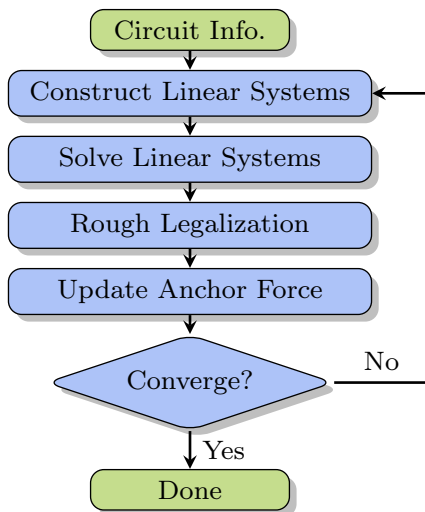
Figure 4.1: A representative rough legalization-based quadratic placement flow.

A representative rough legalization-based quadratic placement flow is

presented in Figure 4.1. To show its runtime bottlenecks, we implement a pure wirelength-driven UTPlaceF[1] and perform experiments on the ISPD 2016 benchmark suite with a single thread. As illustrated in Figure 4.2, the three major runtime contributors, on average, are solving linear systems (85.2%), rough legalization (7.5%), and linear system construction (7.3%), respectively.
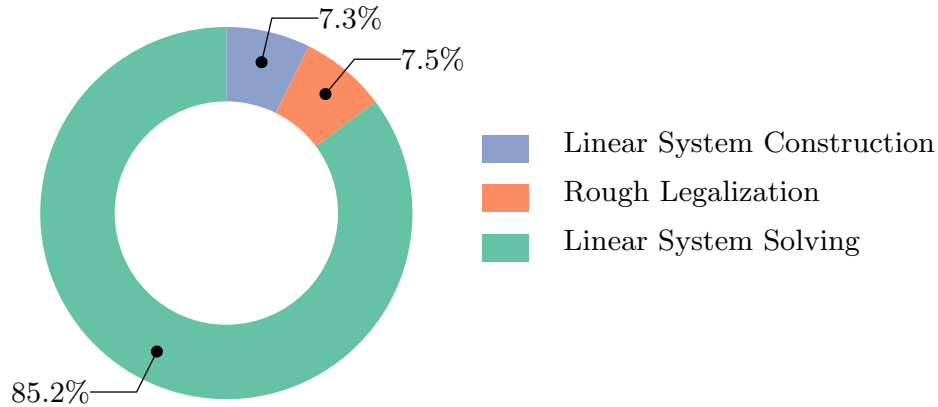
Figure 4.2: Normalized runtime of the three major runtime contributors in our pure wirelength-driven UTPlaceF on the ISPD 2016 benchmark suite.

Due to the decomposability of x- and y-directed wirelength in quadratic placement, the linear systems (4.3a) and (4.3b) can be constructed and solved perfectly in parallel using two threads. However, further parallelization becomes harder and inefficient.

The most efficient linear system construction approach is to scan and fill Hessian matrices Q_x and Q_y in a net-by-net manner. Considering multiple nets might contribute to the same entry in Q_x and Q_y (e.g., multiple nets

---

[1]We removed the routability optimization and global-move refinement in the original UTPlaceF described in Section 2.3.

share the same cell), there are two potential parallelization strategies: (1) processing nets in parallel and adding mutex lock[2] to each non-zero entry; (2) partitioning nets into groups and constructing partial Hessian matrices for each group in parallel, then, joining all partial Hessian matrices together. As can be seen, both strategies introduce considerable runtime overhead, therefore, linear system construction is difficult to be parallelized.

Solving linear systems, as the most time-consuming step, is inherently hard to parallelize due to the iterative nature of PCG. Experiments in [49] showed that only $1.8\times$ speedup can be achieved by using the cutting-edge PCG solver in Intel MKL library [45] on an 8-core machine.

Unless special care is taken, rough legalization can only handle one cell density hotspot at a time since spreading windows of multiple hotspots might intersect to each other. For each hotspot, the frequent cell sorting for horizontal and vertical cell spreading dominates the runtime. Although parallelization can be applied here, experiments in [33] only achieved $1.62\times$ speedup by using 8 threads.

To manifest the placement parallelization crisis, we further parallelize our pure wirelength-driven UTPlaceF in the following way utilizing OpenMP [2]: (1) constructing and solving linear systems for x and y in parallel; (2) enabling parallelization for matrix-vector operations in PCG if more than two threads are available; (3) substituting the sequential sorting in rough

---

[2]Mutex (mutually exclusive) lock is a mechanism that guarantees a resource can only be accessed by one thread at a time in a multi-threading environment.
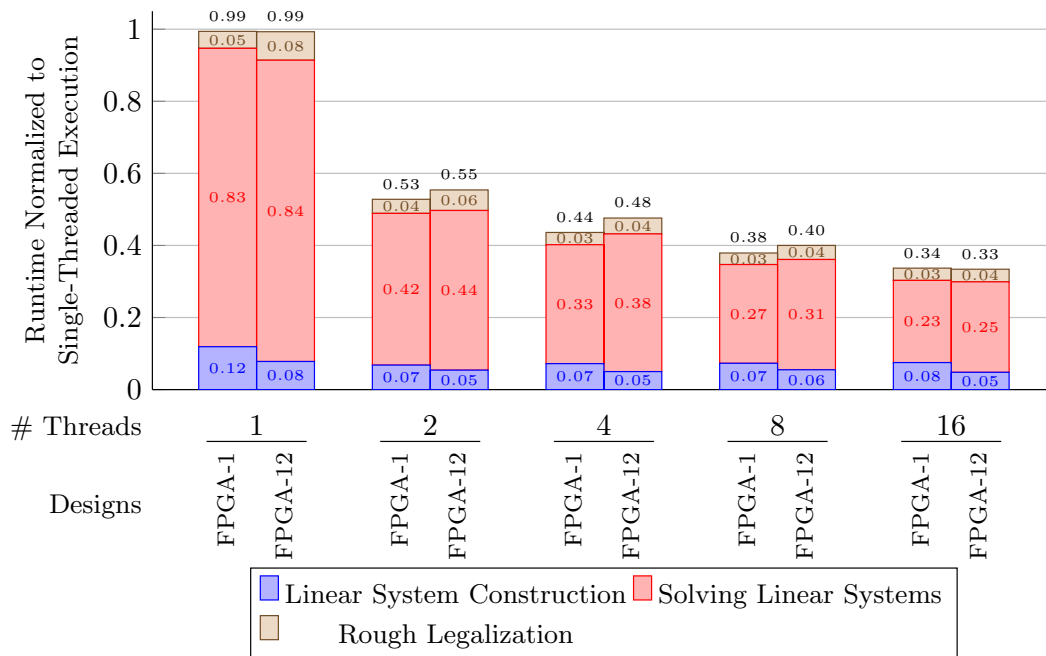
Figure 4.3: Runtime scaling of the three major runtime contributors in our pure wirelength-driven UTPlaceF by multi-threading. FPGA-1 and FPGA-12 contain 0.1M and 1.1M cells, respectively.

legalization with its equivalent parallel version in GNU libstdc++ parallel mode [46]. We then perform experiments using a 2.60 GHz Intel Xeon E5-2690 v3 CPU with 24 cores on two representative (the smallest and the largest) designs in the ISPD 2016 benchmark suite. We collect the runtime results under 1, 2, 4, 8, and 16 threads. Figure 4.3 presents the runtime scaling of the three major runtime contributors. As shown, most PCG speedup is from simultaneously solving x and y (1 thread vs 2 threads) and it plateaus out quickly on matrix-vector level parallelization (from 2 threads to 16 threads). Both linear system construction and rough legalization scale poorly. The overall speedup

saturates at around $3\times$.

In summary, decomposing x and y and low-level parallelization (i.e., parallel sorting and matrix-vector operations) alone can only achieve about $3\times$ speedup. In this work, we will explore innovative high-level approaches to break this parallelism wall.

4.1.3   UTPlaceF 3.0 Algorithms

4.1.3.1   Overall Flow
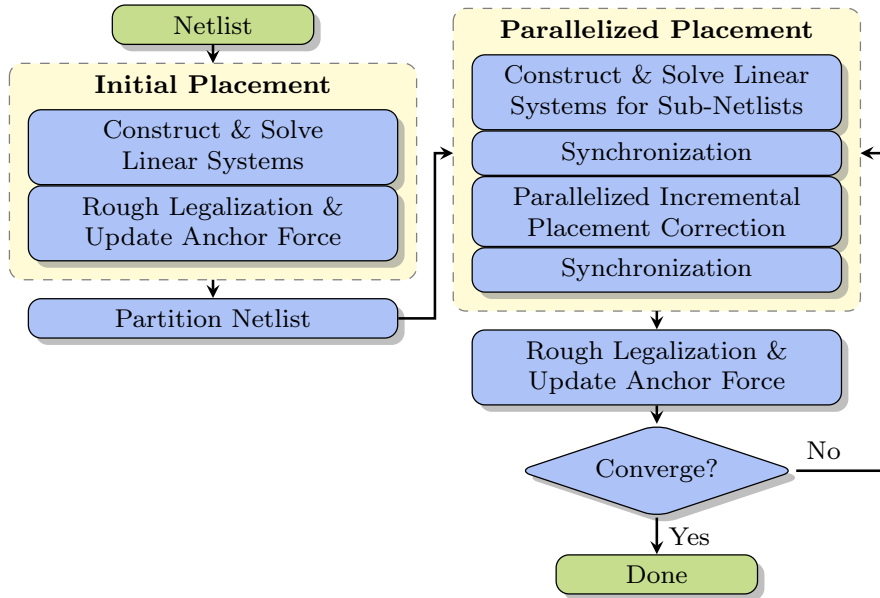
Figure 4.4: The overall flow of UTPlaceF 3.0.

The overall flow of UTPlaceF 3.0 is illustrated in Figure 4.4. The whole flow starts with the initial placement consisting of one pass of linear system construction and solving followed by the rough legalization and anchor force updating. After that, the netlist is divided into several sub-netlists utilizing

hypergraph minimum-cut partitioning techniques. It should be noted that the resulting sub-netlists are not necessarily physically separated. In practice, cells from different sub-netlists are likely mixed together. Once the partitioning is done, the linear systems of each sub-netlist are constructed and solved independently in parallel. However, because of the interdependency among sub-netlists (i.e., inter-partition nets), sub-netlist placement might not converge to the optimal solution, which leads to placement quality loss. Therefore, an incremental placement correction step, which takes the inter-partition dependency into consideration, is performed to reduce the quality degradation after all sub-netlist placements are done. Finally, rough legalization and anchor force updating are executed. The loop of the parallelized placement, rough legalization, and anchor force updating is repeated until the wirelength converges.

4.1.3.2 Placement-Driven Block-Jacobi Preconditioning

As demonstrated by the empirical study in Section 4.1.2, PCG dominates the total runtime of global placement. Thus, parallelizing PCG is of top priority. A common and effective PCG parallelization scheme is to leverage the block-Jacobi method. The block-Jacobi method essentially exploits the divide-and-conquer approach to transform a large linear system into a set of independent smaller sub-systems and solve each of them individually.

Figure 4.5 shows an example of applying an 8-way block-Jacobi method on a sparse SPD matrix Q. The first step (from Figure 4.5(a) to Figure 4.5(b))

(a)   (b)   (c)
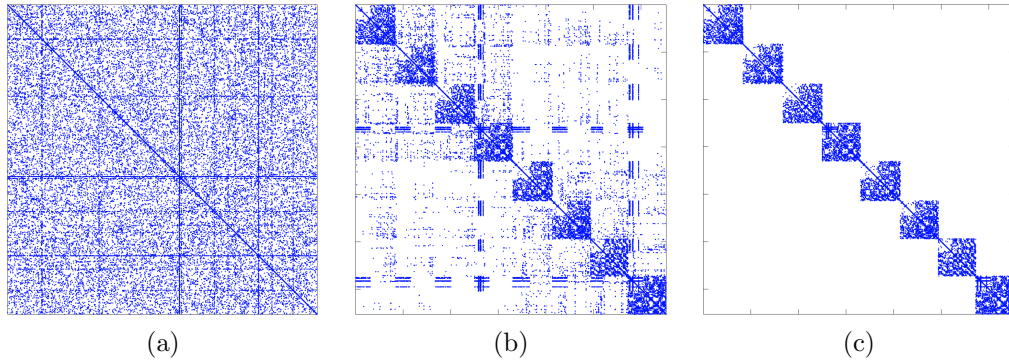
Figure 4.5: An example of building a block-Jacobi preconditioner from a sparse SPD matrix Q based on an 8-way partitioning. (a) The matrix Q. (b) After an 8-way partitioning by row/column permutation. (c) The resulting block-Jacobi preconditioner of Q.

is to find a good row/column permutation such that the sum of off-block-diagonal non-zero entries is minimized. As an SPD matrix can be represented as an undirected weighted graph (UWG), finding the best permutation for k diagonal blocks is equivalent to finding the k-way minimum-cut partitioning of the induced UWG. It should be noted that the permutated Q (Figure 4.5b) is mathematically equivalent to the original Q (Figure 4.5a). Thus, with the proper permutation on decision vectors ($\boldsymbol{x}$ and $\boldsymbol{y}$) and right-hand sides ($\boldsymbol{c}_x$ and $\boldsymbol{c}_y$), a permutated linear system is also equivalent to the original one. After the permutation, the block-Jacobi preconditioner in Figure 4.5(c) can be obtained by simply ignoring all off-block-diagonal non-zero entries in the permutated matrix in Figure 4.5(b).

From the placement point of view, each diagonal block in the block-Jacobi preconditioner corresponds to a partition (sub-netlist). Solving lin-

209

ear systems (4.3a) and (4.3b) with block-Jacobi preconditioners is physically equivalent to performing placement for each partition individually by assuming all off-partition cells are fixed at the location $(x, y) = (0, 0)^3$. Although this approach scales almost linearly with the number of partitions created, the placement quality could degrade dramatically as wrong physical locations are assumed for cells in inter-partition nets.

To mitigate this problem, rather than using (0, 0) as the fixed point, off-partition cells are considered fixed at their locations induced from the linear system solutions of the previous placement iteration. An example of a three-pin net spanning three partitions is shown in Figure 4.6. We call this process "placement-driven block-Jacobi preconditioning".

(a) (b) (c) (d)
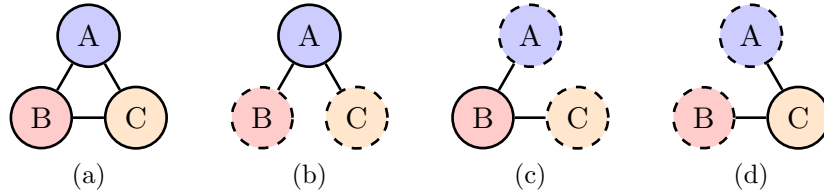
Figure 4.6: An inter-partition net spanning three partitions. (a) illustrates the net in the original netlist. (b), (c), and (d) show the net in the partition containing A, B, and C, respectively. Dashed circles represent off-partition cells that are fixed at their last locations.

Given a netlist consisting of the cell set $\mathcal{V}$, a partitioning $\mathcal{P}$ on $\mathcal{V}$, and the $\mathcal{P}$-permutated linear systems (4.3a) and (4.3b), if we denote the belonging

---

[3]The wirelength cost in x-direction for two cells can be written as $w_{i,j}(x_i - x_j)^2$. If cell i and j are not in the same partition, the term $-2w_{i,j}x_i x_j$ will be ignored in the block-Jacobi preconditioner. Therefore, this cost term becomes $w_{i,j}(x_i - 0)^2 + w_{i,j}(x_j - 0)^2$. For cell i (j), this is equivalent to assuming cell j (i) is fixed at $x = 0$. Similar conclusion is hold for y-direction.

partition of cell $i \in \mathcal{V}$ by $\pi(i)$ and denote the placement-driven block-Jacobi preconditioned linear systems of (4.3a) and (4.3b) by

$$\widehat{Q}_x \boldsymbol{x} = -\widehat{\boldsymbol{c}}_x, \tag{4.4a}$$

$$\widehat{Q}_y \boldsymbol{y} = -\widehat{\boldsymbol{c}}_y, \tag{4.4b}$$

then, $\widehat{Q}$ ($\widehat{Q}_x$ and $\widehat{Q}_y$) and $\widehat{\boldsymbol{c}}$ ($\widehat{\boldsymbol{c}}_x$ and $\widehat{\boldsymbol{c}}_y$) can be defined as

$$\widehat{Q}_{i,j} = \begin{cases} 0, & \text{if } \pi(i) \neq \pi(j), \forall i, j \in \mathcal{V}, \\ Q_{i,j}, & \text{otherwise}, \end{cases} \tag{4.5a}$$

$$\widehat{\boldsymbol{c}}_i = \boldsymbol{c}_i + \sum_{k \in \mathcal{V} \backslash \pi(i)} Q_{i,k} \, l_k, \forall i \in \mathcal{V}, \tag{4.5b}$$

where l_k denotes the (x or y) coordinate of cell $k \in \mathcal{V}$ in the previous placement iteration. Compared with the original block-Jacobi preconditioning, our placement-driven block-Jacobi preconditioning only differs by the right-hand sides in linear systems. Therefore, the SPD property is retained in linear systems (4.4a) and (4.4b) and diagonal blocks can still be solved independently by PCG.

Since the matrices Q_x and Q_y are placement dependent, from the quality perspective, it is ideal to perform two (one each for x and y) UWG minimum-cut partitionings for each placement iteration. However, doing so will result in a dramatic amount of runtime overhead. Therefore, only one partitioning is done right before the first parallelized placement step as shown in Figure 4.4. Considering Q_x and Q_y change after every placement iteration, hypergraph minimum-cut partitioning is adopted to approximate the varied

211

optimal matrix partitioning solutions.

4.1.3.3 Parallelized Incremental Placement Correction (PIPC)

Although applying placement-driven block-Jacobi preconditioning can reduce quality loss to some extent, it is no longer effective enough once more partitions (e.g., more than 4) are created. To further mitigate the placement quality degradation introduced by partitioning, the idea of additive correction multigrid method [28] is adopted to incrementally correct the solutions of the linear systems (4.4a) and (4.4b).

Without loss of generality, we only discuss x-direction in the rest of this session, but conclusions that hold for x-direction are also applicable to y-direction.

Let $\boldsymbol{x}^{(0)}$ be the solution of the linear system (4.4a), that is,

$$\widehat{Q}\boldsymbol{x}^{(0)} = -\widehat{\boldsymbol{c}}, \tag{4.6}$$

where $\widehat{Q}$ and $\widehat{\boldsymbol{c}}$ are defined in Equation (4.5a) and Equation (4.5b), respectively. If we can find $\Delta\boldsymbol{x}$ satisfying

$$Q\Delta\boldsymbol{x} = -\boldsymbol{c} - Q\boldsymbol{x}^{(0)}, \tag{4.7}$$

the solution of the original linear system (4.3a) will be $\boldsymbol{x}^{(0)} + \Delta\boldsymbol{x}$. Intuitively, $\Delta\boldsymbol{x}$ represents the discrepancy between current placement $\boldsymbol{x}^{(0)}$ and the optimal placement $\boldsymbol{x}^*$ and it is, physically, a very local and smooth perturbation around the placement $\boldsymbol{x}^{(0)}$.

Now we present the approach to find $\Delta \boldsymbol{x}$. Let $\boldsymbol{r}^{(0)} = -\boldsymbol{c} - Q\boldsymbol{x}^{(0)}$ and $N = \widehat{Q} - Q$. Then, solving linear system (4.7) is equivalent to solving

$$\widehat{Q}\Delta \boldsymbol{x} = N\Delta \boldsymbol{x} + \boldsymbol{r}^{(0)}. \tag{4.8}$$

The iterative method now becomes solving $\Delta \boldsymbol{x}^{(k)}$ for $k \geq 1$ with given $\Delta \boldsymbol{x}^{(0)}$ by

$$\widehat{Q}\Delta \boldsymbol{x}^{(k+1)} = N\Delta \boldsymbol{x}^{(k)} + \boldsymbol{r}^{(0)}, \tag{4.9}$$

which can be further written as

$$\Delta \boldsymbol{x}^{(k+1)} = \Delta \boldsymbol{x}^{(k)} + \widehat{Q}^{-1}(\boldsymbol{r}^{(0)} - Q\Delta \boldsymbol{x}^{(k)}). \tag{4.10}$$

Intuitively, if $\Delta \boldsymbol{x}^{(k)}$ and $\Delta \boldsymbol{x}^{(k+1)}$ are becoming closer and closer (i.e., $\Delta \boldsymbol{x}^{(k)}$ converges), we have $\boldsymbol{r}^{(0)} - Q\Delta \boldsymbol{x}^{(k)}$ approaches to 0. This indicates that once $\Delta \boldsymbol{x}^{(k)}$ converges, it will always converge to the same solution. To formally prove the convergence, consider the following lemma [65].

Lemma 4.1.1 *Let $Q = \widehat{Q} - N$, with Q and $\widehat{Q}$ symmetric and positive definite. If the matrix $2\widehat{Q} - Q$ is positive definite, then the iterative method defined in Equation (4.10) is convergent for any choice of the initial $\Delta \boldsymbol{x}^{(0)}$. Moreover, the convergence of the iteration is monotone with respect to the norm $|| \cdot ||_Q$ (i.e., $||\boldsymbol{r}^{(0)} - Q\Delta \boldsymbol{x}^{(k+1)}|| < ||\boldsymbol{r}^{(0)} - Q\Delta \boldsymbol{x}^{(k)}||$, $k = 0, 1, ...$).*

Now we can conclude the following proposition.

Proposition 4.1.2 *By picking any placement-driven block-Jacobi precondi-tioner of Q as $\widehat{Q}$, the iterative method defined in Equation (4.10) is monoton-*

ically convergent for any choice of $\Delta\boldsymbol{x}^{(0)}$.

Proof 1 *Since $|Q_{i,i}| > \sum_{j \neq i} |Q_{i,j}|, \forall i \in \mathcal{V}$, both Q and $\widehat{Q}$ are symmetric strictly diagonally dominant matrices. By the definition of $\widehat{Q}$ in Equation (4.5a), the matrix $\Omega = 2\widehat{Q} - Q$ can be defined as*

$$\Omega_{i,j} = \begin{cases} -Q_{i,j}, & \text{if } \pi(i) \neq \pi(j), \forall i,j \in \mathcal{V}, \\ Q_{i,j}, & \text{otherwise.} \end{cases}$$

So $\Omega = 2\widehat{Q} - Q$ is also a symmetric strictly diagonally dominant matrix. A symmetric strictly diagonally dominant real matrix with nonnegative diagonal entries is positive definite. Therefore, $2\widehat{Q} - Q$ is positive definite and, thus, the iterative scheme defined by Equation (4.10) will converge monotonically.

Proposition 4.1.2 reveals two important properties of the iterative method in Equation (4.10): (1) with a sufficient number of iterations, the exact solution, $\Delta\boldsymbol{x}$, of the linear system (4.7) can be reached, hence, we can obtain the exact solution, $\boldsymbol{x}^{(0)} + \Delta\boldsymbol{x}$, of the linear system (4.3a); (2) since the convergence is monotone, more iterations guarantee better solutions (i.e., solutions that are closer to the exact solution). These two properties imply that, by using this iterative method, placement quality and runtime can be perfectly traded to each other. Thus, different trade-offs can be made accordingly under different scenarios.

Another attractive property of the iterative method in Equation (4.10) is its strong parallelizability. Almost all the runtime of solving Equation (4.10)

is taken by computing $\widehat{Q}^{-1}(\boldsymbol{r}^{(0)} - Q\Delta\boldsymbol{x}^{(k)})$, which is equivalent to solving

$$\widehat{Q}\boldsymbol{x} = \boldsymbol{r}^{(0)} - Q\Delta\boldsymbol{x}^{(k)}. \tag{4.11}$$
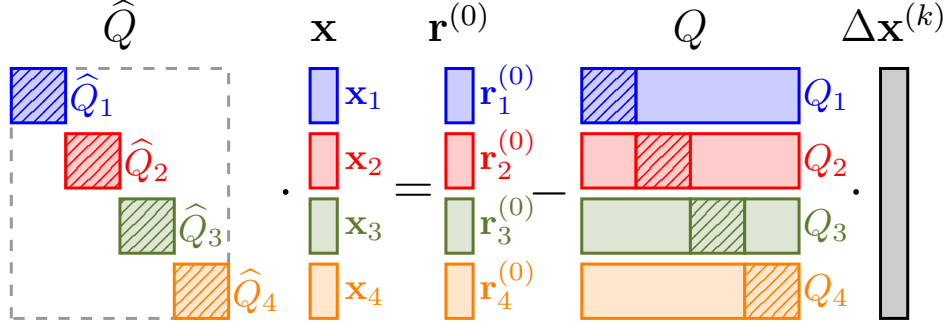
Figure 4.7: Parallelization scheme of solving linear system (4.11) with four partitions. Shaded regions represent diagonal blocks in $\widehat{Q}$ and Q. Different colors denote different partitions that can be solved in parallel.

Figure 4.7 illustrates the parallelization scheme of solving this linear system by an example with four partitions. By blocking matrix Q along rows, each strap in Q can be constructed independently and solving linear system (4.11) becomes solving the following sub-system for each partition separately and then assembling their solutions ($\boldsymbol{x}_i$) together.

$$\widehat{Q}_i\boldsymbol{x}_i = \boldsymbol{r}_i^{(0)} - Q_i\Delta\boldsymbol{x}^{(k)}, \forall i \in \mathcal{P}, \tag{4.12}$$

where $\widehat{Q}_i$, x_i, $\boldsymbol{r}_i^{(0)}$, and Q_i are partial matrices and vectors corresponding to partition i as illustrated in Figure 4.7, and $\mathcal{P}$ denotes the set of partitions.

Algorithm 15 summarizes the overall flow of our parallelized incremental placement correction (PIPC) scheme. The partial matrix Q_i of Q for each

partition is constructed in parallel from line 1 to line 3. Then the linear system (4.12) for each partition is solved in parallel from line 7 to line 9. After that, the solution of the linear system (4.11) is obtained by assembling partial solutions together at line 10. The solution after each correction iteration is incrementally updated at line 11. The loop from line 6 to line 13 is repeated until the correction iteration limit I is reached.

Algorithm 15: Parallelized Incremental Placement Correction

> **Input** : The set of partitions $\mathcal{P}$, the solution of the linear system (4.4a) $\boldsymbol{x}^{(0)}$, and the number of correction iterations I.
> **Output :** A better solution than $\boldsymbol{x}^{(0)}$.

1 **parallel foreach** $i \in \mathcal{P}$ **do**
2   Construct Q_i (See Figure (4.7));
3 **end**
4 $k \leftarrow 0$;
5 $\Delta\boldsymbol{x}^{(0)} \leftarrow 0$;
6 **while** $k < I$ **do**
7   **parallel foreach** $i \in \mathcal{P}$ **do**
8   Solve $\widehat{Q}_i \Delta\boldsymbol{x}_i^{(k+1)} = \boldsymbol{r}_i^{(0)} - Q_i \Delta\boldsymbol{x}^{(k)}$;
9   **end**
10      $\Delta\boldsymbol{x}^{(k+1)} \leftarrow (\Delta\boldsymbol{x}_1^{(k+1)}, \Delta\boldsymbol{x}_2^{(k+1)}, ..., \Delta\boldsymbol{x}_{|\mathcal{P}|}^{(k+1)})$;
11      $\Delta\boldsymbol{x}^{(k+1)} \leftarrow \Delta\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}^{(k+1)}$;
12      $k \leftarrow k + 1$;
13 **end**
14 **return** $\boldsymbol{x}^{(0)} + \Delta\boldsymbol{x}^{(I)}$;

4.1.3.4 Varied PIPC Configuration

In general, the placement quality degrades as the number of partitions increases. Therefore, it is not wise to perform the same amount of PIPC for different numbers of partitions, which might be overkill for small partition

counts but insufficient for a large number of partitions. To resolve this issue, PIPC is configured based on the number of partitions created.

In UTPlaceF 3.0, we configure three related variables: (1) the frequency of applying PIPC; (2) the number of correction iterations in each PIPC; (3) the number of PCG iterations in PIPC. For small partition counts, good placement quality can still be maintained by infrequent PIPC (e.g., once every 3 placement iterations) with only a few correction iterations (e.g., 1 or 2) each time. As the partition count increases, both PIPC frequency and correction iteration count need to be increased properly to control the quality degradation. The third variable, PCG iteration count in PIPC, is tuned to further save runtime. Since the solution of PIPC ($\Delta \boldsymbol{x}$) is essentially a local perturbation of the existing placement ($\boldsymbol{x}^{(0)}$), its magnitude is typically much smaller than $\boldsymbol{x}^{(0)}$. Therefore, we can save some PCG iterations in PIPC without losing too much placement quality. The detailed configuration will be further discussed in Section 4.1.4.1.

4.1.4   Experimental Results

UTPlaceF 3.0 is implemented in C++ and compiled by g++ 4.8.4. OpenMP 4.0 [2] is used to support multi-threading, GNU libstdc++ parallel mode [46] is used to parallelize the sorting in rough legalization, hypergraph partitioning tool PaToh [61] is used to partition netlists, and the PCG in Eigen3 [1] is used to solve sparse SPD linear systems. All the experiments are performed on a Linux machine running with Intel Xeon E5-2690 v3 CPUs

(2.60 GHz, 24 cores, and 30M L3 cache) and 64 GB RAM.

Table 4.2: ISPD 2016 Placement Contest Benchmarks Statistics

Benchmark	#LUT	#FF	#RAM	#DSP
FPGA-01	50K	55K	0	0
FPGA-02	100K	66K	100	100
FPGA-03	250K	170K	600	500
FPGA-04	250K	172K	600	500
FPGA-05	250K	174K	600	500
FPGA-06	350K	352K	1000	600
FPGA-07	350K	355K	1000	600
FPGA-08	500K	216K	600	500
FPGA-09	500K	366K	1000	600
FPGA-10	350K	600K	1000	600
FPGA-11	480K	363K	1000	400
FPGA-12	500K	602K	600	500
Resources	538K	1075K	1728	768

The benchmark suite released by Xilinx for the ISPD 2016 FPGA place-ment contest [79] is used to evaluate the efficiency of UTPlaceF 3.0. The statistics of the benchmarks are listed in Table 4.2. Since this work is focused on placement parallelization, all routability optimizations are removed and only wirelength is considered. In the experiments, the CLB resource demands of all LUTs and FFs are set to 0.08 and the target density is set to 1.0 for all benchmarks. In the target FPGA architecture, considering a vertical route needs to go through about twice as many switch boxes as a horizontal route needs for the same length, the wirelength is measured by the scaled HPWL (sHPWL) defined as

$$\text{sHPWL} = 0.5\,\text{HPWL}_x + \text{HPWL}_y, \qquad (4.13)$$

where HPWL_x and HPWL_y denote the x- and y-directed components of HPWL

in Equation (4.1), respectively.

4.1.4.1   Parallelization Configuration

Table 4.3: Parallelization Configuration of UTPlaceF 3.0 Under Different Numbers of Threads

# Threads	1	2	4	8	16
# Partitions	1	1	2	4	8
# PCG Iter.	250	250	250	250	250
PIPC Period	-	-	-	3	1
# Iter. of each PIPC	-	-	-	1	1
# PCG Iter. in PIPC	-	-	-	150	50

The parallelization configuration used for our experiments is presented in Table 4.3. If N ($N > 1$) threads are available, we always generate $\lfloor \frac{N}{2} \rfloor$ partitions and deal x and y separately using 2 threads in each partition. The number of PCG iterations for solving linear systems (4.3a), (4.3b), (4.4a), and (4.4b) are universally set to 250. PIPC is disabled for 1, 2, and 4 threads, since good placement quality can be achieved with placement-driven block-Jacobi preconditioning alone. For the case of 8 threads, PIPC is applied every 3 placement iterations and each time only one correction iteration is performed. For 16 threads, PIPC needs to be applied at every placement iteration to maintain good solution quality. The numbers of PCG iterations in PIPC are set to 150 and 50, respectively, for 8 and 16 threads. Although fewer PCG iterations are performed for 16 threads, the quality can be guaranteed by its more frequent PIPC calls.

219

4.1.4.2 PIPC Effectiveness Validation

The experimental results without PIPC (but with placement-driven block-Jacobi preconditioning) are presented in Table 4.4. We report the sH-PWL (WL) in the unit of 10^3, runtime (RT) in the unit of second and their ratios (WLR and RTR) normalized to corresponding 1-thread execution.

On average, without PIPC, UTPlaceF 3.0 achieves $1.7\times$, $2.8\times$, $4.4\times$, and $5.9\times$ speedup by using 2, 4, 8, and 16 threads. With 8 and 16 threads, the wirelength degrades by 2.4% and 6.5%, respectively.

Since PIPC is not applied for the cases of 1, 2, and 4 threads as shown in Table 4.3, we only report the results with PIPC enabled for 8 and 16 threads in Table 4.5. As can be seen, PIPC effectively reduces the wirelength degradation from 2.4% and 6.5% to 1.7% and 3.0% with only a little runtime overhead. With PIPC, UTPlaceF 3.0 can still achieve $5\times$ speedup by using 16 threads.

It is worthwhile to mention that PIPC is a general technique that can be configured for different runtime and quality trade-offs. The result shown in Table 4.5 is only a set of experiments to demonstrate the effectiveness of PIPC by using the configuration in Table 4.3.

Table 4.4: Comparison of UTPlaceF 3.0 without PIPC Under Different Numbers of Threads

Designs	1 Thread				2 Threads				4 Threads				8 Threads				16 Threads			
	WL	RT	WLR*	RTR†	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR
FPGA-01	227	44	1.000	1.000	227	24	1.000	0.546	225	15	0.987	0.341	233	10	1.025	0.220	233	7	1.027	0.165
FPGA-02	437	72	1.000	1.000	437	40	1.000	0.557	427	29	0.977	0.394	450	17	1.029	0.238	474	12	1.085	0.172
FPGA-03	1663	232	1.000	1.000	1663	131	1.000	0.567	1676	85	1.008	0.368	1750	51	1.052	0.222	1771	39	1.065	0.169
FPGA-04	3375	250	1.000	1.000	3375	148	1.000	0.589	3361	90	0.996	0.359	3408	61	1.010	0.242	3449	43	1.022	0.172
FPGA-05	6504	293	1.000	1.000	6504	173	1.000	0.589	6322	112	0.972	0.380	6489	71	0.998	0.243	6579	54	1.011	0.185
FPGA-06	3144	450	1.000	1.000	3144	249	1.000	0.555	3188	153	1.014	0.341	3106	103	0.988	0.229	3301	75	1.050	0.167
FPGA-07	5851	446	1.000	1.000	5851	251	1.000	0.562	5910	164	1.010	0.368	6004	99	1.026	0.223	6500	81	1.111	0.181
FPGA-08	5555	526	1.000	1.000	5555	298	1.000	0.567	5667	179	1.020	0.340	6059	119	1.091	0.225	6077	86	1.094	0.163
FPGA-09	7318	580	1.000	1.000	7318	323	1.000	0.557	7551	206	1.032	0.355	7564	136	1.034	0.235	7523	101	1.028	0.174
FPGA-10	3062	579	1.000	1.000	3062	316	1.000	0.546	3059	200	0.999	0.346	3058	128	0.999	0.221	3130	96	1.022	0.165
FPGA-11	6975	601	1.000	1.000	6975	335	1.000	0.558	7004	202	1.004	0.336	7053	133	1.011	0.221	8807	96	1.263	0.160
FPGA-12	3837	824	1.000	1.000	3837	407	1.000	0.494	3723	260	0.970	0.316	3943	168	1.028	0.205	3926	125	1.023	0.151
Geo. Mean	-	-	1.000	1.000	-	-	1.000	0.557	-	-	0.999	0.353	-	-	1.024	0.227	-	-	1.065	0.169

* WLR: Wirelength ratio normalized to the 1-thread execution.

† RTR: Runtime ratio normalized to the 1-thread execution.

Table 4.5: Comparison of UTPlaceF 3.0 with PIPC Under Different Numbers of Threads

Designs	8 Threads				16 Threads			
	WL	RT	WLR	RTR	WL	RT	WLR	RTR
FPGA-01	230	11	1.013	0.256	232	9	1.020	0.200
FPGA-02	448	19	1.024	0.261	453	15	1.037	0.207
FPGA-03	1719	57	1.034	0.247	1674	46	1.007	0.201
FPGA-04	3406	68	1.009	0.270	3379	51	1.001	0.204
FPGA-05	6413	85	0.986	0.290	6549	64	1.007	0.217
FPGA-06	3125	114	0.994	0.254	3203	91	1.019	0.203
FPGA-07	6160	115	1.053	0.258	6507	90	1.112	0.201
FPGA-08	5848	130	1.053	0.247	6151	103	1.107	0.197
FPGA-09	7576	156	1.035	0.268	7341	114	1.003	0.197
FPGA-10	3043	150	0.994	0.258	3036	116	0.992	0.200
FPGA-11	7077	154	1.015	0.256	7287	113	1.045	0.189
FPGA-12	3845	187	1.002	0.227	3897	144	1.016	0.175
Geo. Mean	-	-	1.017	0.257	-	-	1.030	0.199

Table 4.6: Runtime Breakdown of UTPlaceF 3.0 Under Different Numbers of Threads

Components	1 Threads		2 Threads		4 Threads		8 Threads		16 Threads	
	RTR[*]	RT%[†]	RTR	RT %	RTR	RT %	RTR	RT %	RTR	RT %
Solve Linear System	0.851	85.1%	0.449	80.6%	0.231	65.6%	0.125	48.5%	0.076	38.0%
Constr. Linear System	0.073	7.3%	0.047	8.4%	0.047	13.3%	0.034	13.4%	0.025	12.6%
Rough Legalization	0.075	7.5%	0.053	9.5%	0.042	11.9%	0.036	14.0%	0.033	16.7%
Partition Netlist	-	-	-	-	0.005	1.5%	0.009	3.6%	0.016	7.9%
PIPC	-	-	-	-	-	-	0.033	12.8%	0.034	16.9%
Others	0.001	0.1%	0.008	1.5%	0.027	7.7%	0.020	7.7%	0.016	7.9%
Total	1.000	100.0%	0.557	100.0%	0.353	100.0%	0.257	100.0%	0.199	100.0%

[*] RTR: Runtime ratio of each component w.r.t. the total runtime of 1-thread execution.
[†] RT%: The percentage of total runtime taken by each component under different threads.

4.1.4.3 Runtime Analysis

Table 4.6 presents the runtime breakdown of UTPlaceF 3.0 under different numbers of threads. We divide the total runtime into components of solving linear systems, linear system construction, rough legalization, netlist

partitioning, PIPC, and others. The normalized runtime (Norm. RT) and runtime percentage (RT %) are reported for each component under each thread count. As the number of threads increases, the runtime of solving linear systems reduces nearly linearly. However, the speedups of linear system construction and rough legalization saturate quickly. For the case of 16 threads, the runtime taken by each component becomes pretty much comparable.

According to the runtime breakdown, we can see that further speedup with more threads becomes difficult for the following four main reasons: (1) solving linear systems does not dominate the total runtime anymore, thus, the overall gain from it becomes limited; (2) linear system construction and rough legalization scale poorly; (3) the runtime of netlist partitioning is almost linear to the number of partitions; (4) more PIPC would be needed to maintain good placement quality. However, it is still possible to push the runtime down to the limit by the following several improvements: (1) combine low-level (matrix-vector operations) parallelization with our partition-based framework; (2) combine the high-level parallelization scheme proposed by [33] with our parallelization for sorting; (3) use parallelized partitioning tools (the one in UTPlaceF 3.0 now is single-threaded).

4.1.5 Summary

In this section, we have proposed a parallelization framework, UT-PlaceF 3.0, for modern FPGA quadratic placement. A placement-driven block-Jacobi preconditioning technique as well as a parallelized incremental

placement correction technique are proposed for boosting the overall performance of FPGA global placement. Our experiments demonstrate that, by fully leveraging today's multi-core systems, UTPlaceF 3.0 can achieve significant speedup while maintaining competitive placement quality.

Chapter 5

Conclusion

This dissertation has proposed a set of algorithms and methodologies for large-scale heterogeneous FPGAs. The major contributions can be summarized as follows.

- Chapter 2 has explored different core FPGA placement algorithms and methodologies. (a) Section 2.3 has presented UTPlaceF, a quadratic placer with a physical- and routability-aware packing algorithm. (b) Section 2.4 has described UTPlaceF-DL, a quadratic placer with a novel simultaneous packing and legalization algorithm that is inspired by the real-world college admission process. (c) Section 2.5 has developed elf-Place, a nonlinear placer that casts the heterogeneous density constraints to separate but unified electrostatic systems. UTPlaceF won the first-place award of the ISPD 2016 routability-driven FPGA placement contest held by Xilinx. UTPlaceF-DL and elfPlace further improve the essential placement quality and demonstrate that they are among the best approaches in the academic research.

- Chapter 3 has focused on algorithms to honor clock network feasibility during placement. (a) Section 3.3 has presented UTPlaceF 2.0,

which approximates clock demands based on a net bounding box model and resolves clock routing congestions by a novel iterative minimum-cost flow-based cell assignment algorithm. (b) Section 3.4 has described UTPlaceF 2.X, which is an extension of UTPlaceF 2.0. Instead of using bounding-box estimation, UTPlaceF 2.X explicitly constructs clock trees and explores a much larger clock routing solution space based on the branch-and-bound idea. UTPlaceF 2.0 won the first-place award of the ISPD 2017 clock-aware FPGA placement contest held by Xilinx. UT-PlaceF 2.X demonstrates even better solution quality over UTPlaceF 2.0, especially on designs with high clock utilization.

- Chapter 4 has explored placement algorithms to exploit the parallelism on multi-core systems. A general parallelization framework, UTPlaceF 3.0, is presented to achieve ultra-fast yet high-quality placement solutions. It leverages the idea of block-Jacobi preconditioning and additive multigrid method and demonstrates $5\times$ speedup with 3% wirelength degradation on industrial-strength benchmarks.

After the above explorations and discussions, this dissertation has demonstrated the significance of placement to the FPGA implementation quality and efficiency. With the recent successes of machine learning and the boom of hardware accelerators (e.g., FPGAs and GPUs), many conventional and emerging issues in FPGA implementations can be relieved. In particular, the following future research directions are of great interest and potential:

226

- Most FPGA implementation flows are developed targeting CPU platforms and executed on local machines. The recent advancement in distributed computing and FPGA- and GPU-based hardware acceleration can potentially reduce the FPGA implementation time by leveraging the power of data centers.

- Various early estimations can be made and improved by machine learning techniques. Potential candidates include but not limited to the prediction and estimation of timing, routability, power, and resource usage during or even before placement.

- Existing analytical models can be improved by machine learning models. One such example is the wirelength model. Existing models approximate HPWL using quadratic and nonlinear functions. While by leveraging deep neural networks, more accurate Steiner tree wirelength can be potentially modeled to better guide the analytical placement optimization.

Future FPGAs are expected to involve more cross-layer optimizations, where architectural design, logical design, and physical design are tightly linked together to push the performance of future FPGAs to their limits. Meanwhile, future FPGAs are likely to be designed and optimized for each specific application, e.g., deep neural networks, network dynamic routing, etc. Such a highly customized FPGA design methodology requires design automation tools to leverage domain-specific knowledge and achieve fast compilation in a self-adaptive manner.

Bibliography

[1] Eigen 3. `http://eigen.tuxfamily.org`.

[2] OpenMP 4.0. `http://www.openmp.org`.

[3] Ziad Abuowaimer, Dani Maarouf, Timothy Martin, Jeremy Foxcroft, Gary Gréwal, Shawki Areibi, and Anthony Vannelli. GPlace3.0: Routability-driven analytic placer for UltraScale FPGA architectures. *ACM Transactions on Design Automation of Electronic Systems (TO-DAES)*, 23(5):66:1–66:33, 2018.

[4] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: Theory, algorithms, and applications.* Prentice-Hall, Inc., 1993.

[5] Larry C. Andrews. *Special functions of mathematics for engineers.* SPIE Optical Engineering Press, 1992.

[6] Vaughn Betz and Jonathan Rose. Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size. In *IEEE Custom Integrated Circuits Conference (CICC)*, pages 551–554, 1997.

[7] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 213–222, 1997.

[8] Elaheh Bozorgzadeh, Seda Ogrenci-Memik, and Majid Sarrafzadeh. RPack: Routability-driven packing for cluster-based FPGAs. In *IEEE/ACM Asia and South Pacific Design Automation Conference (AS-PDAC)*, pages 629–634, 2001.

[9] Doris T. Chen, Kristofer Vorwerk, and Andrew Kennings. Improving timing-driven FPGA packing with physical information. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 117–123, 2007.

[10] Gang Chen and Jason Cong. Simultaneous timing driven clustering and placement for FPGAs. *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 158–167, 2004.

[11] Gang Chen and Jason Cong. Simultaneous placement with clustering and duplication. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(3):740–772, 2006.

[12] Gengjie Chen, Chak-Wa Pui, Wing-Kai Chow, Ka-Chun Lam, Jian Kuang, Evangeline F.Y. Young, and Bei Yu. RippleFPGA: Routability-driven simultaneous packing and placement for modern FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 37(10):2022–2035, 2018.

[13] Tung-Chieh Chen, Zhe-Wei Jiang, Tien-Chang Hsu, Hsin-Chen Chen, and Yao-Wen Chang. NTUplace3: An analytical placer for large-scale

mixed-size designs with preplaced blocks and density constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1228–1240, 2008.

[14] Yu-Chen Chen, Sheng-Yen Chen, and Yao-Wen Chang. Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 647–654, 2014.

[15] Chih-Liang Eric Cheng. RISA: Accurate and efficient placement routability modeling. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 690–695, 1994.

[16] Chung-Kuan Cheng, Andrew B. Kahng, Ilgweon Kang, and Lutong Wang. RePlAce: Advancing solution quality and routability validation in global placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

[17] ISPD 2017 clock-aware FPGA placement contest. `http://www.ispd.cc/contests/17`.

[18] Intel (Altera) Corp. `http://www.altera.com`.

[19] Nima Karimpour Darav, Andrew Kennings, Kristofer Vorwerk, and Arun Kundu. Multi-commodity flow-based spreading in a commercial analytic placer. In *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 122–131, 2019.

[20] Wenyi Feng. K-way partitioning based packing for FPGA logic blocks without input bandwidth constraint. In *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 8–15, 2012.

[21] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *ACM/IEEE Design Automation Conference (DAC)*, pages 175–181, 1982.

[22] Marshall L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management science*, 27(1):1–18, 1981.

[23] David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[24] Padmini Gopalakrishnan, Xin Li, and Lawrence Pileggi. Architecture-aware FPGA placement using metric embedding. In *ACM/IEEE Design Automation Conference (DAC)*, pages 460–465, 2006.

[25] Marcel Gort and Jason H. Anderson. Analytical placement for heterogeneous FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 143–150, 2012.

[26] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[27] Dwight Hill. Method and system for high speed detailed placement of cells within an integrated circuit design, 2002. US Patent 6,370,673.

[28] B. R. Hutchinson and G. D. Raithby. A multigrid method based on the additive correction strategy. *Numerical Heat Transfer, Part A: Applications*, 9(5):511–537, 1986.

[29] Xilinx Inc. `http://www.xilinx.com`.

[30] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[31] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *VLSI Design*, 11(3):285–300, 2000.

[32] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[33] Myung-Chul Kim, Dong-Jin Lee, and Igor L. Markov. SimPL: An effective placement algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 31(1):50–60, 2012.

[34] Yun-Chih Kuo, Chau-Chin Huang, Shih-Chun Chen, Chun-Han Chiang, Yao-Wen Chang, and Sy-Yen Kuo. Clock-aware placement for large-scale heterogeneous FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 519–526, 2017.

[35] Julien Lamoureux and Steven J. E. Wilton. On the trade-off between power and flexibility of FPGA clock networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(3):13:1–13:33, 2008.

[36] Eugene L. Lawler and David E. Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[37] Claude Lemaréchal. Lagrangian relaxation. In *Computational combinatorial optimization*, pages 112–156. Springer, 2001.

[38] Wuxi Li, Mehrdad E. Dehkordi, Stephen Yang, and David Z. Pan. Simultaneous placement and clock tree construction for modern FPGAs. In *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 132–141, 2019.

[39] Wuxi Li, Shounak Dhar, and David Z. Pan. UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing. In *IEEE/ACM International Conference on Computer-Aided Design (IC-CAD)*, pages 66:1–66:7, 2016.

[40] Wuxi Li, Shounak Dhar, and David Z. Pan. UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 37(4):869–882, 2018.

[41] Wuxi Li, Meng Li, Jiajun Wang, and David Z. Pan. UTPlaceF 3.0: A parallelization framework for modern FPGA global placement. In *IEEE/ACM International Conference on Computer-Aided Design (IC-CAD)*, pages 908–914, 2017.

[42] Wuxi Li, Yibo Lin, Meng Li, Shounak Dhar, and David Z. Pan. UT-PlaceF 2.0: A high-performance clock-aware FPGA placement engine. *ACM Transactions on Design Automation of Electronic Systems (TO-DAES)*, 23(4):42, 2018.

[43] Wuxi Li, Yibo Lin, and David Z. Pan. elfPlace: Electrostatics-based placement for large-scale heterogeneous FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[44] Wuxi Li and David Z. Pan. A new paradigm for FPGA placement without explicit packing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

[45] Intel Math Kernel Library. `https://software.intel.com/en-us/mkl`.

[46] GNU libstdc++ Parallel Mode. `https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html`.

[47] Tao Lin and Chris C.N. Chu. POLAR 2.0: An effective routability-driven placer. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[48] Tao Lin, Chris C.N. Chu, Joseph R. Shinnerl, Ismail Bustany, and Ivailo Nedelchev. POLAR: Placement based on novel rough legalization and refinement. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 357–362, 2013.

[49] Tao Lin, Chris C.N. Chu, and Gang Wu. POLAR 3.0: An ultrafast global placement engine. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 520–527, 2015.

[50] Tzu-Hen Lin, Pritha Banerjee, and Yao-Wen Chang. An efficient and effective analytical placer for FPGAs. In *ACM/IEEE Design Automation Conference (DAC)*, pages 10:1–10:6, 2013.

[51] Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z. Pan. DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement. In *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[52] Hanyu Liu and Ali Akoglu. Timing-driven nonuniform depopulation-based clustering. *International Journal of Reconfigurable Computing (IJRC)*, Article 3, 2010.

[53] Wen-Hao Liu, Yih-Lang Li, and Cheng-Kok Koh. A fast maze-free routing congestion estimator with hybrid unilateral monotonic routing. In *IEEE/ACM International Conference on Computer-Aided Design (IC-CAD)*, pages 713–719, 2012.

[54] Jingwei Lu, Pengwen Chen, Chin-Chih Chang, Lu Sha, Dennis Jen-Hsin Huang, Chin-Chi Teng, and Chung-Kuan Cheng. ePlace: Electrostatics-based placement using fast fourier transform and Nesterov's method. *ACM Transactions on Design Automation of Electronic Systems (TO-DAES)*, 20(2):17, 2015.

[55] Jingwei Lu, Hao Zhuang, Pengwen Chen, Hongliang Chang, Chin-Chih Chang, Yiu-Chung Wong, Lu Sha, Dennis Huang, Yufeng Luo, Chin-Chi Teng, et al. ePlace-MS: Electrostatics-based placement for mixed-size circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 34(5):685–698, 2015.

[56] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. Timing-driven partitioning-based placement for island style FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 24(3):395–406, 2005.

[57] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *ACM Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[58] Alexander S. Marquardt, Vaughn Betz, and Jonathan Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 37–46, 1999.

[59] Zied Marrakchi, Hayder Mrabet, and Habib Mehrez. Hierarchical FPGA clustering based on a multilevel partitioning approach to improve routability and reduce power dissapation. In *IEEE International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 25–28, 2005.

[60] Gi-Joon Nam, Sherief Reda, Charles J. Alpert, Paul G. Villarrubia, and Andrew B. Kahng. A fast hierarchical quadratic placement algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(4):678–691, 2006.

[61] PaToH. `http://bmi.osu.edu/umit/software.html`.

[62] Ryan Pattison, Ziad Abuowaimer, Shawki Areibi, Gary Gréwal, and Anthony Vannelli. GPlace: A congestion-aware placement tool for UltraScale FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 68:1–68:7, 2016.

[63] Chak-Wa Pui, Gengjie Chen, Wing-Kai Chow, Ka-Chun Lam, Jian Kuang, Peishan Tu, Hang Zhang, Evangeline F.Y. Young, and Bei Yu. RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 67:1–67:8, 2016.

[64] Chak-Wa Pui, Gengjie Chen, Yuzhe Ma, Evangeline F.Y. Young, and Bei Yu. Clock-aware UltraScale FPGA placement with machine learning routability prediction. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 915–922, 2017.

[65] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics.* Springer Science & Business Media, 2010.

[66] Senthilkumar Thoravi Rajavel and Ali Akoglu. MO-Pack: Many-objective clustering for FPGA CAD. In *ACM/IEEE Design Automation Conference (DAC)*, pages 818–823, 2011.

[67] Amit Singh, Ganapathy Parthasarathy, and Malgorzata Marek-Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):643–663, 2002.

[68] Love Singhal, Mahesh A. Iyer, and Saurabh Adya. LSC: A large-scale consensus-based clustering algorithm for high-performance FPGAs. In *ACM/IEEE Design Automation Conference (DAC)*, pages 30:1–30:6, 2017.

[69] Peter Spindler and Frank M. Johannes. Fast and accurate routing demand estimation for efficient routability-driven placement. In *IEEE/ACM Design, Automation and Test in Eurpoe (DATE)*, pages 1226–1231, 2007.

[70] Peter Spindler, Ulf Schlichtmann, and Frank M. Johannes. Kraftwerk2-a fast force-directed quadratic placement approach using an accurate net model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(8):1398–1411, 2008.

[71] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[72] Xilinx Vivado Design Suite. `https://www.xilinx.com/products/design-tools/vivado.html`.

[73] Russell Tessier and Heather Giza. Balancing logic utilization and area efficiency in FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 535–544. 2000.

[74] Marvin Tom and Guy Lemieux. Logic block clustering of large designs for channel-width constrained FPGAs. In *ACM/IEEE Design Automation Conference (DAC)*, pages 726–731, 2005.

[75] Marvin Tom, David Leong, and Guy Lemieux. Un/DoPack: Re-clustering of large system-on-chip designs with interconnect variation for low-cost FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 680–687, 2006.

[76] Natarajan Viswanathan and Chris C.N. Chu. FastPlace: Efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 24(5):722–733, 2005.

[77] M Xu, Gary Gréwal, and Shawki Areibi. StarPlace: A new analytic method for FPGA placement. *Integration, the VLSI Journal*, 44(3):192–204, 2011.

[78] Yonghong Xu and Mohammed A.S. Khalid. QPF: Efficient quadratic

placement for FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 555–558, 2005.

[79] Stephen Yang, Aman Gayasen, Chandra Mulpuri, Sainath Reddy, and Rajat Aggarwal. Routability-driven FPGA placement contest. In *ACM International Symposium on Physical Design (ISPD)*, pages 139–143, 2016.

[80] Stephen Yang, Chandra Mulpuri, Sainath Reddy, Meghraj Kalase, Srinivasan Dasasathyan, Mehrdad E. Dehkordi, Marvin Tom, and Rajat Aggarwal. Clock-aware FPGA placement contest. In *ACM International Symposium on Physical Design (ISPD)*, pages 159–164, 2017.

Vita

Wuxi Li received the B.S. degree in microelectronics from Shanghai Jiao Tong University, Shanghai, China, in 2013, and the M.S.E. degree in engineering from the University of Texas at Austin, Texas, US, in 2015. He started his Ph.D. program at the University of Texas at Austin in 2016, with research advisor David Z. Pan. He has interned at ARM, Austin in 2014 summer, Apple, Cupertino in 2014 fall, Apple, Austin in 2015 spring, summer, and fall, Cadence, Austin in 2016 summer and fall, and Xilinx, San Jose in 2018 summer.

Wuxi Li's research interests include VLSI design automation and optimization. He has received the Best Paper Award at DAC 2019, the Silver Medal in ACM Student Research Contest at ICCAD 2018, and the 1st-place awards in the FPGA placement contests of ISPD 2016 and 2017.

Permanent address: wuxi.li@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.