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

May 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 motivates me to become a better person not only mentally, but physically and spiritually. This dissertation would not have been possible without his unfailing guidance, support, and encouragement. 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 industry truly cares about. His enthusiasm and patience make 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

internship at Cadence, Austin. They shared their precious insights on physical design algorithms and taught me industrial-strength programming skills when I was a junior Ph.D. student. Dr. Mehrdad E. Dehkordi was mentor for my internship at Xilinx, San Jose. He provided various domain-specific knowledge and experiences on FPGA design automation.

In addition to my advisor, committee members, and colleagues, I would also like to recognize the help and effort that I received from many other people: Dr. Jhih-Rong Gao, Dr. Natarajan Viswanathan, and Dr. Mehmet Yildiz at Cadence, Dr. Ismail S. K. Bustany, Dr. Maogang Wang, Dr. Grigor Gasparyan, Dr. Yuji Kukimoto, Dr. Kai Zhu, and Vishal Suthar at Xilinx, Dr. Mahesh A. Iyer and Dr. Love Singhal at Intel, and UTDA members and alumni including Dr. Yibo Lin, Dr. Xiaoqing Xu, Dr. Meng Li, Dr. Bei Yu, Dr. Jiaojiao Ou, Dr. Derong Liu, Dr. Biying Xu, Shounak Dhar, Wei Ye, Zheng Zhao, Mohamed Baker Alawieh, Keren Zhu, Mingjie Liu, Rachel S. Rajarathnam, Zixuan Jiang, Jiaqi Gu, Che-Lun Hsu, and Jingyi Zhou. Their continuous encouragement and inspiring discussion throughout my years of study help develop and polish this dissertation.

I am deeply indebted to my family for their love, support, understanding, and sacrifices. Without them, this dissertation would never have been written. I dedicate this dissertation to my father, Dr. Wenbin Li, who initially stimulated my curiosity in the world of science and engineering and endowed himself over years on my education and intellectual development. I am also extremely grateful to my mother, Libo Wang, who has been a source of

motivation and strength during moments of discouragement. This last word of acknowledgment I have saved for my beloved wife, Muqiao Lin, who has always understood and been there for me.

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 often 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. In addition, 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. Given the special standing of FPGAs in fast

prototyping and frequent reprogramming, the implementation time is 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 non-linear 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 approahces are demonstrated with extensive experiments on industrial-strength benchmarks.

Table of Contents

List of Tables

List of Figures

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 Fig. 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 [73] and Vivado HLS [30]) 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

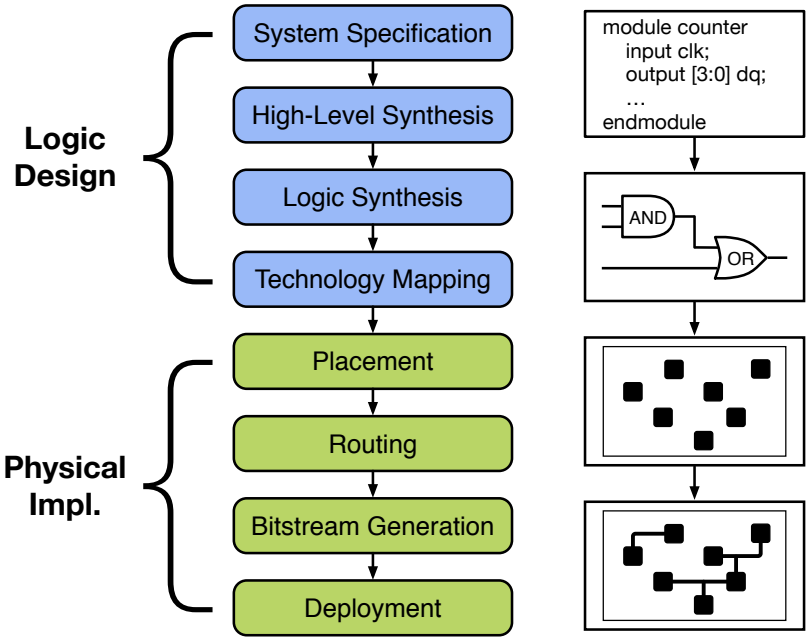


Figure 1.1: A typical FPGA CAD flow.

design is to realize these abstract logical representation into geometric implementation on the FPGA fabric. Physical design contains placement step to determine instance physical locations and routing step to realize physical connections among instances. At last, 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-

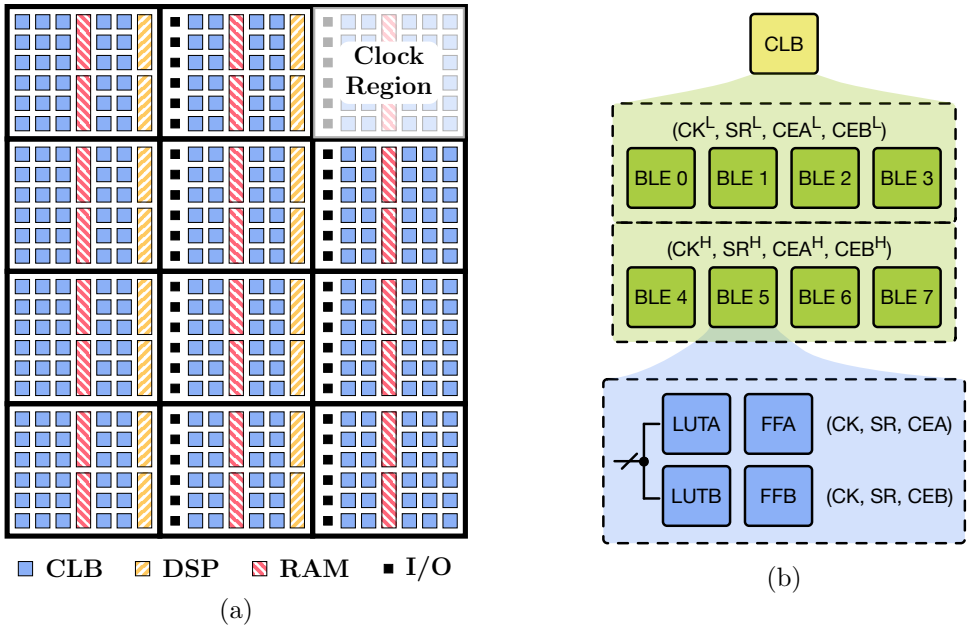


Figure 1.2: Illustrations of (a) a heterogeneous FPGA and (b) a configurable logic block (CLB).

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 often exclusively scattered over discrete locations on the FPGA fabric, as illustrated in Fig. 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 often burdens 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 have to optimize various objectives to meet the design targets and ensure smooth implementation con-

vergence. Typical placement objectives include wirelength, routability, timing, and power. Minimizing total wirelength is often regarded as the major objective, since wirelength 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 critical path. To meet a given frequency target, a placer have 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 have to honor a set of architectural constraints to produce feasible placement solutions. As shown in Fig. 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 maintian 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 to be 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 to placement. As shown in Fig. 1.2a, 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 aformentioned placement objectives. Packing aims at addressing the packing-legality issue, which is often not considered during gloabal 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 [6, 10] dominiated industry and academic research. SA approaches iteratively perform probabilistic swapping to progressively improve placement solutions. Desipte working well for small designs, it was no longer scalable as the FPGA capacity grows rapidly. Industry and academia then turned to min-cut approaches [58], which distribute placable instances by recursively partitioning the netlist. By leveraging the advancement in graph and hypergraph partitioning [20, 31–33], min-cut approaches were able to provide leading-edge performance on designs with tens of thousands gates. As the gate count of FPGAs reached the scale of millions, analytical approaches started to steadily outperform min-cut approaches in both runtime and quality. Different from the combinatorial-driven SA and min-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 [2, 11, 23, 24, 41, 45, 79, 80] and nonlinear approaches [13, 44, 51]. Quadratic approaches approximate placement objectives using efficient quadratic functions, while nonlinear approahces use more expressive higher-order ones.

The first milestone packing algorithm in the literature, VPack [5], was proposed in late 1990s. VPack pioneered the seed-based packing approaches. It constructs each CLB by first choosing a seed instance and iteratively grow-

ing 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 [7,53,60,68,69,75–77]. In contrast to the bottom-up strategy of seed-based approaches, min-cut packing approaches adopt the top-down recursive partitioning scheme to produce packing solutions [19, 61]. However, min-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 late 2000s, the concept of physical packing was first introduced to smooth the gap between packing and placement [8]. Besides considering netlist information, physical packing approahces further incorporate spatial information from a inital rough placement to ease the acutal 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 [2, 11, 41, 70].

1.3 Challenges of FPGA Placement

Although FPGA placement 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

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 great impact on the overall FPGA implementation performance, and therefore, essentailly improving placement quaity has always been a fundamental need. However, the increasingly complex and heterogeneous FPGA architectures has been consistently challenged the effectivenss 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 placement 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 of 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, enhancing algorithm scalability and improving 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 designs 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 [41], a quadratic placer with a sophisticated physical-aware packing algorithm; (2) UTPlaceF-DL [45], a quadratic placer with a novel simultaneous packing and legalization algorithm inspired by the real-world college admission process; (3) elfPlace [44], a nonlinear placer motivateid by the analogy between electrostatic system and placement.

Chapter 3 proposes two clock-aware placers that are capable of ensuring clock network feasiblity while achieving satisfiable solution quality: (1) UTPlaceF 2.0 [43], an extension of UTPlaceF that resolves clock routing congestion by a novel iterative minimum-cost flow method; (2) UTPlaceF 2.X [39], a generalization of UTPlaceF 2.0 that explores a even larger clock routing solution space based on a branch-and-bound algorithm.

Chapter 4 proposes a parallelization framework, UTPlaceF 3.0 [42], 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 satsifiable

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 problem 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.

This chapter is based on the following publications.

1. 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 37.4 (2018): 869-882.
2. Wuxi Li, and David Z. Pan. "Li, Wuxi, and David Z. Pan. "A New Paradigm for FPGA Placement without Explicit Packing." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2018).
3. Wuxi Li, Yibo Lin, and David Z. Pan. "elfPlace: Electrostatics-based Placement for Large-Scale Heterogeneous FPGAs." Submitted to the 38th International Conference on Computer-Aided Design (2019).

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

Quadratic placers approximate wirelength objective using quadratic functions [2, 11, 18, 23, 24, 79, 80], 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 overlapping. 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 use 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 overlapping-free placement is achieved.

Nonlinear placers, on the other hand, approximate placement objectives using higher-order nonlinear functions [13, 35, 51]. Different from quadratic placers, nonlinear placers formulate the overlapping-free 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 often achieve better solution quality compared with quadratic placers, but with the cost. 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

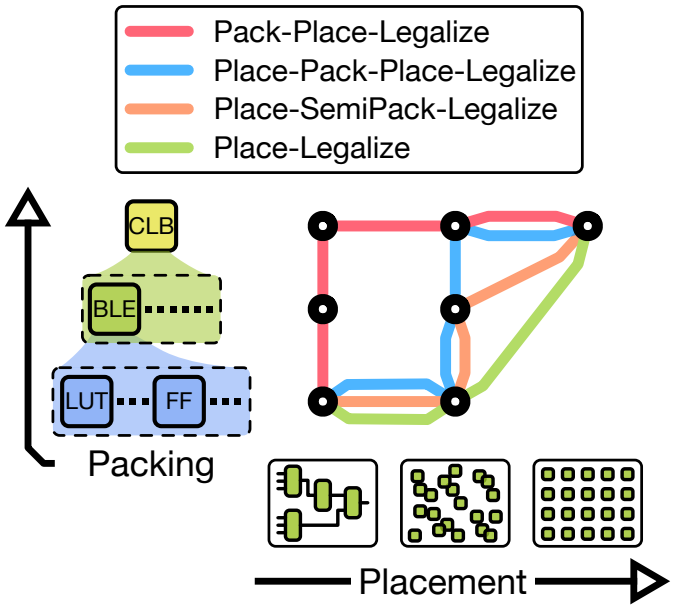


Figure 2.1: Representative FPGA placement methodologies.

age, *Pack-Place-Legalize* flows (the red path), such as [5–7,19,53,60,61,68,69], 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* flows do not incorporate placement/spatial information during their packing decision making, thus are likely to cause poor design quality. *Place-Pack-Place-Legalize* flows (the blue path) then emerged as a remedy to this issue [8, 35, 70]. 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 flows, namely *Place-SemiPack-Legalize* flows (the orange

path), blur the boundary between placement and packing [11]. In particular, after a FIP similar to that in *Place-Pack-Place-Legalize* flows, 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 analytical placement engines with different core algorithms and methodologies:

- **UTPlaceF**, a quadratic placer with *Place-Pack-Place-Legalize* flow.
- **UTPlaceF-DL**, a quadratic placer with a novel simultaneous packing and legaizaiton flow, as shown by the green path in Fig. 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

The all three placers that will be proposed in this chapter are developed based on Xilinx *UltraScale VU095* [30]. 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 Fig. 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 usually 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 constraint. More specifically, as shown in Fig. 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 CE (CEA/CEB).

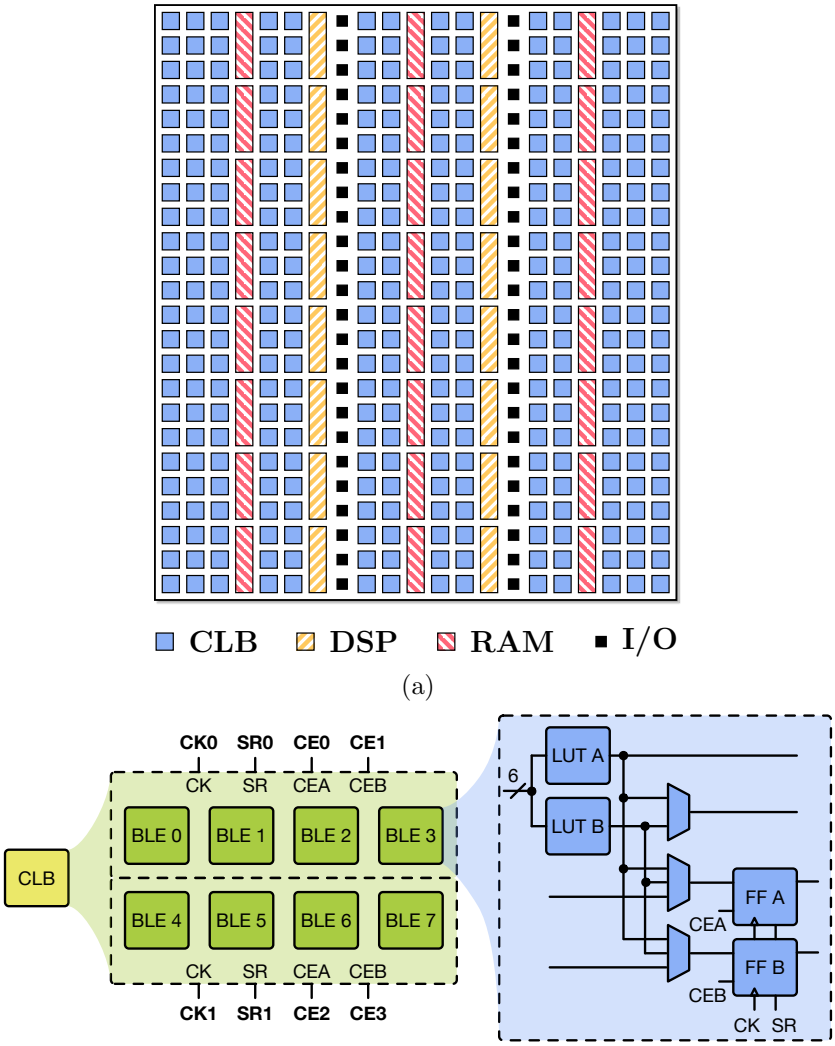


Figure 2.2: Illustration of (a) the layout and (b) the configurable logic block (CLB) structure of Xilinx UltraScale architecture.

2.3 UTPlaceF: A Packing-Based FPGA Placer

Packing-based methodologies are among the most successful FPGA placement approahces. In a packing-based placement methodology, a 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, power, etc.).

Although the packing-based placement methodologies have been intensively studied in the literature [5–7, 19, 53, 60, 61, 68, 69], exisiting approahces 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:

- 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 [81] benchmark suite released by Xilinx. Compared to the ISPD 2016 contest top-3 winners 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 presents the UTPlaceF framework overview. 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 $H = (V, E)$, where $V = \{v_1, v_2, ..., v_{|V|}\}$ is the set of cells, and $E = \{e_1, e_2, ..., e_{|E|}\}$ is the set

of nets. Let $\boldsymbol{x} = \{x_1, x_2, ..., x_{|V|}\}$ and $\boldsymbol{y} = \{y_1, y_2, ..., y_{|V|}\}$ 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 bin density constraint. Wirelength is measured by the half-perimeter wirelength (HPWL),

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

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

Fig. 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.

FIP is responsible for generating cells' 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 not perturb it too much. There-

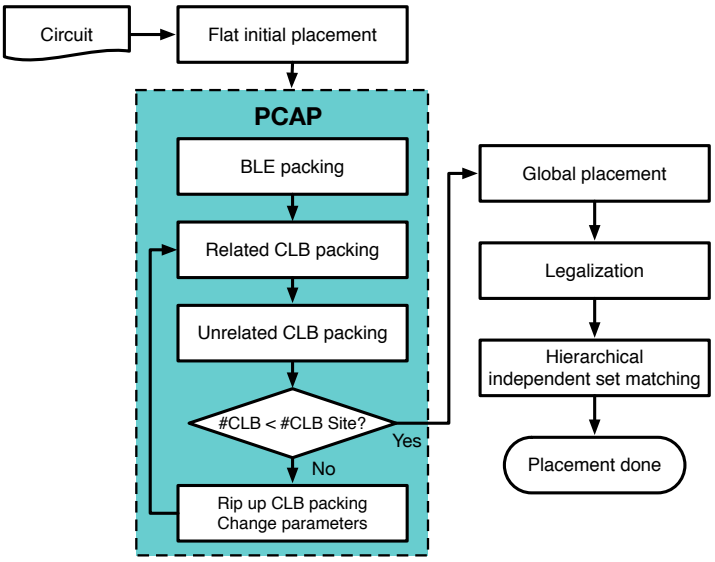


Figure 2.3: Overview of UTPlaceF.

fore PCAP, with cell physical locations information, disallows long-distance packing and prefers close packing. Similar to iRAC, absorbing small nets is treated as the main objective during 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. Leveraged by 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 whole netlists 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 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 optimized global placement solution, white spaces and cells in congested regions are handled specially throughout the detailed placement stage.

2.3.2 Flat Initial Placement

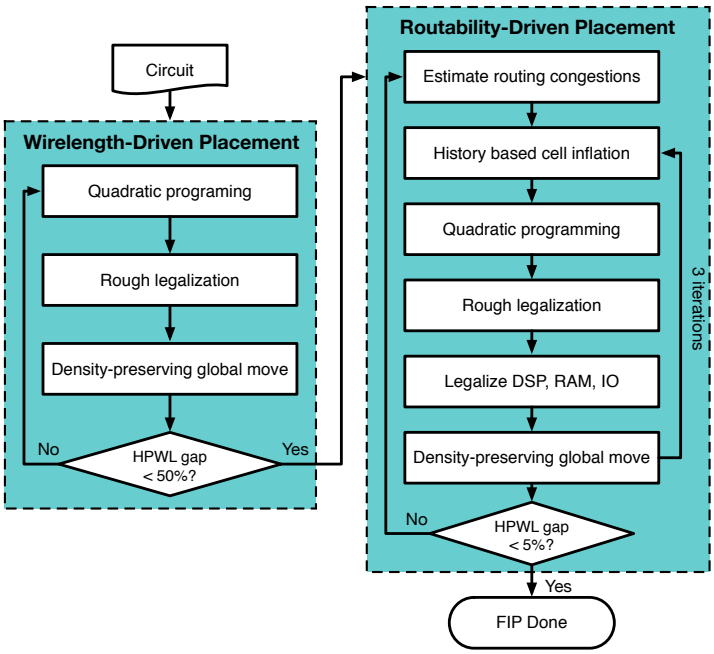


Figure 2.4: Overall flow of FIP.

Our FIP adopts the main framework of an ASIC placer *POLAR 2.0* [49]. Its overall flow is shown in Fig. 2.4. In each iteration of wirelength-driven placement, a quadratic program is solved followed by rough legalization

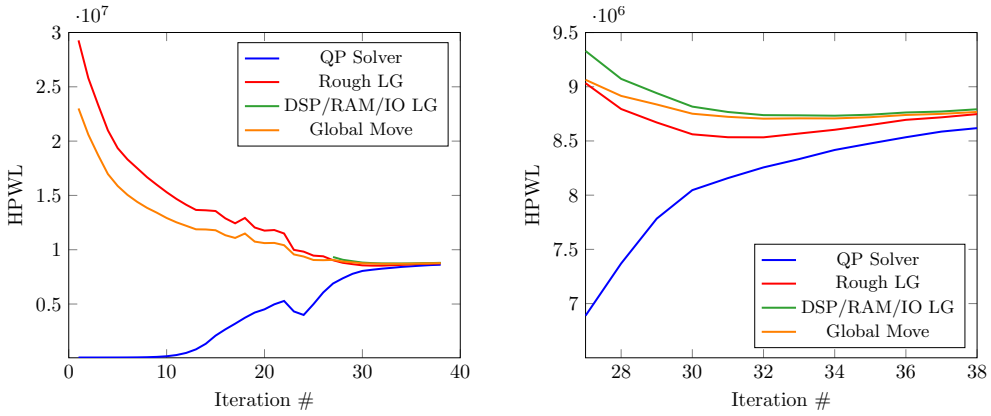


Figure 2.5: HPWL at different steps in FIP of benchmark FPGA-5. (a) All placement iterations (b) Placement iterations in routability-driven stage.

[34] for reducing cells overlaps. Then the density preserving global move [50] is applied to improve the wirelength of the rough-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 a certain number, which is 50% in PCAP. In the routability-driven placement stage, after a certain number of placement iterations, a fast global router NCTUgr [54] is called for routing congestion estimation, then similar to POLAR 2.0, cells in congested regions are inflated by a small ratio, and the 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 fi-

nal FIP solution. This discrepancy would introduce inaccuracy of cell relative positions into FIP. 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. Fig. 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 Fig. 2.5(b).

The two objectives of FIP are: (1) generating physical locations for each LUT and FF; (2) detecting LUTs and FFs 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 history based cell inflation. The insight of cell inflation is quantifying 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 LUT and FF would have a physical location and a cell area, which indicates the congestion level associated with it.

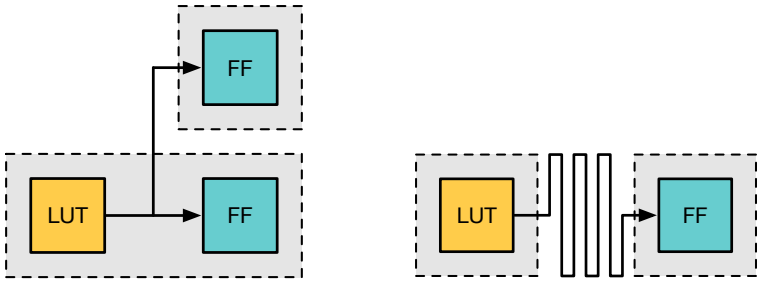


Figure 2.6: Different LUT and FF pairing scenarios. (a) LUT fanouts to multiple FFs, and group with the closest one. (b) LUT fanouts to one FF that is far away and the grouping is rejected.

2.3.3 Physical and Congestion Aware Packing

2.3.3.1 Max-Weighted-Matching-Based BLE Packing

As a BLE in our target FPGA architecture, Xilinx Ultrascale, contains 2 LUTs and 2 FFs, existing VPR-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 Fig. 2.6, we group each LUT to the closest FF in its fanout. Besides that, long-distance packing is rejected, and only packing within *maximum packing distance of BLE* ($\overline{\lambda_b}$) 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, *max-weighted matching* is used for finalizing the

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

- v_i and v_j are connected in the netlist.
- Merging v_i and v_j into the same BLE does not violate any packing rules.
- The merging attraction is greater than the *minimum packing attraction for BLEs* ($\underline{\phi_b}$).

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

$$\phi_b(v_i, v_j) = (1 - e^{\gamma_b(dist(v_i,v_j)-\overline{\lambda_b})}) \sum_{p \in Net(v_i \cap v_j)} \frac{k_b}{deg(p) - 1}. \tag{2.3}$$

where γ_b is a constant value being experimentally set as 0.2, $dist(v_i, v_j)$ is the Manhattan distance between v_i and v_j, $\overline{\lambda_b}$ is the maximum packing distance of BLE which is 4 in PCAP, $Net(c_i \cap c_j)$ is the set of nets shared between v_i and v_j, $deg(p)$ is the total number of pins of net p that is exposed in cluster level, and k_b is 2 for 2-pin nets and 1 for other nets.

The first term, $1 - e^{\gamma_b(dist(v_i,v_j)-\overline{\lambda_b})}$, is a packing distance penalty factor in range $(-\infty, 1 - e^{-\gamma_b\overline{\lambda_b}})$. This factor is very close to 1 for mergings of distance much less than $\overline{\lambda_b}$. It drops quickly as $dist(v_i, v_j)$ gets close to $\overline{\lambda_b}$, and

becomes negative once $dist(v_i, v_j)$ is greater than $\overline{\lambda_b}$. By using the first term, short-distance mergings in PCAP are always preferred. The second term, $\sum_{p \in Net(v_i \cap v_j)} \frac{k_b}{deg(p)-1}$, is introduced for reducing the number of nets exposed in cluster level, which in turn improves wirelength and routability. With the second term, merging two clusters that share more small nets is of high priority, and 2-pin nets are given even higher weight by the factor $k_b = 2$. In PCAP, the minimum packing attraction for BLEs ($\underline{\lambda_b}$) is set to 0 by default.

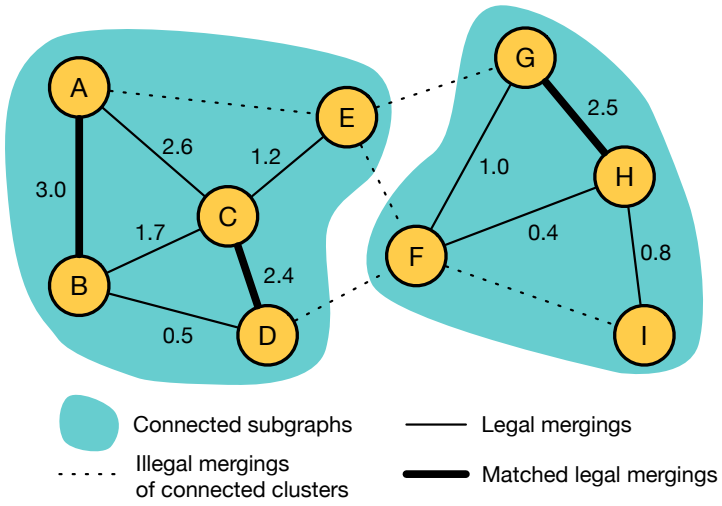


Figure 2.7: A simple max-weighted cluster matching example.

Fig. 2.7 shows a simple cluster matching example. Due to our rules for legal mergings, the constructed UWG typically is not connected and comprises many small connected subgraphs. Mergings in different connected subgraphs are independent, so PCAP performs a max-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 stage and resolves the max-weighted matching for each new connected subgraph until no more legal merging exists.

The pseudo-code of the max-weighted-matching-based BLE packing is summarized in Alg. 1. Each connected subgraph is constructed by calling the function `ConstructConnectedUWG` 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 max-weighted matching would be run on the constructed subgraph at line 7, and mergings corresponding to matched edges would be committed to the netlist from line 8 to line 12. The loop of constructing subgraphs, solving max-weighted matching, and committing matched mergings are iteratively executed from line 1 to line 13, and stops at line 13 when no more merging can be performed. In PACP, it's very time-consuming to consider all neighbors of a cell connected by high-degree nets at line 22. So for each cell, we only consider its neighbors connected by nets containing at most 16 pins.

The time complexity of Alg. 1 turns out to be $\mathcal{O}(|V|(k^3\frac{|V|}{|E|} + |E_c|log|V_c|))$, where V and E denote the set of clusters and nets in the netlist, respectively, k denotes the average number of pins in each cluster in V, and $|V_c|$ and $|E_c|$ are the average numbers of vertices and edges in each connected subgraph in the UWG.

On average, each cluster is incident to k nets and each net contains

Algorithm 1: Max-Weighted-Matching-Based BLE Packing

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

while *true* **do**
  $numMatching \leftarrow 0$;
  $status[u] \leftarrow Untouched\ \forall u \in V$;
  foreach $s \in V$ **do**
    if $status[s] \neq Untouched$ **then continue**;
    $g \leftarrow$ `ConstructConnectedUWG`(s);
    Run max-weighted matching on g;
    foreach *matched edge* $(u, v) \in g$ **do**
      $c \leftarrow \{u, v\}$;
      $V \leftarrow V \setminus \{u, v\} \cup \{c\}$;
      $status[c] \leftarrow Popped$;
      $numMatching \leftarrow numMatcahing + 1$;
  if $numMatching == 0$ **then return**;

Function `ConstructConnectedUWG(`s`):`
  Initialize an empty UWG g;
  Initialize an empty queue q;
  Push s into q;
  $status[s] \leftarrow InQueue$;
  Add s into g;
  while q *is not empty* **do**
    $t \leftarrow$ fetch and pop q top;
    foreach v *connected to* t **do**
      if $status[v] \neq Popped$ *and* $\phi_b(t, v) \geq \underline{\phi_b}$ **then**
        if $status[v] == Untouched$ **then**
          Push v into q;
          $status[v] \leftarrow InQueue$;
          Add v into g;
        Add edge $(t, v, \phi_b(t, v))$ into g;
  return g;

$k\frac{|V|}{|E|}$ clusters, so each cluster in V has $\mathcal{O}(k^2\frac{|V|}{|E|})$ connected neighbors at line 22. Since each attraction calculation (Eq. (2.3)) at line 23 takes $\mathcal{O}(k)$ time, the graph construction (`ConstructConnectedUWG`) for each connected subgraph, $G_c = (V_c, E_c)$, has time complexity of $\mathcal{O}(k^3\frac{|V|}{|E|}|V_c|)$. Besides, solving the max-weighted matching at line 7 takes $\mathcal{O}(|V_c||E_c|log|V_c|)$ time and the cell mergings from line 8 to line 12 can be done in $\mathcal{O}(|E_c|)$ time. The time complexity of each connected subgraph (line 6 – 12) can be bounded by $\mathcal{O}(|V_c|(k^3\frac{|V|}{|E|} + |E_c|log|V_c|))$. Considering we need to handle $\mathcal{O}(\frac{|V|}{|V_c|})$ such connected subgraphs, Alg. 1, therefore, has total time complexity of $\mathcal{O}(|V|(k^3\frac{|V|}{|E|} + |E_c|log|V_c|))$. In practice, both $|V_c|$ and $|E_c|$ are relatively small (less than 200) and independent to the netlist size.

2.3.3.2 Related CLB Packing with Congestion-Aware Depopulation

After BLE packing, we create a CLB for each single BLE. CLB packing is done by successively merging smaller CLBs into larger ones. CLBs that share common nets are said to be related, and in this stage, only related CLBs mergings are considered.

The *BestChoice Clustering (BC)* [62] is used as our main engine for related CLB packing. In BC, the attractions of all legal CLB mergings are calculated first, then the algorithm iteratively merges CLB pairs with the highest attraction using a priority queue (PQ). The location and area of a merged CLB is set as the average location and total area of cells it contains, respec-

tively. After each merging, the legality and attraction of mergings related to the new CLB are updated accordingly.

In PCAP, a related CLB merging (c_i, c_j) is said to be legal if and only if

- c_i and c_j are connected in the netlist.
- Merging c_i and c_j into the same CLB does not violate any packing rules.
- The merging attraction is greater than the *minimum packing attraction for related CLBs* ($\underline{\phi_{rc}}$).
- Total area of c_i and c_j is no greater than the *maximum CLB area* ($\overline{a_c}$).

The first three rules are inherited from our BLE packing. The fourth rule is introduced to perform congestion-aware depopulation and avoid over-packing in routing congested regions. Note that all the cell areas used in the fourth rule are from the accumulated cell inflation in FIP. As discussed in Section 2.3.2, cells with larger areas indicate higher possibility to be placed into routing congested regions. By constraining area of each CLB, PCAP would apply loose packing in routing congested regions, and tight packing in other regions. This congestion awareness makes PCAP able to achieve a good trade-off between wirelength and routability. Fig. 2.8 illustrates our congestion-aware depopulation technique. Note that BLEs with larger areas are in routing congested regions.

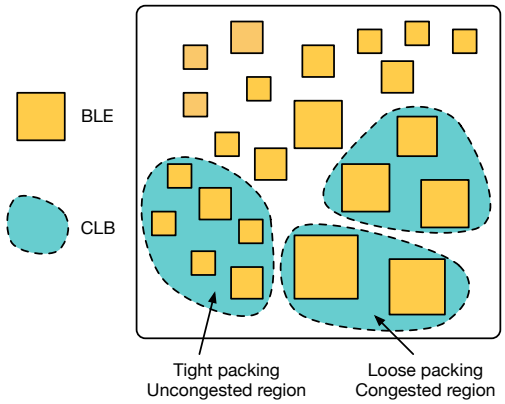


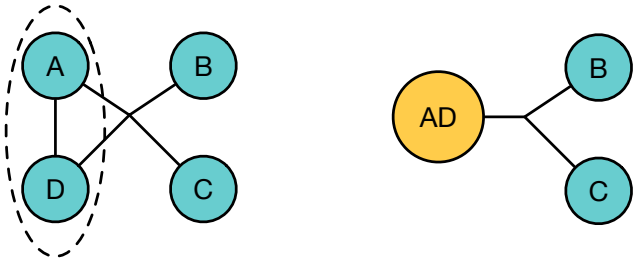
Figure 2.8: Congestion-aware depopulation of PCAP during CLB packing.

The attraction function of related CLB mergings (c_i, c_j) is defined as,

$$\phi_{rc}(c_i, c_j) = (1 - e^{\gamma_{rc}(dist(c_i,c_j)-\overline{\lambda_{rc}})}) \sum_{p \in Net(c_i \cap c_j)} \frac{k_{rc}}{deg(p) - 1}. \tag{2.4}$$

Eq. (2.4) is basically a replica of our BLE packing attraction function defined in Eq. (2.3), but differs only by some constant parameters. We experimentally set γ_{rc} to 0.2, and $\overline{\lambda_{rc}}$ to 6. k_{rc} is 2 for 2-pin nets and 1 for other nets. The minimum packing attraction for related CLBs ($\underline{\phi_{rc}}$) is set to 0.1, and the maximum CLB area ($\overline{a_c}$) is set as 1.8 times average CLB area by default.

Our BC-based related CLB packing algorithm has several major differences compared with the traditional BC clusterings. In traditional BC, a speedup technique called *lazy update* [62] is widely adopted, and this technique relies on their observation that the vast majority of attraction updates are decreasing their ranking in the PQ. However, this observation does not apply to our packing algorithm. Fig. 2.9 shows a simple example, before merging clus-



(a) (b)

Figure 2.9: Merging a pair of clusters A and D, assuming $dist(B, C) = 1$, $\gamma_{rc} = 0.2$, $\overline{\lambda_{rc}} = 6$. (a) Before merging A and D, $\phi_{rc}(B, C) \approx 0.211$, (b) After merging A and D, $\phi_{rc}(B, C) \approx 0.316$.

ters A and D, B and C only share a 4-pin net, and after the merging, the 4-pin net becomes a 3-pin net in the cluster level netlist, therefore the attraction between B and C increases due to the second term in Eq. (2.4). We can see that this kind of attraction increases could happen to any cluster pairs that share nets with more than 3 pins and all attraction updates in our BC-based related CLB packing would increase their ranking in the PQ. Therefore, 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 attractions affected by each committed merging.

The pseudo-code of our BC-based related CLB packing is summarized in Alg. 2. Initially, all legal merging candidates are pushed into a PQ from line 3 to line 6. Each time the merging candidate with the highest attraction is popped from the PQ top at line 8 and is committed to the netlist from line

Algorithm 2: BC-based Related CLB Packing

Input : BLE packing is done.

Output: CLB level netlist with external nets reduced. All CLB packing rules are satisfied. Cells in routing congested regions are not overpacked.

Create an empty priority queue PQ;
$valid[u] \leftarrow true\ \forall u \in V$;
foreach $u \in V$ **do**
  foreach v *connected to* u **do**
    if $\phi_{rc}(u, v) \geq \underline{\phi_{rc}}$ **then**
      Push $(u, v, \phi_{rc}(u, v))$ into PQ
while PQ *is not empty* **do**
  $(c_i, c_j, \phi_{rc}(c_i, c_j)) \leftarrow$ fetch and pop PQ top;
  if $valid[c_i]$ *and* $valid[c_j]$ **then**
    $c_{ij} \leftarrow \{c_i, c_j\}$;
    $V \leftarrow V \setminus \{c_i, c_j\} \cup \{c_{ij}\}$;
    $valid[c_i], valid[c_j] \leftarrow false$;
    $valid[c_{ij}] \leftarrow true$;
    foreach c_k *connected to* c_{ij} **do**
      if $\phi_{rc}(c_k, c_{ij}) \geq \underline{\phi_{rc}}$ **then**
        Push $(c_k, c_{ij}, \phi_{rc}(c_k, c_{ij}))$ into PQ
    foreach *net* e *shared by* c_i *and* c_j **do**
      foreach $(c_p, c_q, \phi_{rc}^{pq})$ *such that that* c_p *and* c_q *share* e **do**
        if $valid[c_p]$ *and* $valid[c_q]$ **then**
          $\phi_{rc}^{pq} \leftarrow \phi_{rc}(c_p, c_q)$;
          Update the ranking of $(c_p, c_q, \phi_{rc}^{pq})$ in PQ;

10 to line 11. New merging candidates related to 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 21. The loop from line 7 to line 21 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 Alg. 2 line 4 and line 14 to speed up the runtime.

Alg. 2 turns out to have $\mathcal{O}((1 + \frac{|V|}{|E|})k^2\frac{|V|^2}{|E|}(k + log(k^2\frac{|V|^2}{|E|})))$ time complexity, where V and E denote the set of clusters and nets in the BLE-level netlist, respectively, and k is the average number of pins in each cluster in V.

We first consider the PQ initialization from line 1 to line 6. Each cluster in V has $\mathcal{O}(k^2\frac{|V|}{|E|})$ connected neighbors, since, on average, each cluster is incident to k nets and each net contains $k\frac{|V|}{|E|}$ clusters. Thus, initially, there are $\mathcal{O}(k^2\frac{|V|^2}{|E|})$ merging candidates being pushed into the PQ and the initialization time can be bounded by $\mathcal{O}(k^2\frac{|V|^2}{|E|}log(k^2\frac{|V|^2}{|E|}))$.

Then, we analyze the loop from line 7 to line 21. The number of mergings performed at line 10 is bounded by $\mathcal{O}(|V|)$, and after each merging, $\mathcal{O}(k^2\frac{|V|}{|E|})$ PQ push operations are executed from line 14 to line 16. Therefore, from line 7 to line 21, the total number of new merging candidates being pushed into the PQ is bounded by $\mathcal{O}(k^2\frac{|V|^2}{|E|})$. Considering there are $\mathcal{O}(k^2\frac{|V|^2}{|E|})$ merging candidates in the initial PQ, the PQ size then can be bounded by $\mathcal{O}(k^2\frac{|V|^2}{|E|})$ at any time during Alg. 2 execution and the total time taken by PQ pop at line 8 can be bounded by $\mathcal{O}(k^2\frac{|V|^2}{|E|}log(k^2\frac{|V|^2}{|E|}))$. Since each attraction evaluation (Eq. (2.4)) at line 15 takes $\mathcal{O}(k)$ time and each PQ push at line 16

takes $\mathcal{O}(log(k^2\frac{|V|^2}{|E|}))$ time, the loop from line 14 to line 16 has time complexity of $\mathcal{O}(k^2\frac{|V|}{|E|}(k + log(k^2\frac{|V|^2}{|E|})))$. By assuming any pair of merging clusters shares $\mathcal{O}(1)$ common nets, the PQ update at line 20 and line 21 needs to be performed $\mathcal{O}(k^2\frac{|V|^2}{|E|^2})$ times for each c_{ij}. So the time complexity of the loop from line 17 to line 21 is $\mathcal{O}(k^2\frac{|V|^2}{|E|^2}(k + log(k^2\frac{|V|^2}{|E|})))$. Combining all these time complexities and the $\mathcal{O}(|V|)$ bound of the execution counts for the two loops (line 14 – 16 and line 17 – 21), the main loop from line 7 to line 21 turns out to have time complexity of $\mathcal{O}((1 + \frac{|V|}{|E|})k^2\frac{|V|^2}{|E|}(k + log(k^2\frac{|V|^2}{|E|})))$.

The final $\mathcal{O}((1 + \frac{|V|}{|E|})k^2\frac{|V|^2}{|E|}(k + log(k^2\frac{|V|^2}{|E|})))$ time complexity of Alg. 2 then can be derived by summing the analysis results of the PQ initialization and the loop from line 7 to line 21.

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

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

BC-based approaches typically could yield very good packing solutions for given attraction functions. However, they have an inherent drawback – 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 unrelated CLB packing. By assigning higher priority to mergings producing larger CLBs, medium-sized CLBs would be promoted quickly. As a result, much tighter packing solutions can be achieved.

Unlike the related CLB packing technique in Alg. 2, where all merging candidates are in one single PQ, we have a separate PQ for each merging size in the unrelated CLB packing. In other words, merging candidates are separated by the number of BLEs in their resulting CLBs, and only mergings result in same BLE count could be placed into the same PQ. The PQ corresponding to larger merging size is grant higher priority and always be processed first.

Within each PQ, BC-based unrelated CLB packing is performed in a manner similar to our related CLB packing. For a CLB, however, instead of considering all its connected CLBs, its K-nearest neighbors (in terms of physical distance) within distance $\overline{\lambda_{uc}}$ would be considered in our unrelated CLB packing. Besides, a different attraction function defined in Eq. (2.5) is used.

$$\phi_{uc}(c_i, c_j) = 1 - e^{\gamma_{uc}(dist(c_i,c_j)-\overline{\lambda_{uc}})}. \tag{2.5}$$

The attraction function ϕ_{uc} is a packing distance penalty factor similar to the first terms of Eq. (2.3) and Eq. (2.4). In UTPlaceF, we set γ_{uc} to 0.2, $\overline{\lambda_{uc}}$ to 8, and K to 30 by default. Note that, although the objective of our unrelated 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 unrelated CLB packing is summarized Alg. 3. Initially, all merging candidates are pushed into a PQ array ($pq[*]$) from line 2 to line 7. The merging candidates of each cluster are added by calling the function `AddKNearestNeighbors` from line 31 to line 42. Each time, among all non-empty PQs in $pq[*]$, the one corresponding to the largest merging size is fetched at line 9. The merging candidate with the highest attraction in the fetched PQ is popped from the PQ top at line 10 and is committed to the netlist from line 12 to line 13. New merging candidates that include the new cluster are pushed into $pq[*]$ by calling function `AddKNearestNeighbors` at line 17. Line 18 to line 27 guarantee that each cluster has merging candidates in $pq[*]$ if legal merging exists. The loop from line 8 to line 29 keeps executing until no more merging candidates exist in $pq[*]$.

The time complexity of Alg. 3 turns out to be $\mathcal{O}(|V|\overline{\lambda_{uc}}^2 + K|V|log(K|V|))$, where V denotes the set of clusters in the netlist and K is the maximum number of neighbors being considered for each cluster during the packing.

It is reasonable to assume that each cluster has average of $\mathcal{O}(K)$ valid mergings candidates in $pq[*]$, since `AddKNearestNeighbors` is only called during $pq[*]$ initialization (at line 4) and when a certain cluster doesn't have any valid merging candidates in $pq[*]$ (at line 14 and line 20). Given this assumption, each merging performed at line 9 would invalidate $\mathcal{O}(K)$ merging candidates in $pq[*]$ on average, which could trigger average of $\mathcal{O}(1)$ calls

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

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

$pq[i] \leftarrow \varnothing, \forall i \in 2, 3, ..., N$;
$valid[u] \leftarrow true, \forall u \in V$;
$numMergings[u] \leftarrow 0 \ \forall u \in V$;
foreach $u \in V$ **do** `AddKNearestNeighbors` (u);
while $\exists i \in 2, 3, ..., N : pq[i] \neq \varnothing$ **do**
  $PQ \leftarrow pq[i]$ where $pq[i] \neq \varnothing$ and $pq[j] == \varnothing \ \forall j \in i+1, ..., N$;
  $(c_i, c_j, \phi_{uc}(c_i, c_j)) \leftarrow$ fetch and pop PQ top;
  if $valid[c_i]$ *and* $valid[c_j]$ **then**
    $c_{ij} \leftarrow \{c_i, c_j\}$;
    $V \leftarrow V \setminus \{c_i, c_j\} \cup \{c_{ij}\}$;
    $valid[c_{ij}] \leftarrow true$;
    $valid[c_i], valid[c_j] \leftarrow false$;
    $numMergings[c_{ij}] \leftarrow 0$;
    `AddKNearestNeighbors` (c_{ij});
    foreach $u \in c_i, c_j$ **do**
      foreach v *such that* $(u, v, \phi_{uc}(u, v)) \in pq[*]$ **do**
        if $valid[v]$ **then**
          $numMergings[v] \leftarrow numMergings[v] - 1$;
          if $numMergings[v] == 0$ **then**
            `AddKNearestNeighbors` (v)

Function `AddKNearestNeighbors(`u`):`
  foreach $v \in \{v \in V | dist(u, v) \leq \overline{\lambda_{uc}}\}$ *sorted by* $dist(u, v)$ **do**
    if $\phi_{uc}(u, v) \geq \underline{\phi_{uc}}$ **then**
      Push $(u, v, \phi_{uc}(u, v))$ into $pq[numBLEs(u \cup v)]$;
      $numMergings[u] \leftarrow numMergings[u] + 1$;
      $numMergings[v] \leftarrow numMergings[v] + 1$;
      if $numMergings[u] == K$ **then return**;

of `AddKNearestNeighbors` at line 20 for clusters that lose all of their valid merging candidates in $pq[*]$. Therefore, `AddKNearestNeighbors` is only called $\mathcal{O}(1)$ times (at line 14 and line 20) on average for each merging performed from line 9 to line 20. Considering the number of merging performed is bounded by $\mathcal{O}(|V|)$ and there are $\mathcal{O}(|V|)$ calls of `AddKNearestNeighbors` during $pq[*]$ initialization at line 4, we can get the following three intermediate bounds: (1) there are total of $\mathcal{O}(|V|)$ function calls of `AddKNearestNeighbors` in Alg. 3; (2) there are total of $\mathcal{O}(K|V|)$ merging candidates being pushed into and popped out of $pq[*]$ in Alg. 3; (3) the size of $pq[*]$ is bounded by $\mathcal{O}(K|V|)$ at any time during Alg. 3 execution.

We first analyze the total time consumed by `AddKNearestNeighbors`. The time complexity of finding the K-nearest feasible neighbors within the distance of $\overline{\lambda_{uc}}$ for a given cluster u can be bounded by $\mathcal{O}(log|V| + \overline{\lambda_{uc}}^2)$, since collecting feasible clusters within the distance of $\overline{\lambda_{uc}}$ to u takes $\mathcal{O}(log|V| + \overline{\lambda_{uc}}^2)$ time if all cluster locations are stored in an R-tree or a k-d tree, and getting the K-nearest neighbors of u in $\mathcal{O}(\overline{\lambda_{uc}}^2)$ clusters can be achieved in amortized $\mathcal{O}(\overline{\lambda_{uc}}^2)$ time. Besides, each of the $\mathcal{O}(K)$ PQ pushes at line 24 takes $\mathcal{O}(log(K|V|))$ time, the time complexity of `AddKNearestNeighbors` turns out to be $\mathcal{O}(\overline{\lambda_{uc}}^2 + Klog(K|V|))$. Considering there are $\mathcal{O}(|V|)$ calls of `AddKNearestNeighbors` in Alg. 3, the total time taken by `AddKNearestNeighbors` then can be bounded by $\mathcal{O}(|V|\overline{\lambda_{uc}}^2 + K|V|log(K|V|))$.

As for non-`AddKNearestNeighbors` parts, the PQ pop at line 6 and

line 7 takes total of $\mathcal{O}(K|V|log(K|V|))$ time by assuming $pq[*]$ contains a constant number of PQs. Since the number of mergings performed is bounded by $\mathcal{O}(|V|)$, the code from line 9 to line 20 can only be executed $\mathcal{O}(|V|)$ times. In each of these $\mathcal{O}(|V|)$ executions, by ignoring the `AddKNearestNeighbors` at line 20, the loop from line 15 to line 20 takes $\mathcal{O}(K)$ time. Therefore, without considering the `AddKNearestNeighbors` at line 14 and line 20, the time complexity of the loop from line 5 to line 20 turns out to be $\mathcal{O}(K|V|log(K|V|) + K|V|)$, which can be simplified to $\mathcal{O}(K|V|log(K|V|))$.

Combining the results of the `AddKNearestNeighbors` part and the non-`AddKNearestNeighbors` parts, we can finally bound the time complexity of Alg. 3 by $\mathcal{O}(|V|\overline{\lambda_{uc}}^2 + K|V|log(K|V|))$.

For high-utilization designs, our default unrelated CLB packing might still not be able to generate tight enough packing solutions that satisfy FPGA capacity constraint. In this case, the existing packing solution will be ripped up, and new related and unrelated CLB packing parameters will be adopted to generate a tighter packing solution in the next packing pass. PCAP would iteratively perform 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 related CLB packing works effectively to reduce the number of external nets, however, it often yields relatively loose packing due to the in-

herent shortcoming of BC mentioned in section 2.3.3.3. In contrast, our unrelated CLB packing is capable of aggressively reducing the number of CLBs and achieving tight packing. Therefore, if more packing is performed in the related CLB packing stage, a loose packing solution with less external nets would be delivered. However, if we only do a small portion of packing in the related CLB packing stage and leave most of the work to unrelated CLB packing, the final packing would be more inclined to the "tight" side with more external nets.

In PCAP, the minimum related CLB packing attraction ($\underline{\phi_{rc}}$) and the maximum unrelated CLB packing distance ($\overline{\lambda_{uc}}$) are used to control the amount of packing work for each (related/unrelated) CLB packing stage. Initially, $\underline{\phi_{rc}}$ is set as 0.1 to aggressively reduce the number of external nets, and $\overline{\lambda_{uc}}$ is set as 8 to only allow close packing in unrelated 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 number of CLBs exceeds the FPGA capacity. To address this problem, PCAP would discard the existing CLB packing solution (but respect BLE packing solution) and perform a repacking step, which applies related and unrelated CLB packing again. In the repacking phase, however, $\underline{\phi_{rc}}$ is increased to reduce related CLB packing, and $\overline{\lambda_{uc}}$ is also increased to allow unrelated CLB packing of longer distance. As results, the repacking step would achieve tighter packing but sacrifice net reduction. The repacking step is repeated until the CLBs

utilization target is satisfied.

Algorithm 4: CLB Packing and Rip-up and Repacking

Input : BLE packing is done. Maximum FPGA CLB utilization U_{max}, maximum unrelated CLB packing distance incrasing rate $\beta_{\overline{\lambda_{uc}}}$, and minimum related CLB packing attraction increasing rate $\Delta\underline{\phi_{rc}}$.

Output: Maximum FPGA CLB utilization constraint is satisfied.

```
while true do
    Perform related CLB packing;
    Perfrom unrelated CLB packing;
    if CLB utilization ≤ U_max then return;
    λ_uc ← λ_uc · β_λ_uc;
    φ_rc ← φ_rc + Δφ_rc;
end
```

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

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 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 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

Fig. 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 global placement.

2.3.4.2 Min-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 min-cost-max-cardinality bipartite matching problem with pin movement as cost. By solving the corresponding bipartite matching problem, global placement can be legalized with 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 min-cost bipartite matching for each partition. To further speed up runtime, edges in bipartite graphs are pruned based on Manhattan distance and are incrementally added when necessary.

Our legalization approach is summarized in Alg. 5. The placement region is partition 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 make sure each min-cost bipartite matching has enough sites to accommodate all cells, neighbor available sites are added when necessary from

Algorithm 5: Min-Cost Bipartite Matching Based Legalization

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

Output: Legalized placement with minimum total pin movement.

```
1  Partition the placement region into a set of rectangle regions P;
2  foreach p ∈ P do
3      L ← unlegalized CLBs in p;
4      R ← unoccupied sites in p;
5      while |L| > |R| do
6          Add the closest unoccupied site to p into R;
7      Construct a |L| × |R| bipartite graph g;
8      d_min ← 0;
9      d_max ← D_max;
10     while true do
11         foreach l ∈ L, r ∈ R do
12             if d_min ≤ dist(l, r) < d_max then
13                 cost ← dist(l, r) · numPins(l);
14                 Add edge (l, r, cost) into g;
15         Run min-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 ← d_max;
23         d_max ← d_max + ΔD;
```

line 5 to line 6. The bipartite graph is constructed at line 7 with all the cells as left vertices and all the sites as right vertices in the graph. 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 min-cost bipartite matching is solved at line 15, and cells are moved 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 respectively, initial maximum displacement D_{max} to 4, and maximum displacement increasing rate ΔD_{max} to 2.

Similar to CLBs, heterogeneous blocks like DSPs, RAMs, and I/Os also have the regularity of sizes, so they are legalized separately using Alg. 5 with minor variations in UTPlaceF as well.

The time complexity of Alg. 5 is $\mathcal{O}(\frac{|V|^3}{|P|^2} log \frac{|V|}{|P|} log(\frac{|V|}{|P|}C))$, where V denotes the set of cells, P denotes the set of partitions, and C is the maximum cost value returned at line 13.

The number of cells in each partition is $\mathcal{O}(\frac{|V|}{|P|})$ on average. By assuming the loop from line 5 to line 6 can also be finished in $\mathcal{O}(\frac{|V|}{|P|})$ time, the bipartite graph initialization from line 3 to line 7 can be done in $\mathcal{O}(\frac{|V|}{|P|})$ time. Since the edge pruning complicates the complexity analysis, here we only consider the worst case, where $d_{min} = 0$ and $d_{max} = infinity$ at line 8 and line 9, respectively. Given this assumption, each bipartite graph has $\mathcal{O}(\frac{|V|}{|P|})$ vertices and $\mathcal{O}(\frac{|V|^2}{|P|^2})$ edges. As results, the time complexity of the edge

construction from line 11 to line 14 is $\mathcal{O}(\frac{|V|^2}{|P|^2})$. By applying network simplex algorithm [3], the min-cost bipartite matching at line 15 can be solved in $\mathcal{O}(\frac{|V|^3}{|P|^3} log \frac{|V|}{|P|} log(\frac{|V|}{|P|}C))$ time. After that, moving cells to their matched sites from line 16 to line 21 takes $\mathcal{O}(\frac{|V|^2}{|P|^2})$ time. Assembling all pieces together, the time complexity of legalizing one partition can be bounded by $\mathcal{O}(\frac{|V|^3}{|P|^3} log \frac{|V|}{|P|} log(\frac{|V|}{|P|}C))$ and the total time complexity of handling $|P|$ partitions turns out to be $\mathcal{O}(\frac{|V|^3}{|P|^2} log \frac{|V|}{|P|} log(\frac{|V|}{|P|}C))$.

It should be noted that, in practice, the number of edges in each bipartite graph is far less than $\mathcal{O}(\frac{|V|^2}{|P|^2})$ due to our pruning technique, so the empirical runtime of Alg. 5 is much faster than the above theoretical complexity bound.

2.3.4.3 Congestion-Aware Hierarchical Independent Set Matching

The idea of bipartite matching can also be applied to optimize wirelength. For a given set of legalized cells, a wirelength optimization problem can be formulated as a min-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 [12] 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.

In UTPlaceF, ISM is hierarchically applied to CLBs, BLEs and LUT

pairs. One main objective of our packing stage (PCAP) is to absorb small nets into clusters (BLEs, CLBs). Therefore, most CLBs essentially are clusters of LUTs and FFs that have strong connectivity. One of our key observation is that moving cells with strong connectivity 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 clusterd only based on physical distance and connectivity but are not aware of wirelength. Thus ISM for BLEs and LUT pairs are introduced to fix our CLB packing and BLE packing respectively after CLB level ISM.

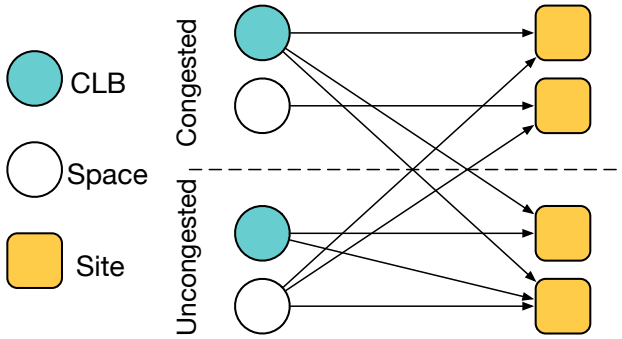


Figure 2.10: Illustration of our congestion-aware ISM.

The ISM works effectively for optimizing HPWL. However, it could ruin the local cell density optimized for routability, especially when spaces are considered 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 disallowed. Fig. 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 Alg. 6. Each independent set is generated at line 2 by calling function `GenerateIndepSet` from line 16 to line 29. The bipartite graph is constructed from line 3 to line 9. The cost of each edge is calculated by function `GetMovingCost` from 31 to 39, the extra routability rules are applied here as well. The min-cost bipartite matching is solved at line 10, and cells are moved to their matched locations from line 11 to line 13. This algorithm is sequentially executed for CLBs, BLEs, and LUT pairs to successively optimize wirelength in different levels. Note that for BLE ISM, the clock legality of CLBs must be preserved, so each independent set can only contain BLEs belong to the same clock net (and may also contain BLEs without clocks). In UTPlaecF, U_{th} is set to 0.7, N_{is} is set to 50, and D_{is} is set to 10 by default.

The time complexity of Alg. 6 turns out to be $\mathcal{O}(|V|(D_{is}^2 + k^2\frac{|V|}{|E|}N_{is} + kN_{is}^2 + N_{is}^3 log N_{is} log(N_{is}C)))$, where V and E are the set of cells and nets, respectively, k is the average number of pins in each cell in V, and C is the maximum non-infinity cost returned by `GetMovingCost`.

In each call of `GenerateIndepSet`, the number of iterations of the loop from line 13 to line 18 is bounded by $\mathcal{O}(D_{is}^2)$, but no more than N_{is} of them can execute the loop from line 17 to line 18. In addition, for each of these N_{is} iterations, line 18 is executed for $\mathcal{O}(k^2\frac{|V|}{|E|})$ times, since, on average,

Algorithm 6: Congestion-Aware Independent Set Matching

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

Output: Legalized placement with shorter wirelength and no routability degradation.

```
1  foreach u ∈ V do
2  |  S ← GenerateIndepSet (u);
3  |  Construct a |S| × |S| min-cost bipartite graph g;
4  |  foreach l ∈ left vertex set of g do
5  |  |  foreach r ∈ right vertex set of g do
6  |  |  |  cost ← GetMovingCost (l, r);
7  |  |  |  Add the edge (l, r, cost) into g;
8  |  Run min-cost bipartite matching on g;
9  |  foreach matched edge (l, r) do
10 |  |  Move l to r's location;
11 Function GenerateIndepSet(s):
12 |  isIndep[u] ← true ∀u ∈ V;
13 |  foreach u ∈ {s} ∪ {x ∈ V|dist(s, x) ≤ D_is} do
14 |  |  if isIndep[u] then
15 |  |  |  S ← S ∪ {u};
16 |  |  |  if |S| ≥ N_is then return S;
17 |  |  |  foreach v connected to u do
18 |  |  |  |  isIndep[v] ← false;
19 Function GetMovingCost(l, r):
20 |  if l is a cell and routing utilization at r > U_is then
21 |  |  return infinity;
22 |  if l is a white space and routing utilization at l > U_is then
23 |  |  return infinity;
24 |  return HPWL increase of moving l to r's location;
```

each cell is connected to k nets and each net have $k\frac{|V|}{|E|}$ cells. Thus, each `GenerateIndepSet` can be finished in $\mathcal{O}(D_{is}^2 + k^2\frac{|V|}{|E|}N_{is})$ time.

In our implementation, each call of `GetMovingCost` takes amortized $\mathcal{O}(k^2\frac{|V|}{|E|N_{is}} + k)$ time. During each bipartite matching run, the HPWL of $\mathcal{O}(kN_{is})$ nets could be changed. We first use brute-force approach to pre-compute the bounding boxes of these $\mathcal{O}(kN_{is})$ nets without considering cells in the independent set S, which takes $\mathcal{O}(k^2\frac{|V|}{|E|}N_{is})$ time. Then, for each of these nets, the HPWL change of moving any cell in S can be obtained in $\mathcal{O}(1)$ time. Consequently, the total HPWL change of moving any cell in S can be computed in $\mathcal{O}(k)$ time. Note that, in each bipartite graph construction from line 3 to line 7, the precomputation for net bounding boxes only needs to be done once but `GetMovingCost` are called $\mathcal{O}(N_{is}^2)$ times, so the amortized time complexity of `GetMovingCost` is $\mathcal{O}(k^2\frac{|V|}{|E|N_{is}} + k)$. The time complexity of each bipartite graph construction from line 3 to line 7 then can be bounded by $\mathcal{O}(k^2\frac{|V|}{|E|}N_{is} + kN_{is}^2)$ time.

By applying network simplex algorithm [3], each min-cost bipartite matching at line 8 can be solved in $\mathcal{O}(N_{is}^3 log N_{is} log(N_{is}C))$ time. After that, moving cells to their matched sites (line 9 – 10) can be done in $\mathcal{O}(N_{is}^2)$ time.

Combining all the analysis results, the time complexity of each iteration of the main loop (line 1 – 10) is $\mathcal{O}(D_{is}^2 + k^2\frac{|V|}{|E|}N_{is} + kN_{is}^2 + N_{is}^3 log N_{is} log(N_{is}C))$. Considering the main loop are executed $\mathcal{O}(|V|)$ times, we can conclude that the time complexity of Alg. 6 is $\mathcal{O}(|V|(D_{is}^2 + k^2\frac{|V|}{|E|}N_{is} + kN_{is}^2 + N_{is}^3 log N_{is} log(N_{is}C)))$.

2.3.5 Experimental Results

UTPlaceF was implemented in C++ and tested on a Linux machine with 3.40 GHz CPU and 32GB RAM. The benchmark suite released by Xilinx for ISPD'16 FPGA placement contest was used to validate the effectiveness of UTPlaceF. Related executables, placement solutions, and benchmarks are released at link (`http://wuxili.net/project/utplacef/`).

2.3.5.1 Benchmark Characteristics

Table 2.1: ISPD'16 Placement Contest Benchmarks Statistics

Benchmark	#LUT	#FF	#RAM	#DSP	#Ctrl Set
FPGA-1	50K	55K	0	0	12
FPGA-2	100K	66K	100	100	121
FPGA-3	250K	170K	600	500	1281
FPGA-4	250K	172K	600	500	1281
FPGA-5	250K	174K	600	500	1281
FPGA-6	350K	352K	1000	600	2541
FPGA-7	350K	355K	1000	600	2541
FPGA-8	500K	216K	600	500	1281
FPGA-9	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 ISPD'16 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 to yield

Table 2.2: Routed Wirelength (WL in 10^3) and Runtime (RT in seconds) Comparison with ISPD'16 Contest Winners on ISPD 2016 Benchmark Suite

Design	1st Place		2nd Place		3rd Place		UTPlaceF	
	WL	RT	WL	RT	WL	RT	WL	RT
FPGA-01	PE*	N/A	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	UR	N/A	UR	N/A	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	UR†	N/A	13909	5833	UR	N/A	12369	2537
FPGA-10	PE	N/A	PE	N/A	UR	N/A	8795	3182
FPGA-11	11091	3227	11713	7331	UR	N/A	10196	2151
FPGA-12	9022	4539	PE	N/A	UR	N/A	7755	2944
Norm.	1.062	1.55	1.116	2.30	1.291	3.10	1.000	1.00

* PE: Placement error

† UR: Unroutable placement

tight packing solutions to satisfy the CLB capacity constraint, and 2) the capability to reduce 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 ISPD'16 placement contest and other state-of-the-art FPGA placers. The results are shown in Table 2.2 and Table 2.3. All routed wirelength are reported by Xilinx Vivado v2015.4, and runtime of the contest winners are evaluated on a Linux Machine with 3.20 GHz CPU and 32GB RAM. Normalized results in the last row of

Table 2.3: Routed Wirelength (WL in 10^3) and Runtime (RT in seconds) Comparison with State-of-the-Art Academic FPGA Placers on ISPD 2016 Benchmark Suite

Design	[40] *		RippleFPGA [65]		GPlace [64]		UTPlaceF	
	WL	RT	WL	RT	WL	RT	WL	RT
FPGA-1	385	215	363	74	494	30	357	185
FPGA-2	653	399	678	167	903	61	642	305
FPGA-3	3181	1555	3617	1037	3908	289	3215	831
FPGA-4	5504	1289	6037	621	6278	280	5410	824
FPGA-5	10069	1237	10455	1012	UR	N/A	9660	1237
FPGA-6	6411	2827	6960	2772	7643	600	6488	1041
FPGA-7	10041	2588	10248	2170	11255	691	10105	1721
FPGA-8	8113	2705	8874	1426	9323	734	7879	1686
FPGA-9	13617	3407	12954	2683	14003	974	12369	2537
FPGA-10	8866	4091	8564	5555	UR	N/A	8795	3182
FPGA-11	10835	3267	11226	3636	12368	923	10196	2151
FPGA-12	8246	4625	8929	9748	UR	N/A	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 on ICCAD'16.

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, [40], RippleFPGA [65], and GPlace [64] respectively. It should be noted that only UTPlaceF and RippleFPGA are able to route all 12 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.

Table 2.4: Runtime Breakdown of UTPlaceF

Design	FIP	PCAP			GP	LG	ISM			Others	Total
		BLE	Rel. CLB	Unrel. CLB			CLB	BLE	LUT Pair		
FPGA-1	115	1	2	3	9	1	5	26	22	1	185
FPGA-2	187	2	4	4	17	1	8	38	42	2	305
FPGA-3	528	5	12	13	58	1	28	57	120	9	831
FPGA-4	500	6	12	17	56	2	33	61	127	10	824
FPGA-5	663	7	13	23	77	3	41	67	136	11	1041
FPGA-6	1029	8	44	171	101	30	49	77	197	15	1721
FPGA-7	974	9	52	195	126	19	54	81	205	16	1731
FPGA-8	1004	9	26	39	91	16	49	174	274	4	1686
FPGA-9	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

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-1	407	394	359	357
FPGA-2	702	677	647	642
FPGA-3	3339	3264	3236	3215
FPGA-4	5603	5484	5433	5410
FPGA-5	9908	9697	9641	9660
FPGA-6	6824	6686	6613	6488
FPGA-7	10500	10322	10253	10105
FPGA-8	8156	7992	7913	7879
FPGA-9	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

2.3.5.3 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 respectively take 16.2%, 5.1% and 21.4% of the total runtime, 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, related CLB packing takes 2.3% of the total runtime, and the remaining 13.4% is taken by the unrelated CLB packing. In hierarchical ISM, CLB, BLE, and LUT-pair level ISM respectively take 2.6%, 6.7%, and 12.1% of the total runtime.

2.3.5.4 Congestion-Aware Hierarchical Independent Set Matching Effectiveness Validation

To show the effectiveness of our proposed congestion-aware hierarchical ISM, we compared the routed wirelength of the intermediate placement solution after each ISM steps. Table 2.5 summarizes the experimental 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.

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 paper, we have proposed a routability-driven FPGA packing and placement engine called UTPlaceF. A novel packing algorithm PCAP and a 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'16 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, *Place-Pack-Place-Legalize* methodologies currently dominate the state-of-the-art industrial FPGA CAD tools [70]. 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 only seeks to optimize the 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 the aforementioned 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 a initial placement by incorporating the previously separated packing and final placement/legalization steps. Our experiments show that the overall implementation quality, as well as the correlation between FIPs and final legal solutions, can be significantly improved with the proposed flow. Our major contributions are highlighted as follows:

- We present a new paradigm 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 ISPD 2016 contest as well as three state-of-the-art academic placers UTPlaceF [41], RippleFPGA [11], and GPlace [64] in routed wirelength with competitive runtime on ISPD 2016 benchmark suite [81].

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
$\mathbf{x}', \mathbf{y}'$	The x and y coordinates of cells in $\mathcal{V}$ in FIP
$\mathbf{x}, \mathbf{y}$	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 a FIP. Given the notations defined in Table 2.6, the FPGA DL problem can be defined as follows:

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

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

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

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

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 $\lambda \cdot \text{HPWL}(\mathbf{x}, \mathbf{y})$ 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 affinity functions used in conventional packing algorithms [60, 68, 69]. 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 one

and only one CLB slice. The constraint (2.6c) assures that all the architecture 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 have 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 near as possible to a legal solution. Otherwise, the final legal solution cannot be obtained with a small placement perturbation. In a 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 previous works, cell areas were set statically based on some empirical estimations [11, 64]. 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

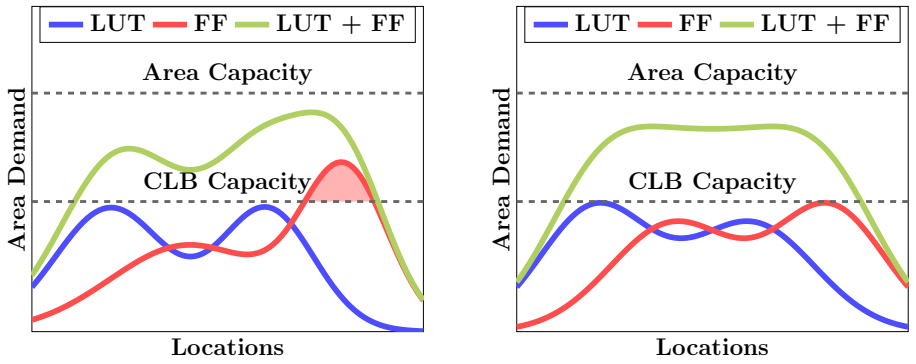


Figure 2.11: (a) An area overflow-free placement but with FF demand overflow (the shaded red region). (b) A legal placement.

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 rough-legal placement, overlapping removal techniques (e.g., rough legalization) distribute cells by evening out area demand 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 Fig. 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 Fig. 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 [28]. 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 sufficently 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 proposed overall flow is illustrated in Fig. 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 area ad-

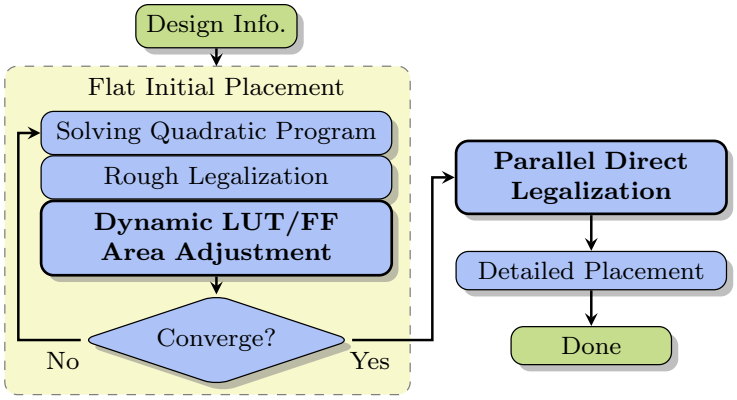


Figure 2.12: The proposed overall flow.

justment step is performed after each placement iteration. This new step is aiming 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.

2.4.3.2 Dynamic LUT/FF Area Adjustment

Since packing and cell utilization calculation are both very local-scoped in nature, it is reasonable to analyze them in a small spatial neighborhood contexts 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, and (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.

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 $(x_v - L, y_v - L, x_v + L, y_v + L)$ 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 (Eq. (2.9)) and Section 2.4.3.2.3 (Eq. (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 [11,41]. The routing congestion is estimated by a fast global router NCTUgr [54], 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 the proposed flow, we perform 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 (Eq. (2.9) and Eq. (2.10)) can result in low resource usage. This issue is prevented by scaling A_v using the local resource utilization U_v in Eq. (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 Fig. 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 (Eq. (2.7)) maps of a design with/without this technique when area constraint is satisfied (see Fig. 2.11). It can be seen that, with this technique, a huge FF hotspot (Fig. 2.13(b)) is resolved (Fig. 2.13(d)). More interestingly, the LUTs that are original in the FF hotspot (Fig. 2.13(a)) tend to follow the move-

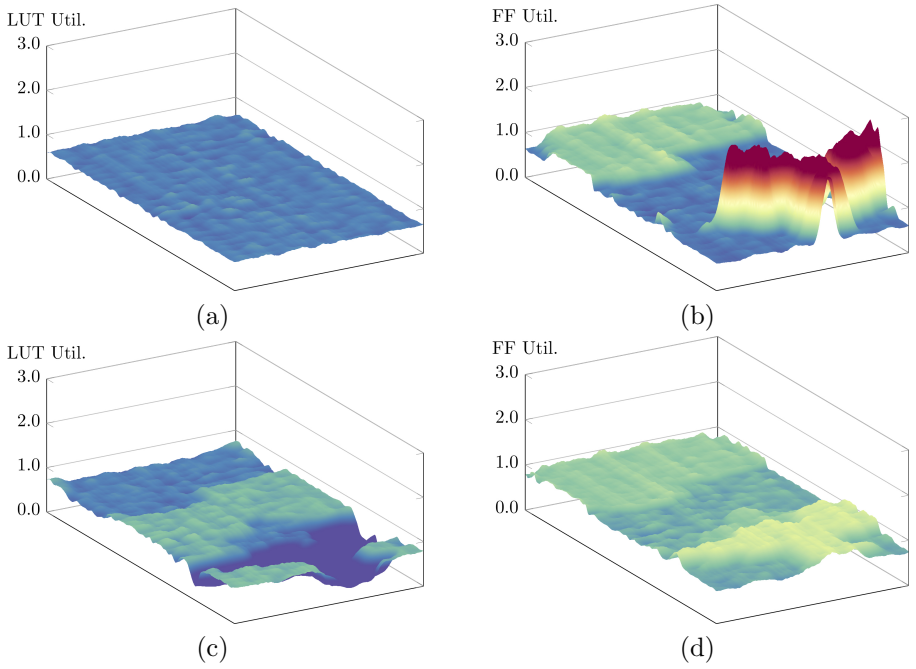


Figure 2.13: (a) and (b) are the 3-D LUT and FF utilization (Eq. (2.7)) maps without our dynamic area adjustment. (c) and (d) are the ones with it applied. The area constraint is satisfied in both placement solutions. This is an example of the challenging case illustrated in Fig. 2.11. The experiments are based on design *FPGA-10* in ISPD 2016 benchmark suite [81].

ment of their connected FFs, which results in a "valley" (Fig. 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 Eq. (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 a FIP, a probabilistic estimation defined in Eq. (2.9) is used instead for a LUT v with its neighbors $\mathcal{N}_v$.

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

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 Eq. (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 Eq. (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*[1]. For instance, in Fig. 2.2b, 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 estimate the lower-bound FF demands can be illustrated by Fig. 2.14. Using the (CK, SR) group of (B, P) in Fig. 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 $\{CE_0, CE_1, \ldots, CE_m\}$ be the set of CE nets in $\mathcal{N}_v^+$. If we

[1]The quarter/half CLBs here only contain FFs, since LUTs are irrelevant in this subsection.

denote the number of FFs in $\mathcal{N}_v^+$ with the control set (CK_0, SR_0, CE_i) as n_i, for $0 \leq i \leq m$, the estimated FF demand of v can be expressed as follows:

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

Note that our analysis in Fig. 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 considering net sharing and other optimization metrics. Therefore, we introduce $\alpha^{(\mathrm{FF})} \geq 1$ in Eq. (2.10) to give more flexibility to the packing. In our framework, we empirically set $\alpha^{(\mathrm{FF})}$ to 1.1.

If we ignore the $\alpha^{(\mathrm{FF})}$ term, the FF demand estimated by Eq. (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 Fig. 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 Fig. 2.13. Therefore, our FF demand estimation method detailed in this section is essential for the proposed *Place-Legalize* flow.

It is worthwhile to mention that, although the LUT/FF resource demand estimations 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.

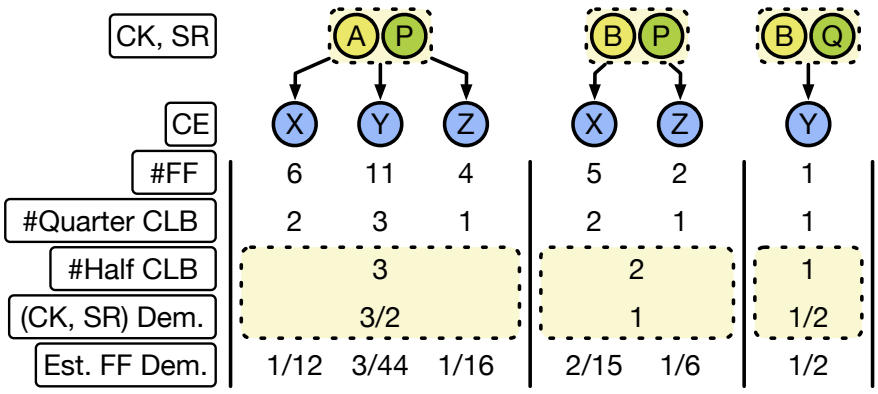


Figure 2.14: Illustration of FF resource demand estimation.

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 (Eq. (2.9) and Eq. (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 rough-legal FIP and produces a legal solution by solving the Formulation (2.6a) – (2.6d). The idea is partially similar to the clustering algorithms in [59, 70], but we are aiming at a legal placement directly. Besides, the cell displacement is explicitly considered in our formulation. Compared to traditional greedy methods, our method can explore a significantly larger solution space by exploiting the strength of parallelism.

For the simplicity, we will use "slice" to denote "CLB slice" in this section.

2.4.3.3.1 A College Admission Analogy

The proposed DL algorithm is inspired by the *College Admission Problem* [22]. One can think that each 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 a 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. "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 Fig. 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

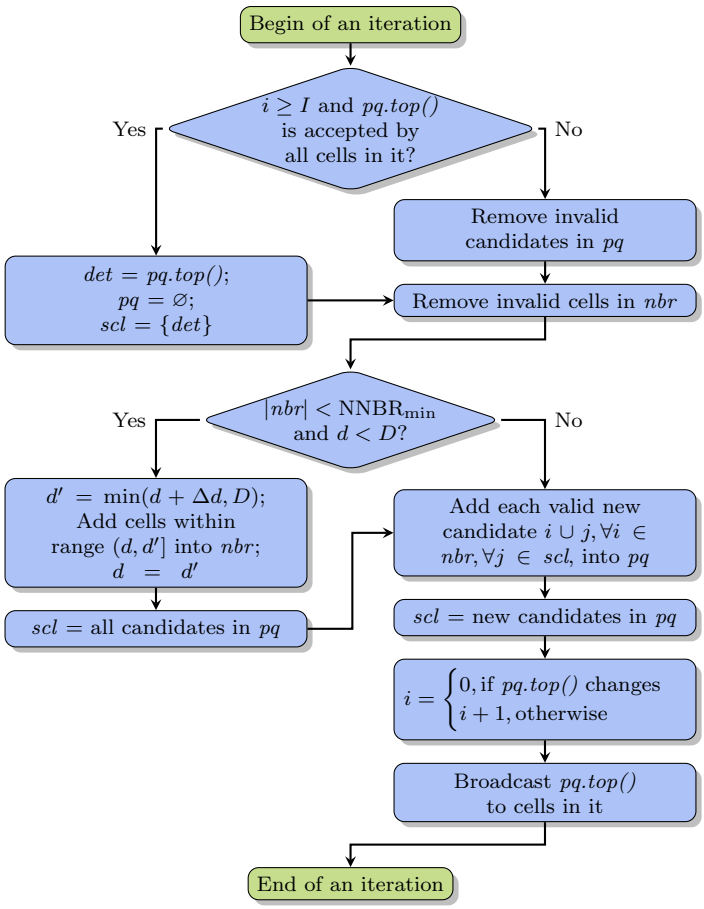


Figure 2.15: The node-centric DL algorithm flow at each computation node (slice).

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 $i = 0$, *det* $= \varnothing$, $pq = \varnothing$, *nbr* as the set of cells within a predefined distance d (default is 1) to the slice, and *scl* to contain an empty cluster without any cells.

In every DL iteration, as shown in Fig. 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*, respectively. 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* ($|nbr| < \text{NNBR}_{\text{min}}$), the current maximum cell displacement constraint d will be relaxed by Δd, and more neighbor cells will be added to guarantee the scope of solution space we can explore. Besides, we will also copy all candidates in *pq* to *scl* to guarantee that they will 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. 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 the 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

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

where 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, ΔSCORE(s) is the amount of score improvement by committing the *pq.top()* in slice s.

Algorithm 7 summarizes the whole fully parallelizable DL process. All computation nodes are initialized in parallel in line 1 – 6. In each DL iteration, the node-centric algorithm presented in Fig. 2.15 is executed by each slice individually in line 8 – 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 (Eq. (2.11)) and inform its acceptance to the selected slice in line 10 – 11. The loop from line 7 to line 11 is repeated until no more valid candidate exists.

Algorithm 7: Fully Parallelizable Direct Legalization

Input : A rough-legal placement, the set of cells $\mathcal{C}$, the set of slice (computation node) $\mathcal{S}$, and the initial maximum displacement constraint d.

Output: A legal placement.

```
1  parallel foreach s ∈ S do
2  |  s.i ← 0;
3  |  s.det ← ∅;
4  |  s.pq ← ∅;
5  |  s.nbr ← {c ∈ C | dist(c, s) ≤ d};
6  |  s.scl ← {an empty cluster};
7  while exists valid pq.top() for s ∈ S do
8  |  parallel foreach s ∈ S do
9  |  |  Run the node-centric algorithm shown in Fig. 2.15;
10 |  parallel foreach c ∈ C do
11 |  |  Among {s ∈ S | c ∈ s.pq.top()}, pick s* that
   |  |   has the highest ΔSCORE(s*) defined in Eq. (2.11)
   |  |   and inform s* that c accepts its pq.top();
```

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 (Eq. (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.

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:

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

where Net(c) denotes the set of nets that have at least one cell in c, TotalPins(e) denotes the total pin count of net e, InternalPins(e, c) represents the number of pins of net e in c, and ΔHPWL(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 Eq. (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 Fig. 2.15 perfectly in parallel in each DL iteration (Algorithm 7 line 9 – 11). Moreover, cells can also process the results generated by slices and send back their decisions individually (Algorithm 7 line 12 – 16).

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 near-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 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 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. 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 question 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

$$\text{SCORE}_{\text{ripup}}(v, s, c) = -\lambda_1 \cdot \Delta\text{HPWL}(v, s) - \lambda_2 \cdot \text{SCORE}(c, s) - \lambda_3 \cdot \text{Area}(c), \tag{2.13}$$

where ΔHPWL(v, s) denotes the HPWL increase of moving v to s, SCORE(c, s) denotes the score of c in s as defined in Eq. (2.12), and 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 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 Eq. (2.9) and Eq. (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 main loop (line 1 – 4), each illegal cell c is legalized with the maximum displacement constraint $D^{(c)}$ using function `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 a illegal cell c with displacement constraint D using `Legalize`(c, D) (line 5 – 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 Eq. (2.13) in descending order (line 7). After that we try to ripup $s \in ls^{(c)}$ one by one and legalize c using function `RipUpAndLegalize`(s, c, D) until the legalization successes (line 8 – 10). If c cannot be legalized by breaking any slice in $ls^{(c)}$, `Legalize`(c, D) will return *fail* (line 11) and the displacement constraint will be relaxed in the main loop (line 3 – 4).

The details of function `RipUpAndLegalize`(s, c, D) are given in line 12 – 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

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.

foreach $c \in \mathcal{C}'$ **do**
  $D^{(c)} \leftarrow D$;
  while `Legalize` *$(c, D^{(c)})$ is fail* **do**
    $D^{(c)} \leftarrow D^{(c)} + 1$;

Function `Legalize`$(c,\ D)$**:**
  $ls^{(c)} \leftarrow \{s \in \mathcal{S} \mid \text{dist}(c, s) \leq D\}$;
  Sort slices in $ls^{(c)}$ by their $\text{SCORE}_{\text{ripup}}$ defined in Eq. (2.13) in descending order;
  foreach $s \in ls^{(c)}$ **do**
    if `RipUpAndLegalize` *(s, c, D) is success* **then**
      return success;
  return fail;

Function `RipUpAndLegalize`(s, c, D)**:**
  $lv \leftarrow \{v \in \mathcal{C} \mid v \in s.det\}$;
  $s.det \leftarrow \{c\}$;
  foreach $v \in lv$ **do**
    $ls^{(v)} \leftarrow \{s \in \mathcal{S} \mid \text{dist}(v, s) \leq D$ and $s.det \cup v$ is legal$\}$;
    if $ls^{(v)}$ *is* $\varnothing$ **then**
      Remove c from $s.det$;
      Put all $v \in lv$ back to $s.det$;
      return fail;
    else
      Pick $s^\star \in ls^{(v)}$ with the highest $\text{SCORE}(s^\star.det \cup v,\ s^\star) - \text{SCORE}(s^\star.det,\ s^\star)$;
      $s^\star.det \leftarrow s^\star.det \cup v$;
  return success;

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 with v (line 16). If no such slice exists, we discard all the changes that have been made in this function call (line 18 – 19) and return *fail* (line 20). Otherwise, among all candidate slices ($ls^{(v)}$), we pick the one with the highest score gain SCORE($s^{\star}.det \cup v$, $s^{\star}$) − SCORE($s^{\star}.det$, $s^{\star}$) and put v into it (line 22 – 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 implemented UTPlaceF-DL in C++ based on UTPlaceF [41] and performed 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 [1] is used to support multi-threading. The benchmark suite

released by Xilinx for ISPD 2016 FPGA placement contest [81] is used to validate the the effectiveness of the proposed approaches. All the routings are conducted by Xilinx Vivado v2015.4 [74]. The characteristics of ISPD 2016 benchmark suite are listed in Table 2.7.

2.4.4.1 Effectiveness Validation of Proposed Techniques

Table 2.8 domonstrates 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. Column "UTPlaceF" represents the original UTPlaceF [41] flow. Column "UTPlaceF + DAA" applys DAA on top of the original UTPlaceF. 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 implemented a greedy Tetris-based direct legalization (greedy DL) for comparison. This greedy DL adopts the same score function Eq. (2.12) used by the proposed DL, and it legalizes one cell at a time to maximize the score improvement defined in Eq. (2.11). The results of substituting the proposed DL in "UTPlaceF-DL" with this greedy DL are shown in 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, so that 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 results in very suboptimal or even illegal solutions. In the proposed DL, all cells simultaneously seek to legalize themselves. Without DAA, however, the FIP solution can be considerably 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.

Table 2.8: Routed Wirelength (WL in 10^3) and Runtime (RT in Seconds) Comparison with UTPlaceF† [41]

Design	UTPlaceF† [41]				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 [41].

We first compare 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 experiemental results shown in Table 2.9, which we will discuss in details later in this section.

We then compare 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 explores a larger solution space and achieves better solution quality.

By comparing 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× 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, our approach outperforms UTPlaceF by 29.5% in routed wirelength. Besides, our approach also consistently excel 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.

Another notable merit of our approach is that the correlation between FIPs and post-legalization solutions can be significantly improved. This property is appealing in a 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 proposed 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

Table 2.9: Displacement Comparison with UTPlaceF

Design	UTPlaceF [41]		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

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 shows the impact of DAA on HPWL and the maximum LUT and FF resource utilization (Eq. (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 Fig. 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

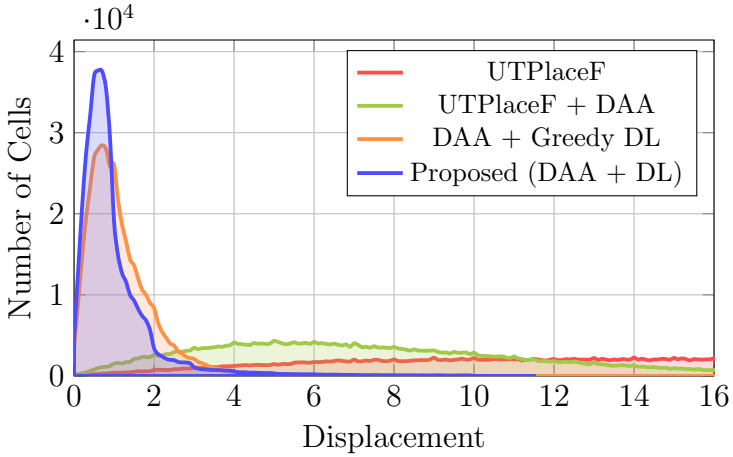


Figure 2.16: The cell displacement distributions of UTPlaceF, UTPlaceF + DAA, DAA + Greedy DL, and the proposed flow (DAA + DL) based on design *FPGA-03*. The displacement of a cell is defined as the Manhattan distance between its locations in the FIP and the post-legalization/DL placement. The average/maximum 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.

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.

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 Eq. (2.8), should be set.

Table 2.10: HPWL and the Maximum LUT and FF Utilizations (Eq. (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

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 Fig. 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. This is because that, 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

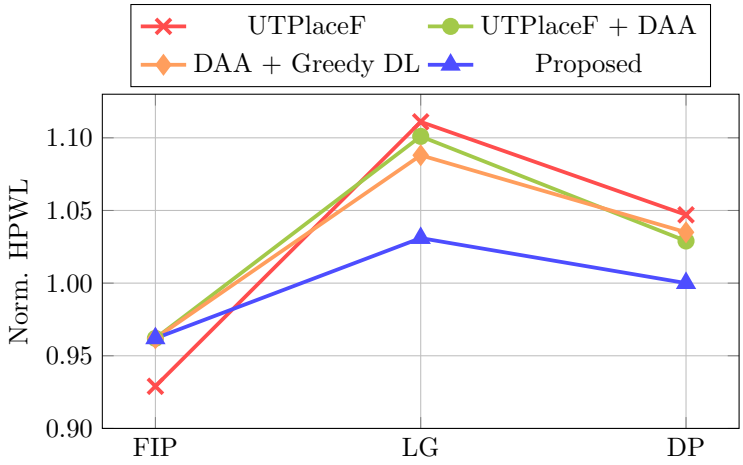


Figure 2.17: The normalized HPWL after the flat initial placement (FIP), legalization/DL (LG), and detailed placement (DP) in the four methodologies listed in Table 2.8. All HPWL values are normalized to the post-DP HPWL of the proposed flow.

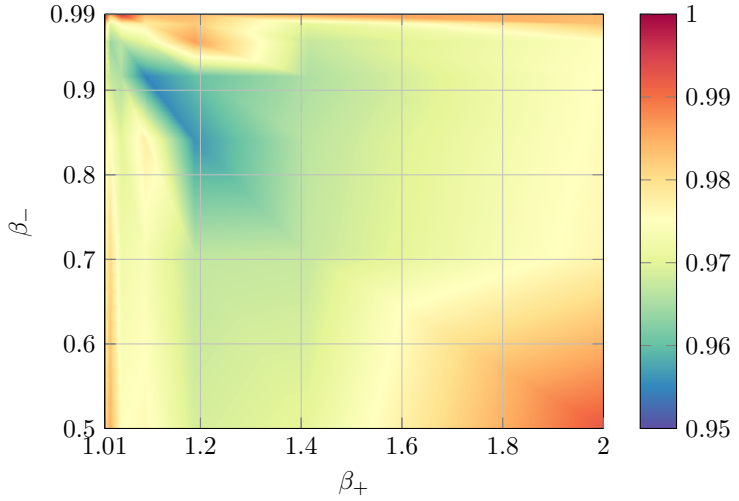


Figure 2.18: Normalized HPWL under different β_+ and β_- in Eq. (2.8) based on design *FPGA-01*. All wirelengths are normalized to the solution without applying dynamic area adjustment.

$\beta_+ = 1.1$ and $\beta_- = 0.95$, and we observe that the final solution quality is not very sensitive to settings around these two values.

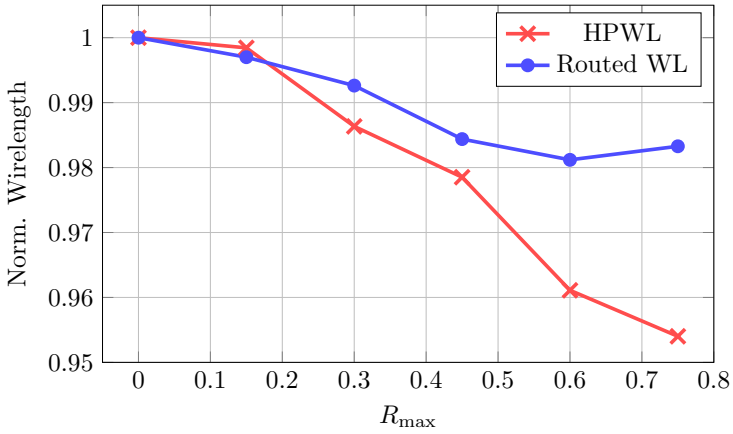


Figure 2.19: Normalized HPWL and routed wirelengths under different $R_{\max}$ in Eq. (2.8) based on design *FPGA-05*. All wirelengths are normalized to the solution with $R_{\max} = 0$ (i.e. area shrinking is disallowed).

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 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 a 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

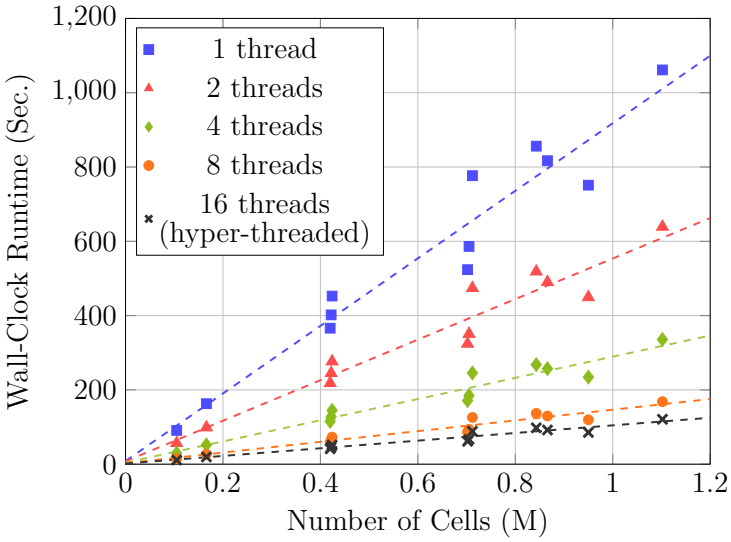


Figure 2.20: Runtime scaling of the direct legalization.

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 hyper-threading). On average, 1.65×, 3.15×, 6.19×, and 8.68× speedups can be achieved with 2, 4, 8, and 16 threads, respectively, compared with the single-thread execution. Note that the sacling starts to saturate from 8 threads to 16 threads. This is because a CPU core can lanuch 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 our approach, we also compare our result with other state-of-the-art academic placers, including RippleFPGA [11], GPlace [64] as well as the top-3 winners of ISPD 2016 contest, on ISPD 2016 benchmark suite. The comparisons of routed wirelength and runtime are presented in Table 2.11.

As 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 [11].

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

Design	1st Place				2nd Place				3rd Place				GPlace [64]				RippleFPGA [11]				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.

Despite of the different global/detailed placement engines and the execution machines, our approach still shows the best overall routed wirelength. On average, our approach outperforms the three contest winners, GPlace, and RippleFPGA by 8.0%, 14.0%, 44.4%, 25.4%, and 4.1%, respectively. Again, our approach 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, our approach is 3.96×, 4.90×, and 5.84× faster than the three contest winners. While comparing with GPlace and RippleFPGA, our approach runs 1.14× and 1.56× slower.

2.4.4.5 Runtime Breakdown

The runtime breakdown of our approach based on all twelve designs in 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.

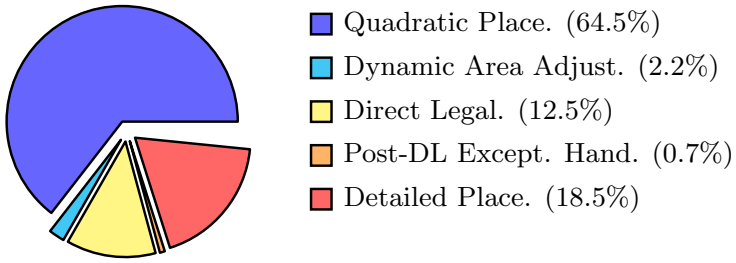


Figure 2.21: The runtime breakdown of our approach based on designs in ISPD 2016 benchmark suite.

2.4.5 Summary

In this section, we have proposed a new paradigm for FPGA placement without explicit packing. The proposed framework significantly improves the correlation between the early placements and the 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 ISPD 2016 benchmark suites 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 further subdivided into quadratic placers and nonlinear placers. Quadratic approaches [2, 11, 23, 24, 79, 80] approximate the placement objective using quadratic functions, while nonlinear approaches [13, 35, 51] 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 [2, 11, 41] or simply packing them before placement [13]. These approaches are usually highly sensitive to the heuristics applied, which could hamper the solution quality and placement robustness. A more recent work [18] proposed a multi-commodity flow-based algorithm for quadratic placers to spread heterogeneous instances, and it demonstrated significant improvement over previous spreading heuristics. However, due to the inherent limitation of quadratic placement, their approach still simplifies spreading as a movement-minimization problem, which cannot explicitly optimize wirelength nor preserve the relative order

among instances of different resource types.

There are also works on optimizing other placement objectives to ease the downstream packing, legalization, and routing steps. Many works [2, 11] adopted instance inflation technique to alleviate routing congestions. Our UTPlaceF-DL, presented in Section 2.4, further considered the impact of downstream packing/legalization and adjusted instance 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 [15, 56] 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 [15, 56] for ASICs to deal with heterogeneous resource types in FPGAs.
- We employ augmented Lagrangian method, instead of the multiplier 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 instance 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 [45] and integrate it, together with routability and pin density optimizations, into elfPlace.
- We demonstrate more than 7% improvement in routed wirelength, on ISPD 2016 benchmark suite [81], 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 [15, 56] is a leading-edge nonlinear global placement algorithm for ASICs. It approximates half-perimeter wirelength (HPWL),

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

using weighted-average (WA) model,

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

Here $\boldsymbol{x}$ and $\boldsymbol{y}$ denote the instance 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 electrostatic system. With this transformation, each instance 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,

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

Table 2.12: Notations used in 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 Eq. (2.17).

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

where Eq. (2.17a) is the Poisson's equation to correlate electric potential, electric field, and charge density, Eq. (2.17b) is Neumann boundary condition (i.e., zero electric field on the boundary of R) to prevent charges from moving out of R, and Eq. (2.17c) neutralizes the overall electric charge and potential to ensure the solution uniqueness of Eq. (2.17). ePlace honors the placement density constraints 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 Eq. (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 $\boldsymbol{\xi} =$

(ξ_x, ξ_y) can be obtained as follows.

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

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

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

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

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 transfrom (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,

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

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 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 instance set and its subset with resource type s
$\mathcal{V}^{\mathrm{P}}$, $\mathcal{V}_s^{\mathrm{P}}$	The physical instance set and its subset of resource type s
$\mathcal{V}^{\mathrm{F}}$, $\mathcal{V}_s^{\mathrm{F}}$	The filler instance set and its subset of resource type s
A_i	The area of instance i
$\mathcal{B}_s$	The bin grid for resource type $s \in \mathcal{S}$
A_b^{P}	The physical instance 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 $(\Phi_{\mathrm{LUT}}, \Phi_{\mathrm{FF}}, \Phi_{\mathrm{DSP}}, \Phi_{\mathrm{RAM}})^T$

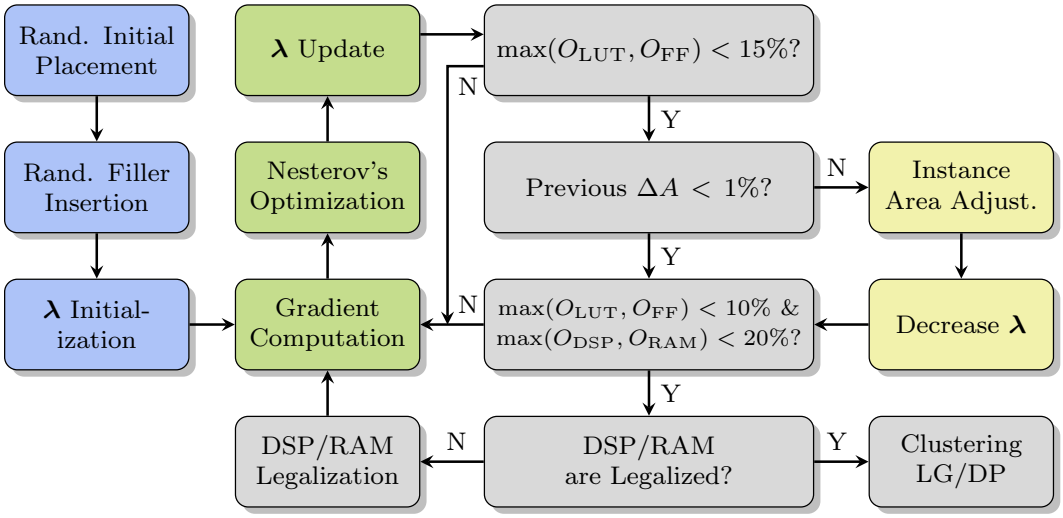


Figure 2.22: The overall flow of elfPlace.

Figure 2.22 illustrates the overall flow of elfPlace. Different from typical initial placement that minimizes wirelength by quadratic programming, elfPlace starts from a random initial placement, which has been observed to

achieve nearly the same quality [52]. In the random initial placement, all movable instances 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 instances 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 the 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 energy 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 gradient of a wirelength-density co-optimization problem is computed and fed to a Nesterov's optimizer [56] 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 instance areas with the consideration of routability, pin density, and downstream clustering compatibility (i.e., LUT input pin constraint and FF control set constraint described in Section 2.2). After the instance 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 Eq. (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 instance 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 [11, 41]. 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 clustering, legalization, and detailed placement approaches proposed in [45] to produce the final legal solution.

2.5.3 Core Placement ALgorithms

2.5.3.1 The Augmented Lagrangian Formulation

With 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 Eq. (2.19), elfPlace uses the augmented Lagrangian method (ALM), as shown in Eq. (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 weight independent to $\boldsymbol{\lambda}$, the magnitude of $\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y})^2$ is also determined by $\boldsymbol{\lambda}$ in Eq. (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 Eq. (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 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 Eq. (2.22) become the penalty method as shown in Eq. (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 Eq. (2.22) become the multiplier method as shown in Eq. (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 potential 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 Eq. (2.22) can be derived as shown in Eq. (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 Eq. (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 in high-potential regions, which often corresponds to locations with relatively large instance overlaps in the placement problem.

The gradient defined in Eq. (2.26) will be preconditioned before being finally fed to the optimizer. Preconditioning can make the local curvature 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 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 Eq. (2.26), we have

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

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

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

where $\mathcal{E}_i$ denotes the set of nets incident to instance 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 method based on Eq. (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.

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

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

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

will be fed to a Nesterov's optimizer [55] to iteratively update the placement solution. Since in FPGA designs, net degrees (e.g., local signal nets and global clock nets) and instance pin counts and sizes (e.g., small LUT/FF instances 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 instances.

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 instance 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 Eq. (2.31), we start from spreading all resource types with the same weight.

Classical optimization approaches ues subgradient method to update $\boldsymbol{\lambda}$ [38]. According to Eq. (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 Eq. (2.22) is not smooth but piecewise linear [38].

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 instances. As a result, using the subgradient in Eq. (2.32) to guide the $\boldsymbol{\lambda}$ updating can lead to severely ill-conditioned problems. To mitigate this issue, the normalized subgradient defined in Eq. (2.33)

is used instead in elfPlace.

$$\begin{aligned}\widehat{\nabla}_{\text{sub}}\boldsymbol{\lambda} &= \Big(\cdots, \frac{1}{\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})}\Big(\Phi_s(\boldsymbol{x}, \boldsymbol{y}) + \frac{c_s}{2}\Phi_s(\boldsymbol{x}, \boldsymbol{y})^2\Big), \cdots\Big)^T, \\ &= \Big(\cdots, \widehat{\Phi}_s(\boldsymbol{x}, \boldsymbol{y}) + \frac{\beta}{2}\widehat{\Phi}_s(\boldsymbol{x}, \boldsymbol{y})^2, \cdots\Big)^T, s \in \mathcal{S}.\end{aligned} \tag{2.33}$$

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 Eq. (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 Eq. (2.34) and our step size updating scheme is further presented by Eq. (2.35).

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

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

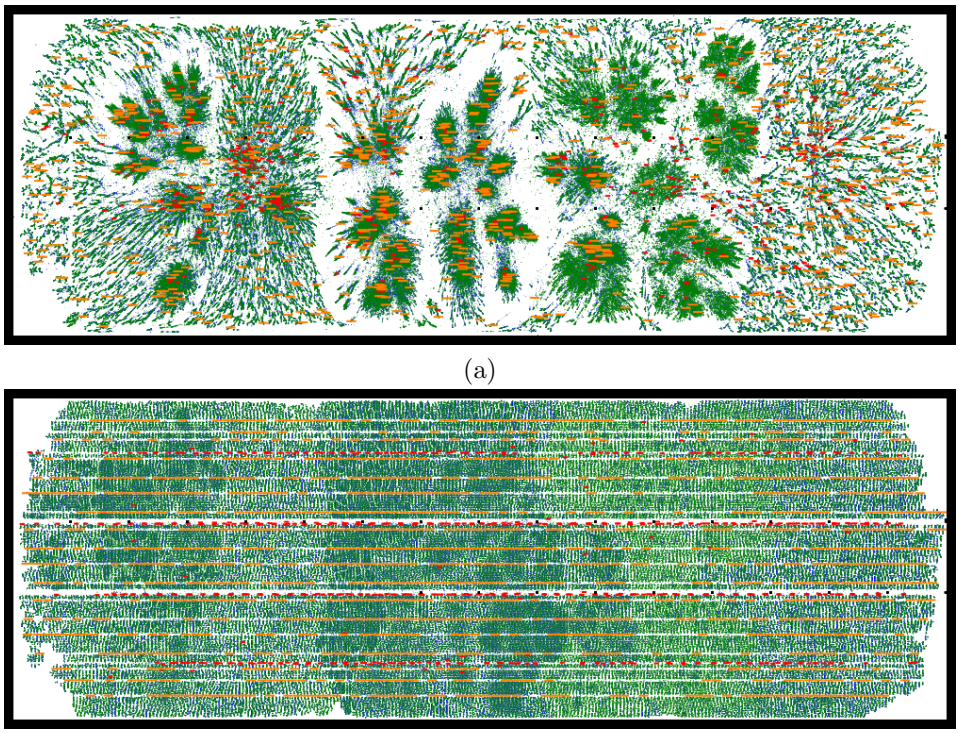
Here β is the same weighting parameter used in Eq. (2.25) and the parameter pair $(\alpha_{\text{L}}, \alpha_{\text{H}})$ defines the range of increasing rate of the step size. The motivation of our step size updating scheme shown in Eq. (2.35) is that the quadratic density penalty term often decays much faster than the linear penalty term in our objective Eq. (2.22). Therefore, we incline to increase the step size faster when the quadratic penalty term dominates (i.e., $\beta\|\widehat{\boldsymbol{\Phi}}^{(k)}\|_2 \gg 1$). As the instance overlaps become smaller, the linear penalty term will start to take over and a slower increasing rate will be used in this case. In our ex-

periments, we set $(\alpha_{\mathrm{L}}, \alpha_{\mathrm{H}})$ 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., $(1.06/1.05)^{500} > 100$).

Figure 2.23 illustrates the heterogeneous spreading process in elfPlace. As can be seen from Fig. 2.23(a), our $\boldsymbol{\lambda}$ updating scheme can greatly preserve those natural physical clusters consisting of heterogeneous instances. The placement right before DSP/RAM legalization shown in Fig. 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 Instance Area Adjustment

Besides minimizing wirelength, elfPlace is also capable of tackling other practical issues in real-world designs, such as routability, pin density, and clustering compatibility. Routability and pin density optimizations have always been the fundamental requirements of placement to achieve routing-friendly solutions. While clustering compatibility optimization, which was recently discussed in [45], is to further consider the effect of downstream clustering (also referred as packing) early in the placement stage. It turns out that all these issues can be addressed by properly adjusting instance areas on top of our wirelength-driven placement. Therefore, in this section, we propose a unified instance area adjustment approach to simultaneously optimize all of them.



(a)

(b)

Figure 2.23: The distributions of physical LUT (green), FF (blue), DSP (red), and RAM (orange) instances 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.

2.5.4.1 The Adjustment Scheme

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

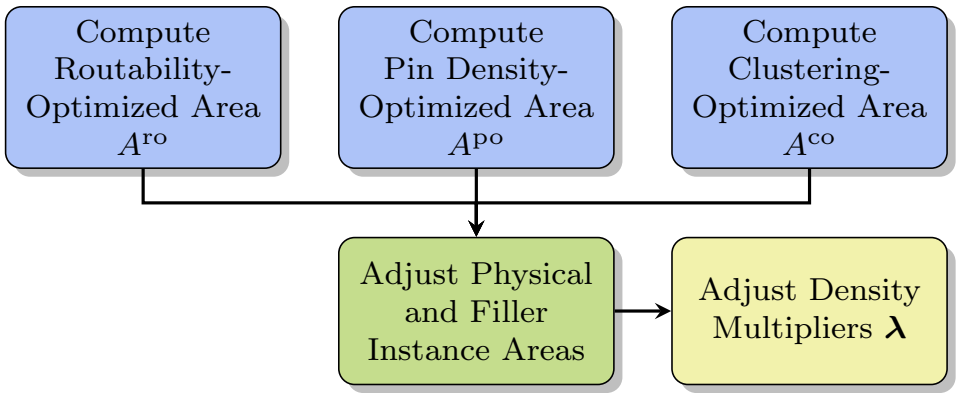


Figure 2.24: The area adjustment flow in elfPlace to simultaneously optimize routability, pin density, and downstream clustering compatibility.

instance i as follows.

$$\Delta A_i = \max(A_i^{\mathrm{ro}}, A_i^{\mathrm{po}}, A_i^{\mathrm{co}}, A_i) - A_i, \forall i \in \mathcal{V}^{\mathrm{P}}. \tag{2.36}$$

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.

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

where $\mathcal{V}_s^{\mathrm{F}}$ denotes the set of filler instances 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 Eq. (2.37) guarantees that the total increased physical instance area is no greater than the total available filler area for each resource type. The final adjusted area A_i' of each physical instance i is then given by Eq. (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 instances to maintain the total positive charge quantity unchanged. The final adjusted area A'_i of each filler instance 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 Eq. (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 Eq. (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 Eq. (2.41), where α_{H} is the same parameter as used in Eq. (2.35).

$$t' = (\alpha_{\mathrm{H}} - 1)\, \|\boldsymbol{\lambda}'\|_2. \tag{2.41}$$

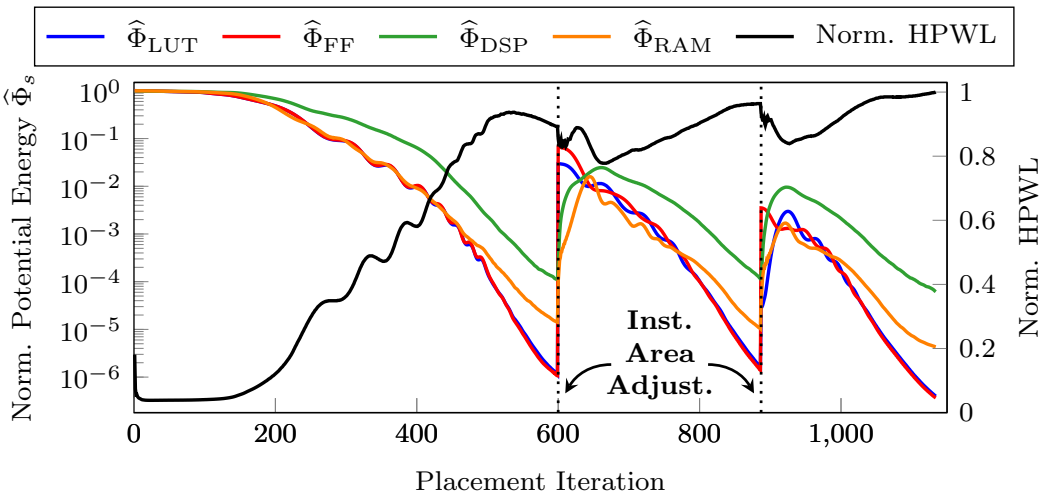


Figure 2.25: The normalized potential energy $\widehat{\boldsymbol{\Phi}}$ and HPWL at different placement iterations on `FPGA-10`.

Figure 2.25 illustrates the impact of instance 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 instance i, elfPlace computes three independent areas that are optimized for routability (A_i^{ro}), pin density (A_i^{po}), and clustering compatibility (A_i^{co}), respectively.

2.5.4.2.1 The Routability-Optimized Area

In order to compute the routability-optimized areas, elfPlace first performs a RISA/RUDY-based [14, 71] routing congestion estimation. Let u_i^{h} and u_i^{v} denote the resulting horizontal and vertical routing utilizations at instance i, then we compute the routability-optimized area of each physical instance i as follows.

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

2.5.4.2.2 The 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 instance 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.

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

Different from the routability-optimized area A_i^{ro}, the pin density-optimized area A_i^{po} here is independent to the current instance 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 Eq. (2.42), it can be excessively over-inflated.

2.5.4.2.3 The Clustering 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 clustering 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 an instance, we intuitively should assign it a larger area.

To estimate the instance areas in a feasible clustering solution, we first assume the instance movement $(\Delta \boldsymbol{x}, \Delta \boldsymbol{y})$ during the downstream clustering/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 instance 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 $(\mathcal{B}_i^{\text{xl}}, \mathcal{B}_i^{\text{yl}}, \mathcal{B}_i^{\text{xh}}, \mathcal{B}_i^{\text{yh}})$ denote the bounding box of the estimation window $\mathcal{B}_i$ of instance i, the expectation of any instance j falling into $\mathcal{B}_i$ then can be defined as

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

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 instance 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 clustering compatibility-optimized area for LUT i as follows.

$$A_i^{\mathrm{co}} = \frac{1}{16}\frac{\sum_{j\in\mathcal{V}_i^{\mathrm{P}}} \mathbf{E}_{j\in\mathcal{B}_i}}{\sum_{j\in\mathcal{V}_{\mathrm{LUT}}^{\mathrm{P}}} \mathbf{E}_{j\in\mathcal{B}_i}} + \frac{1}{8}\frac{\sum_{j\in\overline{\mathcal{V}_i^{\mathrm{P}}}} \mathbf{E}_{j\in\mathcal{B}_i}}{\sum_{j\in\mathcal{V}_{\mathrm{LUT}}^{\mathrm{P}}} \mathbf{E}_{j\in\mathcal{B}_i}}, \forall i \in \mathcal{V}_{\mathrm{LUT}}^{\mathrm{P}}. \tag{2.45}$$

Equation (2.45) essentially is a weighted average of the compatible and incompatible expectations for i within the window $\mathcal{B}_i$. Each $A_i^{\mathrm{co}}, \forall i \in \mathcal{V}_{\mathrm{LUT}}^{\mathrm{P}}$, 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 an FF instance i, let θ_i denote its control set $(\mathrm{CK}, \mathrm{SR}, \mathrm{CE})$ 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 clustering solution formed within $\mathcal{B}_i$ can be estimated by Eq. (2.46), as given in [45].

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

We omit the derivation of Eq. (2.46) due to the page limit. However, it still can be seen that Eq. (2.46) involves many ceiling operations ($\lceil\cdot\rceil$), which make $A_i^{\text{co-disc}}$ discontinuous (disc) and very sensitive to the estimation window $\mathcal{B}_i$ and the placement solution.

In elfPlace, the much smoother Eq. (2.47), instead of Eq. (2.46), is used

as the clustering compatibility-optimized areas for FF instances.

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

It has two notable improvements over Eq. (2.46): (1) it replaces each FF count $n_{i,\theta}$ in Eq. (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 Eq. (2.46) with the soft division-ceiling function $\mathtt{sdc}(x, d)$ defined in Eq. (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 Fig. 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 $\lceil x/d \rceil$.

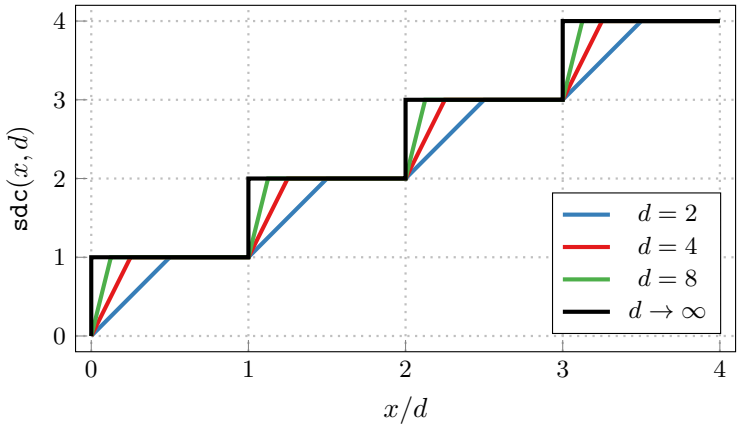


Figure 2.26: The plots of the soft division-ceiling function $\mathtt{sdc}(x, d)$ w.r.t. x/d.

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 framework with the support of OpenMP 4.0 [1]. The ISPD 2016 FPGA placement contest benchmark suite [81] 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 [41], RippleFPGA [11], GPlace3.0 [2], and UTPlaceF-DL [45]. 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 UTPlaceF, 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× slower than the fastest RippleFPGA. By exploiting 10 threads, elfPlace achieves 3.51× 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 [41]				RippleFPGA [11]				GPlace3.0 [2]				UTPlaceF-DL [45]						elfPlace					
	WL	WLR	1-thread		WL	WLR	1-thread		WL	WLR	1-thread		WL	WLR	1-thread		10-thread		WL	WLR	1-thread		10-thread	
			RT	RTR			RT	RTR			RT	RTR			RT	RTR	RT	RTR			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

2.5.5.2 Individual Technique Validation

Table 2.16 further validates the effectiveness of each proposed technique. The column "w/ MM in Eq. (2.24)" shows the results of using the multiplier method (MM) in Eq. (2.24), which is adopted by ePlace, instead of our proposed augmented Lagrangian method (ALM) in Eq. (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 Eq. (2.29) and it can barely converge due to the wide spectrum of instance sizes and net degrees in FPGA designs. The column "w/ ePlace Precond." further gives the results of replacing our preconditioner in Eq. (2.29) with the one proposed in ePlace [55]. 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^{co} in [45]" presents the results of using the clustering compatibility-optimized area proposed in [45] instead of our Gaussian and `sdc`-smoothed Eq. (2.45) and Eq. (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 Eq. (2.24)		w/o Precond.		w/ ePlace Precond.		w/ A_i^{co} in [45]		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 Eq. (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 clustering, legalization, and detailed placement algorithms adopted from [45] consumes total 37.5% of the runtime. While the remaining 11.9% of the runtime is spent on

parsing, placement initialization, and the rest of runtime-insignificant tasks.

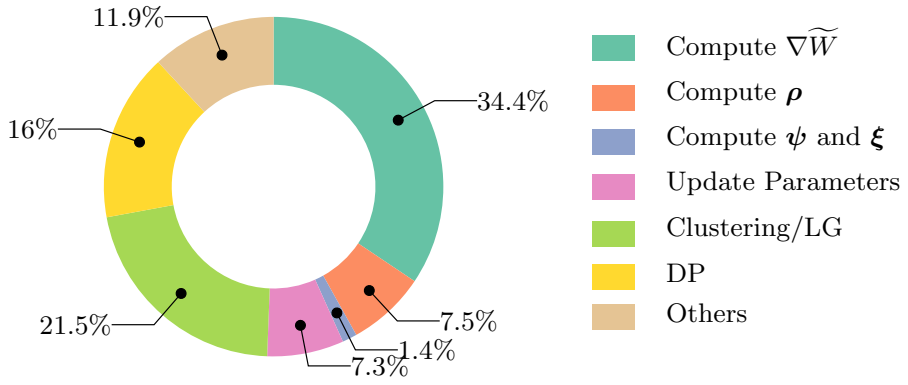


Figure 2.27: The runtime breakdown of the 10-threaded elfPlace based on `FPGA-12`.

2.5.6 Summary

In this paper, 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 clustering compatibility based on a unified instance 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 placement stage. These constraints have great impact 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 developement, 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 logics 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 [2, 6, 11, 13, 24, 41, 45, 51, 60, 68–70, 77, 79], there has been limited research effort on placement with the awareness of clock rules. The closest related previous work is [36], 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.

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 min-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 frameworks, Section 3.3 describes UTPlaceF 2.0, and Section 3.4 details UTPlaceF 2.X.

3.2 Target FPGA Clocking Architecture

Our targeting FPGA device is Xilinx *UltraScale VU095*, which was also adopted in both ISPD 2016 and ISPD 2017 FPGA placement contests [81, 82]. Its clocking architecture is illustrated in Fig. 3.1(a) – (c). The global clocking architecture, as shown in Fig. 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 paper, we will denote the horizontal/vertical routing layer as HR/VR, and the horizontal/vertical distribution layer as HD/VD. In the targeting 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 Fig. 3.1(c). Specifically, it consists of two parts, a D-layer vertical trunk tree connecting all the clock regions containing clock loads, and an R-layer route connecting the clock source and the D-layer trunk tree. More detailed clocking architecture can be found in [26].

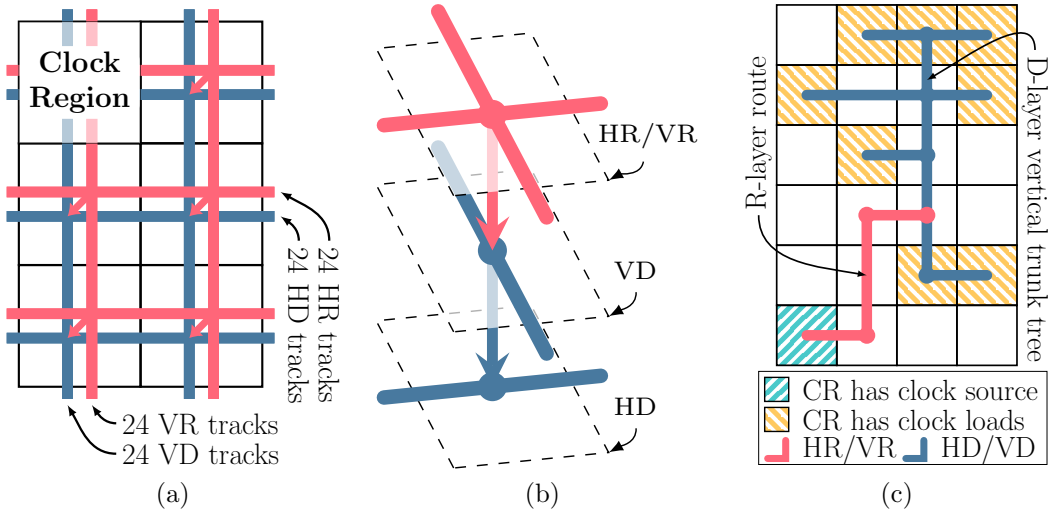


Figure 3.1: Illustration of the targeting clocking architecture. (a) A global view of 2 × 3 clock regions with R-layer (red) and D-layer (blue). (b) A detailed view of HR/VR/HD/VD within a single clock region. (c) The required routing pattern of a clock net.

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

Modern FPGA devices contain complex clock architectures on top of configurable logics. The physical structure of clock networks in an FPGA is pre-manufactured 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 first place in ISPD'17 clock-aware FPGA placement contest [82] on industry-strength benchmarks released by Xilinx [30] and outperforms the second- and third-place winners by 4.0% and 10.0%, respectively, in routed wirelength with competitive runtime.

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

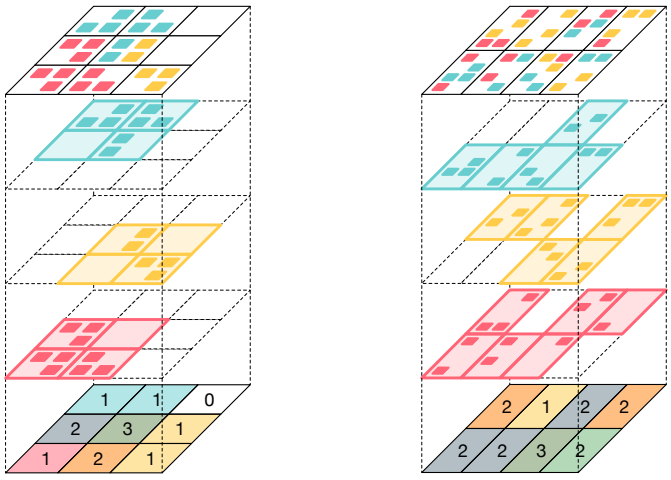
3.3.1 Preliminaries and Overview

In this section, we will briefly introduce the clock constraints and problem formulation for clock-aware placement. At the end of this section, an overview of the proposed UTPlaceF 2.0 framework will be exposed.

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.

Clock region constraint is introduced by the limited clock routing/distribution tracks and it restricts the global clock demand in each clock region must be equal to or less than 24 in our targeted FPGA architecture.



(a) (b)

Figure 3.2: Illustration of 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.

Exact global clock demand in each clock region requires detailed clock routing, which 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 for each clock, 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, 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 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 measured by Half-Perimeter Wirelength (HPWL), and routability. Wirelength is still 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 often results 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, packing rules also need to be satisfied when multiple LUTs and FFs are placed into the same CLB site. The detailed packing rules for our targeted FPGA are elaborated in [81].

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.

3.3.1.3 UTPlaceF 2.0 Overview

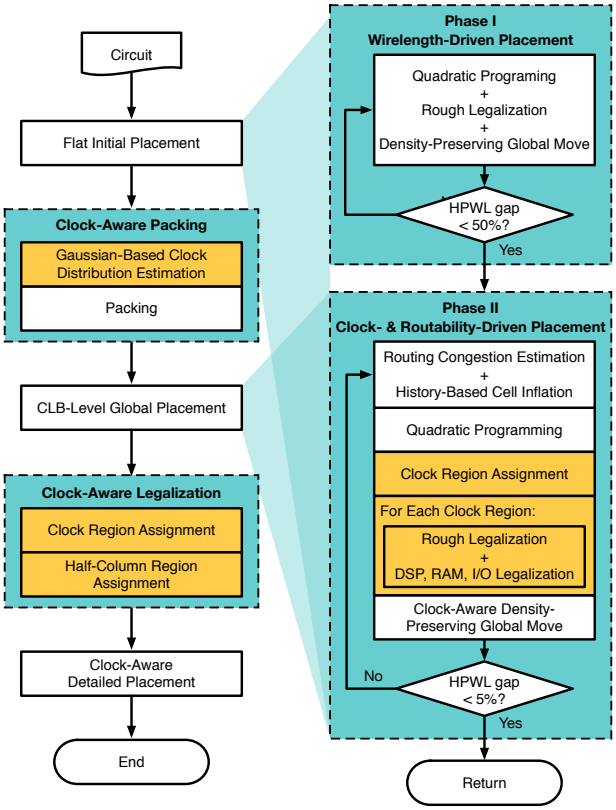


Figure 3.3: The overall flow of UTPlaceF 2.0. Yellow-shaded blocks indicates major new/modified steps that differ from UTPlaceF.

The overall flow of UTPlaceF 2.0 is shown in Figure 3.3. UTPlaceF 2.0 is a natural extension of UTPlaceF [41] and consists of five major steps: 1) flat initial placement (FIP), 2) packing, 3) CLB-level global placement, 4) legalization, and 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.

FIP is responsible for producing physical location and routing congestion information of each cell to better guide decision making in the packing stage. It consists of two phases: 1) pure wirelength-driven phase and 2) clock- and routability-driven phase. In each iteration of the first phase, a quadratic analytical placement utilizing hybrid [78] and bound-to-bound [72] net models is solved to minimize wirelength, followed by the rough legalization technique proposed in [50] for cell overlapping removal. After that, a sequence of global moves are performed to further refine rough-legalized placement while preserving cell density. In the second phase, besides the conventional routability optimization, an additional clock region assignment (see Section 3.3.2.1) 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 heterogeneous cell (e.g., DSPs, RAMs, and I/Os) 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 3.3.2.2).

CLB-level global placement is performed immediately after packing to

further optimize the placement for the post-packing netlist. It shares the same framework with the second phase of FIP, and use the final solution of FIP as the starting point to speed-up placement convergence.

In legalization stage, while minimizing the conventional objective of pin movement, clock region constraint and half-column region constraint are rigorously respected by clock region assignment technique and half-column region assignment technique (see Section 3.3.2.3), respectively.

Finally, detailed placement techniques in UTPlaceF is 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 UTPlaceF 2.0 Algorithms

In this section, we will explain UTPlaceF 2.0 algorithms, including clock region assignment, clock-aware packing, and half-column region assignment, in details.

3.3.2.1 Clock Region Assignment

Clock region assignment is a key step in UTPlaceF 2.0 to ensure the satisfaction of 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 rough-legalized place-*

ment and logic resource capacity of each clock region, assign cells to clock regions to minimize total pin movement without logic resource and 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 $s \in$ {CLB, DSP, RAM}.
P_i	The number of pins in cell i.
$A_i^{(s)}$	The cell i's demand for resource type $s \in$ {CLB, DSP, RAM}.
$\mathcal{R}$	The set of clock regions.
$C_j^{(s)}$	The clock region j's capacity for resource type $s \in$ {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.

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad \sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{R}} P_i \cdot D_{i,j} \cdot \boldsymbol{x}_{i,j}, \tag{3.1a}$$

$$\text{subject to} \quad \boldsymbol{x}_{i,j} \in \{0,1\}, \forall i \in \mathcal{V}, \forall j \in \mathcal{R}, \tag{3.1b}$$

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

$$\sum_{i\in\mathcal{V}} A_i^{(s)} \cdot \boldsymbol{x}_{i,j} \leq C_j^{(s)}, \forall j \in \mathcal{R}, \forall s \in \{\text{CLB, DSP, RAM}\} \tag{3.1d}$$

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

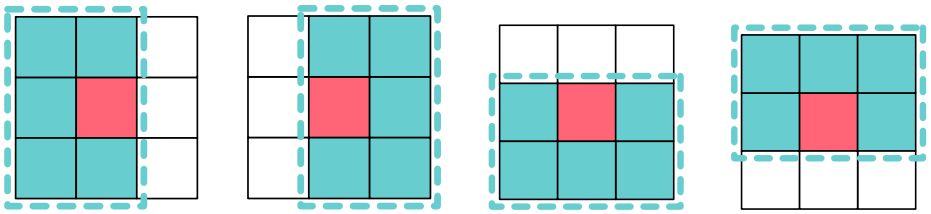


Figure 3.4: Illustration of the sets of clock regions (dashed regions) (a) $\mathcal{L}_j^+$, (b) $\mathcal{R}_j^+$, (c) $\mathcal{B}_j^+$, and (d) $\mathcal{T}_j^+$ of clock region j (red) defined in Table 3.1.

Formulation (3.1) is optimized over binary variables $\boldsymbol{x}_{i,j}$ 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 $\boldsymbol{x}_{i,j} = 1$, otherwise $\boldsymbol{x}_{i,j} = 0$. The constraint (3.1c) ensures that each cell is assigned to one and only one clock region. The constraint (3.1d) guarantees 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 the satisfaction of clock region constraint. The sets of clock regions $\mathcal{L}_j^+$, $\mathcal{R}_j^+$, $\mathcal{B}_j^+$, and $\mathcal{T}_j^+$ of 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 $\mathcal{L}_j^+$, $\mathcal{R}_j^+$, $\mathcal{B}_j^+$, 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.

3.3.2.1.1 The Relaxation Algorithm

With proper modeling, the boolean logical constraint (3.1e) can be transformed to 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 unaffordable 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).

$$\underset{\boldsymbol{x},\lambda}{\text{minimize}} \quad \sum_{\mathrm{i}\in\mathcal{V}}\sum_{\mathrm{j}\in\mathcal{R}}(\mathrm{P_i}\cdot\mathrm{D_{i,j}}+\lambda_{\mathrm{i,j}})\cdot\mathrm{x_{i,j}}, \tag{3.2a}$$

$$\text{subject to} \quad \boldsymbol{x}_{i,j}\in\{0,1\}, \forall i\in\mathcal{V}, \forall j\in\mathcal{R}, \tag{3.2b}$$

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

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

Compared to the original Formulation (3.1), Formulation (3.2) has two major differences: 1) instead of considering 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 three 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 $\boldsymbol{x}_{i,j}$ 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 respect each logic resource constraint separately. Each time after solving Formulation (3.2), we properly penalize assignments causing clock overflow 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.

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, in line 1. Then all penalty multipliers $\lambda_{i,j}, \forall i \in \mathcal{V}, \forall j \in \mathcal{R}$ are initialized to zero in 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 in line 7, penalty

Algorithm 9: Clock Region Assignment with Relaxation

Input : A rough-legalized placement.
Output: A movement-minimized assignment without logic resource and clock overflow.

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 3.3.2.1.2);
6 **end**
7 Update penalty multiplier λ (See Section 3.3.2.1.3);
8 **end**

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 overflow 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.

3.3.2.1.2 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 is a fairly mature field with many efficient algorithms [3].

Figure 3.5 illustrates the minimum-cost flow representation of Formulation (3.2). If all cells in $\mathcal{V}$ have unit resource demand ($A_i = 1, \forall i, j \in \mathcal{V}$),

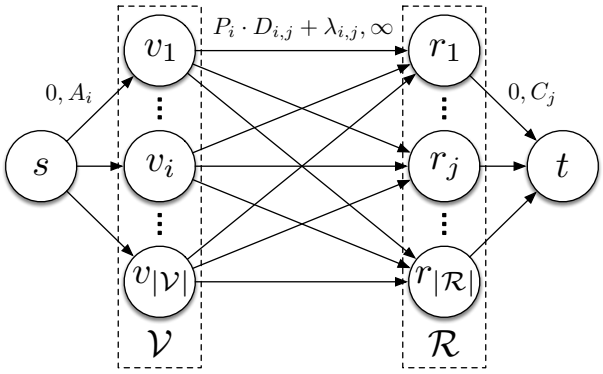


Figure 3.5: The minimum-cost flow representation of Formulation (3.2). Pair of numbers (e.g., $P_i \cdot D_{i,j} + \lambda_{i,j}, \infty$) on each edge represents cost and capacity, respectively, and ∞ means unlimited capacity.

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

3.3.2.1.3 Penalty Multiplier Updating

Algorithm 9 is a generalized relaxation framework for solving the clock region assignment problem. One key step within this framework is how to update the penalty multiplier λ after each assignment iteration. Various updating strategies can converge to different feasible solutions. In this section, we will

only discuss one specific updating method, which is applied in UTPlaceF 2.0, and its effectiveness is demonstrated by our experiments.

The guiding idea of our λ updating method is that we hope clock overflow 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, 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.

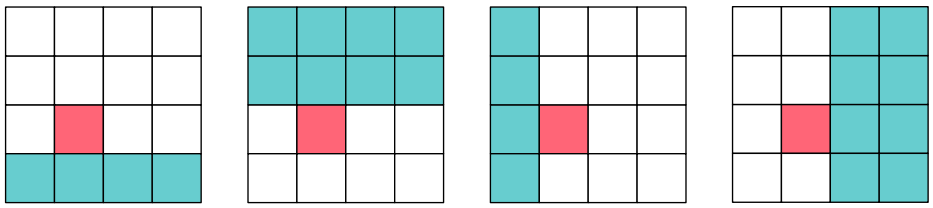


Figure 3.6: The four escaping regions (green) that are defined as the regions (a) below, (b) above, (c) left, and (d) right to a given clock region (red), respectively. To avoid employing clock resources of the given clock region, all cells of a clock net must be simultaneously placed into one of these four escaping regions.

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 in line 1 and locate the four

Algorithm 10: Penalty Multiplier Updating For Clock Region Constraint

Input : Penalty multiplier λ. The maximum clock overflow descent step I_{max} for each clock region in each iteration.

Output: An updated penalty multiplier λ that will results in less clock overflow.

Find the clock region r with the largest clock overflow O_{max};
$B_r \leftarrow$ the region below to r;
$T_r \leftarrow$ the region above to r;
$L_r \leftarrow$ the region left to r;
$R_r \leftarrow$ the region right to r;
$l \leftarrow \varnothing$;
foreach *clock* $e \in \mathcal{E}$ *occupying clock resource of* r **do**
    foreach *region* $b \in \{B_r, T_r, L_r, R_r\}$ **do**
        $l \leftarrow l \cup \{\texttt{Cost}(e, b)\}$ (See Equation (3.3));
Sort candidates in l by ascending order of their cost;
$I \leftarrow \min(O_{max}, I_{max})$;
$i \leftarrow 0$;
$chosen[e] \leftarrow false, \forall e \in \mathcal{E}$;
foreach `Cost` $(e, b) \in$ *sorted* l **do**
    if $chosen[e]$ ***or*** $\neg$`IsFeasible`(e, b, λ) **then continue**;
    foreach *cell* i *in clock* e **do**
        for *each clock region* j *that is not in region* b **do**
            $\lambda_{i,j} \leftarrow \infty$;
    $chosen[e] \leftarrow true$;
    $i \leftarrow i + 1$;
    if $i \geq I$ **then return**;

escaping regions illustrated in Figure 3.6 for the target clock region from line 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 ascending order of their costs. For each candidate, in 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 overflow in the target clock is fully resolved or the number of resolved overflow 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. Under 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 `Cost`(e, b) that computes the cost of pushing clock e to escaping region b. The objective of our assignment is to minimize 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: pin movement cost and logic resource cost, shown as follows,

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

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 `IsFeasible`(e, b, λ). 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 actual applying the new assignment blockings. 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.

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

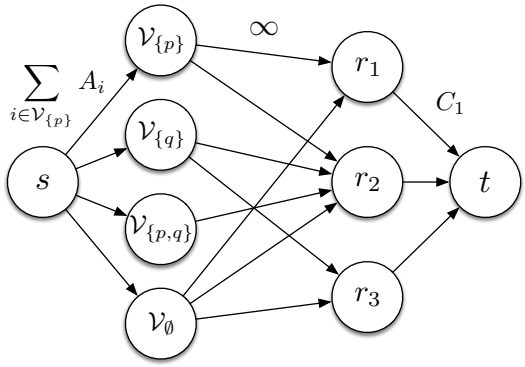


Figure 3.7: Illustration of our maximum-flow-based feasibility checking. Numbers on edges represent their capacities.

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 ($\lambda_{i,j} \neq \infty$), and specific to 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 blockings applied must exist, otherwise, the assignment problem becomes infeasible.

It should be noted that our maximum-flow-based checking are performed for three resource types (CLB, DSP, and RAM) separately each time and we only conclude the updated problem is feasible when all three checks

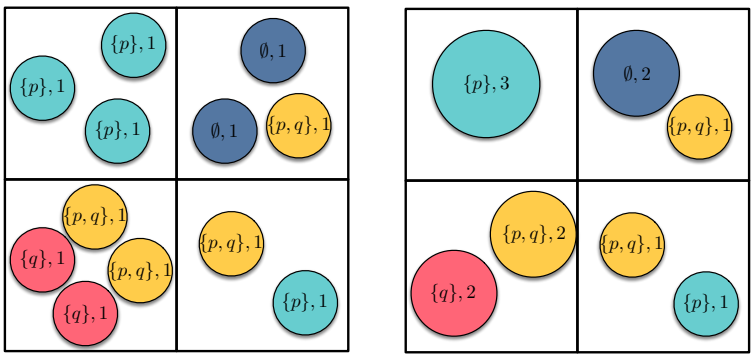
are passed.

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.

3.3.2.1.4 Speed-up Techniques

The minimum-cost flow formulation shown in Figure 3.5 is optimal for given λ and unit logic resource demand but 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 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



(a) (b)

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.

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).

$$\underset{\boldsymbol{x},\lambda}{\text{minimize}} \quad \sum_{\mathrm{k}\in\mathcal{K}}\sum_{\mathrm{j}\in\mathcal{R}}\left(\frac{\mathrm{P_k}}{\mathrm{A_k}}\cdot \mathrm{D_{k,j}} + \lambda_{\mathrm{k,j}}\right)\cdot \mathrm{x_{k,j}}, \tag{3.4a}$$

$$\text{subject to} \quad \boldsymbol{x}_{k,j} \in \{0, 1, 2, \dots\}, \forall k \in \mathcal{K}, \forall j \in \mathcal{R}, \tag{3.4b}$$

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

$$\sum_{k\in\mathcal{K}} \boldsymbol{x}_{k,j} \leq C_j, \forall j \in \mathcal{R}. \tag{3.4d}$$

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 emphasised.

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 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, split assignment would not happen again in these second-level minimum-cost flows and the solution feasibility can be guaranteed.

3.3.2.2 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, packing is performed by pairwisely merging cells based on a set of attraction functions, which grant higher priority 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 solution quality with the newly introduced clock region constraint.

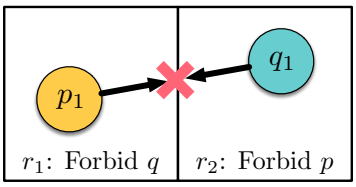


Figure 3.9: An example to show the necessity of clock-aware packing. p and q denote two clock nets. p_1 and q_1 are two cells belonging to clock p and q, respectively.

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 solution feasibility.

3.3.2.2.1 Probability-Based Clock Distribution Estimation

One common idea to avoid the case shown in Figure 3.9 is to let packing algorithms respect 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 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. 3.3.2.2.2) to help us smartly make use of those spare clock resources.

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 relative to their loca-

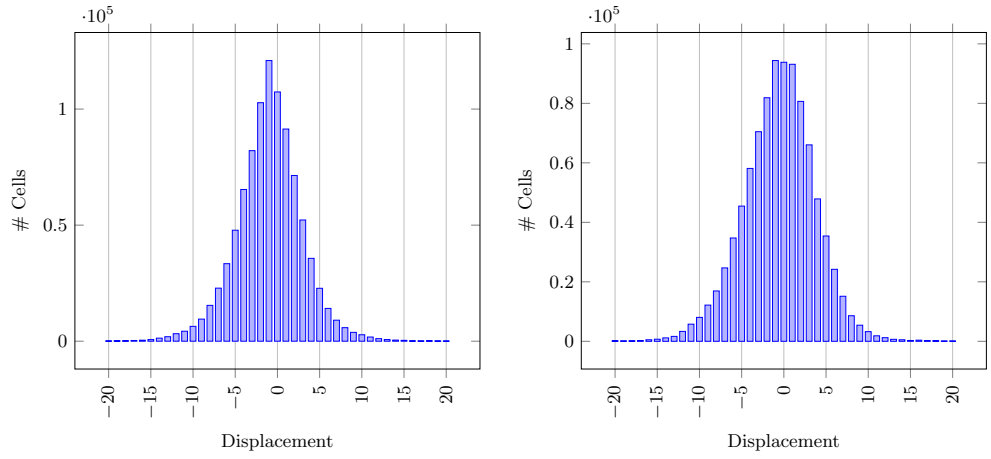


Figure 3.10: Distributions of cell displacement in the final placement relative to FIP in (a) x direction and (b) y direction for a representative benchmark, CLK-DESIGN5 (0.94M cells), in ISPD'17 contest example benchmark suite. All placement solutions are generated by UTPlaceF without clock legality consideration.

tions 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 mean value 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.

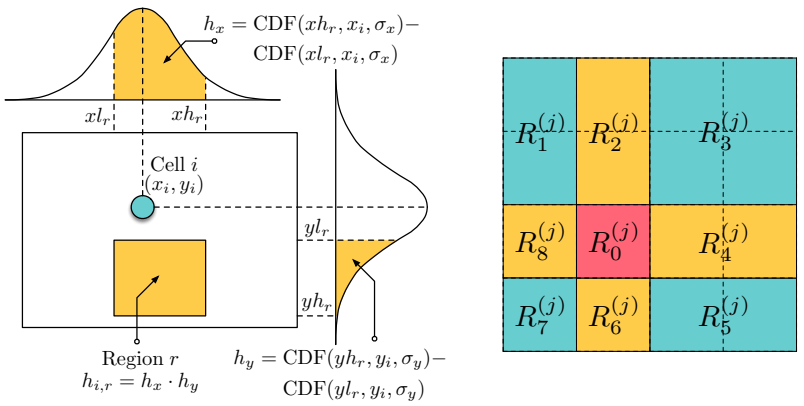
Given the assumption in Equation (3.5), a cell i at location (x_i, y_i) in FIP, and any region r with bounding box (xl_r, yl_r, xh_r, yh_r), the probability of cell i being placed into the region r in the final placement, denoted by $h_{i,r}$, can be written as follows,

$$h_{i,r} = (\text{CDF}(xh_r, x_i, \sigma_x) - \text{CDF}(xl_r, x_i, \sigma_x)) \cdot (\text{CDF}(yh_r, y_i, \sigma_y) - \text{CDF}(yl_r, y_i, \sigma_y)). \tag{3.6}$$

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 [4]. Figure 3.11(a) gives a visual illustration for 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 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



(a) (b)

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).

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

$$H_{j,k} = 1 - \overline{H_{j,k}}. \quad (3.9)$$

3.3.2.2.2 Clock-Aware Packing Attraction Function

UTPlaceF 2.0 adopts the UTPlaceF packing framework, which consists of maximum-weighted-matching-based and BestChoice-based [62] 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 priority for objects that share more small nets [41]. The original packing attraction function for two connected objects i and j then can be generalized as follows,

$$\phi_{i,j} = \left(1 - e^{\gamma \cdot (D_{i,j} - \overline{\lambda})}\right) \cdot \sum_{e \in \text{Net}(i) \cap \text{Net}(j)} \frac{k}{P_e - 1}. \tag{3.10}$$

where $D_{i,j}$ denotes the physical distance between i and j in FIP, $\text{Net}(i) \cap \text{Net}(j)$ denotes the set of nets that are shared by i and j, P_e represents the number of pins in net e, k is 2 for two-pin nets and 1 for other nets, and γ and $\overline{\lambda}$ are tuning parameters determined experimentally.

The proposed new attraction function is 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,

$$\phi_{i,j} = \left(1 - e^{\gamma \cdot (D_{i,j} - \overline{\lambda})}\right) \cdot \sum_{e \in \text{Net}(i) \cap \text{Net}(j)} \frac{k}{P_e - 1} \cdot \prod_{k \in \mathcal{E}(i) \cup \mathcal{E}(j)} H_{t,k}. \tag{3.11}$$

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 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.

3.3.2.3 Half-Column Region Assignment

As shown in Figure 3.3, half-column region constraint is explicitly respected in 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 rough-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 and 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 3.3.2.1.1 and Section 3.3.2.1.2 can be seamlessly applied to solve half-column region assignment problem as well.

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

Input : Penalty multiplier λ.
Output: An updated penalty multiplier λ that will results in less clock overflow.

Find the half-column region r with the largest clock overflow O_{max};
foreach *clock* e *in* r **do**
  $l \leftarrow l \cup \{\texttt{Cost}(e, r)\}$ (See Equation (3.12));
end
Sort candidates in l by ascending order of their cost;
$i \leftarrow 0$;
foreach `Cost` $(e, r) \in$ *sorted* l **do**
  if $\neg$`IsFeasible`(e, r, λ) **then continue**;
  foreach *cell* i *in clock* e **do**
    $\lambda_{i,r} \leftarrow \infty$;
  end
  $i \leftarrow i + 1$;
  if $i \geq O_{max}$ **then return**;
end

The only notable difference from 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 a clock region, we first find the target half-column region with the largest clock overflow in line 1. Then, the cost of moving each clock out of the target half-column region is calculated and stored in a list from 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 check similar to the one illustrated in Figure 3.7 is performed in 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 in line 13.

3.3.3 Experimental Results

UTPlaceF 2.0 was implemented in C++ and compiled by g++ 4.7.2. The official contest evaluation results conducted by Xilinx is used here to

demonstrate the effectiveness of UTPlaceF 2.0.

The characteristics of ISPD'17 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.

Table 3.2: ISPD'17 Placement Contest Benchmarks Statistics

Benchmark	#LUT	#FF	#RAM	#DSP	#Clocks
CLK-FPGA01	215K	236K	170	75	30
CLK-FPGA02	215K	236K	170	75	30
CLK-FPGA03	242K	270K	255	112	33
CLK-FPGA04	268K	300K	340	150	36
CLK-FPGA05	295K	325K	425	187	39
CLK-FPGA06	322K	354K	510	225	42
CLK-FPGA07	350K	384K	595	262	45
CLK-FPGA08	376K	414K	680	300	48
CLK-FPGA09	392K	431K	765	337	51
CLK-FPGA10	408K	449K	850	375	54
CLK-FPGA11	424K	450K	900	397	55
CLK-FPGA12	440K	484K	950	420	56
CLK-FPGA13	456K	503K	1000	442	57
Resources	538K	1075K	1728	768	N/A

As UTPlaceF 2.0 won the first place in ISPD'17 placement contest, we here compare our results with the second- and third-place contest winners. Table 3.3 shows the routed wirelength comparison results that reported by Xilinx Vivado v2016.4. It can be seen that UTPlaceF 2.0 achieves the best overall routed wirelength and outperforms by 4.0% and 11.0% in routed wirelength compared with the other two contest winners, respectively.

The runtime comparison is shown in Table 3.4. 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 detailed placement, CLB-level global placement, packing, and legalization respectively consume 20.0%, 5.3%, 3.3%, and 0.3% of the total runtime. 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.

Table 3.3: Routed Wirelength (WL in 10^3) Comparison with ISPD'17 Contest Winners

Benchmark	2nd Place		3rd Place		UTPlaceF 2.0 (1st Place)	
	WL	Ratio	WL	Ratio	WL	Ratio
CLK-FPGA01	2209	1.001	2269	1.027	2208	1.000
CLK-FPGA02	2274	0.998	2504	1.099	2279	1.000
CLK-FPGA03	6229	1.164	5803	1.084	5353	1.000
CLK-FPGA04	3817	1.032	4086	1.105	3698	1.000
CLK-FPGA05	4995	1.065	5181	1.104	4692	1.000
CLK-FPGA06	5606	1.003	6217	1.112	5589	1.000
CLK-FPGA07	2505	1.024	2676	1.095	2445	1.000
CLK-FPGA08	1990	1.055	2057	1.091	1886	1.000
CLK-FPGA09	2583	0.995	2814	1.084	2597	1.000
CLK-FPGA10	4770	1.069	4840	1.084	4464	1.000
CLK-FPGA11	4208	1.006	4777	1.142	4184	1.000
CLK-FPGA12	3377	1.002	3740	1.110	3369	1.000
CLK-FPGA13	3921	1.019	4320	1.123	3848	1.000
Norm.	1.040	-	1.100	-	1.000	-

Table 3.4: Runtime (RT in seconds) Comparison with ISPD'17 Contest Winners

Benchmark	2nd Place		3rd Place		UTPlaceF 2.0 (1st Place)	
	RT	Ratio	RT	Ratio	RT	Ratio
CLK-FPGA01	3023	5.68	354	0.67	532	1.00
CLK-FPGA02	3153	6.15	333	0.65	513	1.00
CLK-FPGA03	4066	3.91	666	0.64	1039	1.00
CLK-FPGA04	3077	4.33	464	0.65	711	1.00
CLK-FPGA05	3631	3.87	680	0.72	939	1.00
CLK-FPGA06	3836	3.60	695	0.65	1066	1.00
CLK-FPGA07	3953	4.68	410	0.49	845	1.00
CLK-FPGA08	4395	8.31	277	0.52	529	1.00
CLK-FPGA09	5428	6.45	414	0.49	842	1.00
CLK-FPGA10	3305	3.39	516	0.53	974	1.00
CLK-FPGA11	4341	4.06	548	0.51	1068	1.00
CLK-FPGA12	4949	6.39	413	0.53	774	1.00
CLK-FPGA13	3748	3.20	548	0.47	1172	1.00
Norm.	4.63	-	0.57	-	1.00	-

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 3.3.2.1.3 by experiments shown in Table 3.5. 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, a MCF instead of a 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 3.5. The overall speedup of "MCF

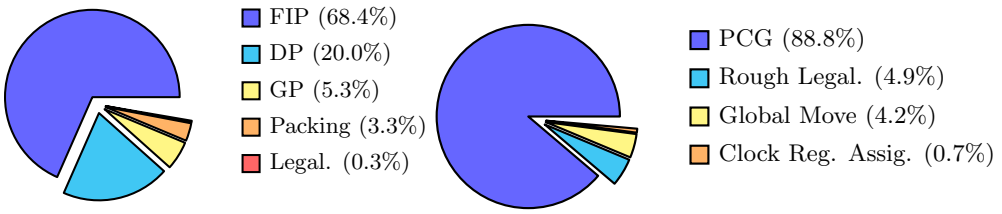


Figure 3.12: Runtime breakdown of (a) UTPlaceF 2.0 and (b) flat initial placement (FIP).

+ MF" over "Pure MCF" for each benchmark is listed in the last column of Table 3.5.

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

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 $\times 2.6$ overall speedup. Note that the runtime of MCF is dependent to the partition sizes mentioned in Section 3.3.2.1.4 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-offs of Different Partition Sizes

Figure 3.13 gives the trade-offs 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

Table 3.5: 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

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 to 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 pro-

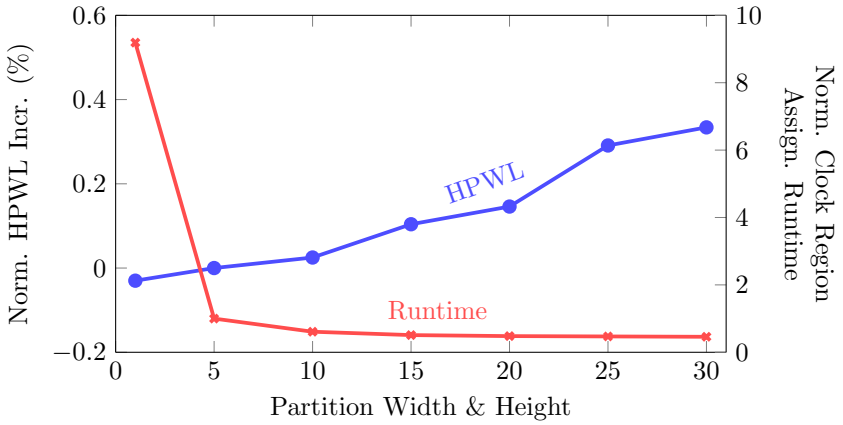


Figure 3.13: Trend of HPWL and clock region assignment runtime with different partition sizes for clustering (Section 3.3.2.1.4) on benchmark CLK-FPGA05. All HPWL and runtime are normalized to the result of partition width and height = 5.

posed clock region assignment and clock-aware packing techniques applied, UTPlaceF 2.0 can satisfy all clock constraints by only paying a little wirelength overhead ($< 1\%$).

3.3.4 Summary

With the increasing complicated FPGA clocking architecture, respecting clock rules is becoming a fundamental issue in modern FPGA implementation flow. In this paper, 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 estimationm method, and a clock-aware packing technique are proposed for better honoring clock legality

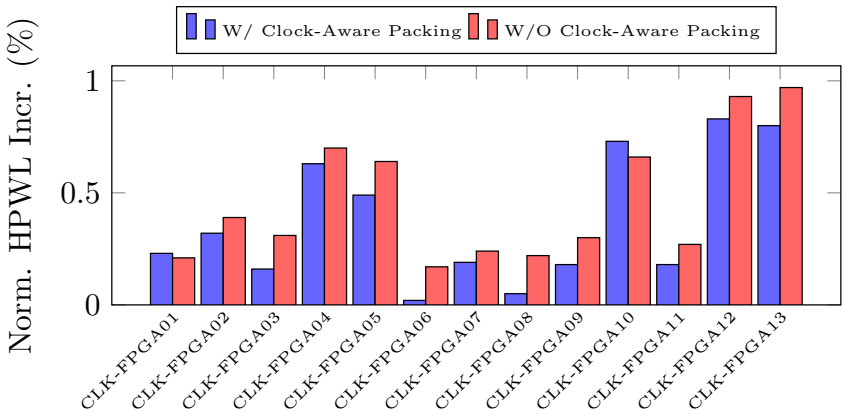


Figure 3.14: Normalized HPWL increases (%) of placements w/ and w/o clock-aware packing compared to placements without any clock constraint considerations.

throughout the whole placement and packing process. As the first place winner of ISPD'17 clock-aware FPGA placement contest, UTPlaceF 2.0 can achieve legal and high-quality placement solutions efficiently, which outperforms all other contest placers in routed wirelength with competitive runtime.

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

Several previous works have tentatively explored placement techniques with the awareness of clock feasibility for FPGAs. In [36], a cost function that penalizes high-clock-usage placements is proposed and integrated into the simulated annealing-based *VPR* framework [6] to produce clock-friendly solutions. In the more recent *ISPD 2017 Clock-Aware Placement Contest* [82] hold by Xilinx, the top-3 winners UTPlaceF 2.0 [43], NTUfplace [13], and RippleFPGA [66] adopt a more realistic commercial FPGA clocking architecture. In these three works, to simplify the clock legalization problem, clock routing of clock networks is approximated by the bounding boxes of their clock loads. 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 Fig. 3.15(b). It shows the same clock loads distribution as that in Fig. 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 Fig. 3.15(a), which consumes 9 clock regions, the same clock network only spans 6 clock regions with tree construction in Fig. 3.15(b).

Apart from the clock routing modeling inaccuracy, all these three works

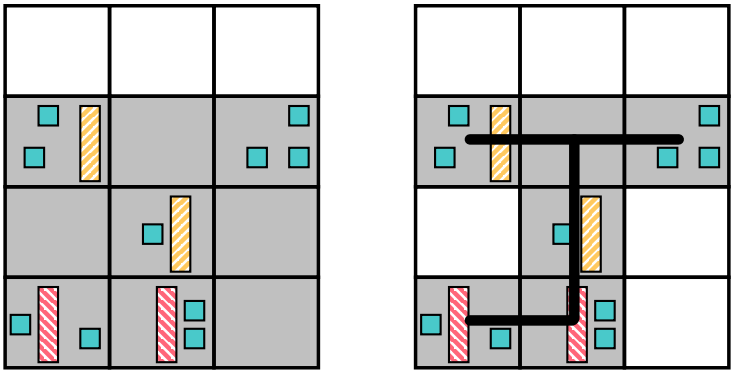


Figure 3.15: Illustration of the clock routing demand calculation using (a) the bounding box of clock loads (adopted in UTPlaceF 2.0 [43], NTUfplace [13], and RippleFPGA [66]) and (b) the actual clock tree. Both figures show the same clock network with the same load distribution. Shaded areas denote the occupied clock regions. By using bounding box modeling in (a), the clock networks occupies 9 clock regions, while the actual clock tree only spans 6 clock regions in (b).

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 solutions.

To remedy the aforementioned deficiencies in previous works, this section presents a generic framework that simultaneously optimizes placement and ensures clock feasibility by explicit clock tree construction. Inspired by the branch-and-bound idea [37], we generalize the clock legalization as a tree-space exploration process. By doing so, our framework can explore a larger

solution space and potentially produce better solutions compared with conventional greedy approaches. 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 perform experiments on the *ISPD 2017 Clock-Aware Placement Contest* [82] benchmark suite. Compared with other state-of-the-art methods in the literature, the proposed approach 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 our overall flow and details the proposed algorithms. Section 3.4.3 shows the experimental results, followed by the conclusion 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 targeting clocking architecture.*

3.4.2 UTPlaceF 2.X Algorithms

3.4.2.1 Overview of the Proposed Flow

The proposed overall flow is shown in Fig. 3.16. Our framework is built on top of a state-of-the-art academic FPGA placer presented in [45], and it consists of three major phases: (1) wirelength-driven placement, (2) clock-driven placement, and (3) legalization and detailed placement.

The wirelength-driven placement adopts the methodology of *SimPL* [34]. In each iteration of this phase, a quadratic program is solved to minimize the wirelength, and the rough legalization [34] technique is conducted to eliminate cell overlapping. This loop is repeated until the lower-bound wire-

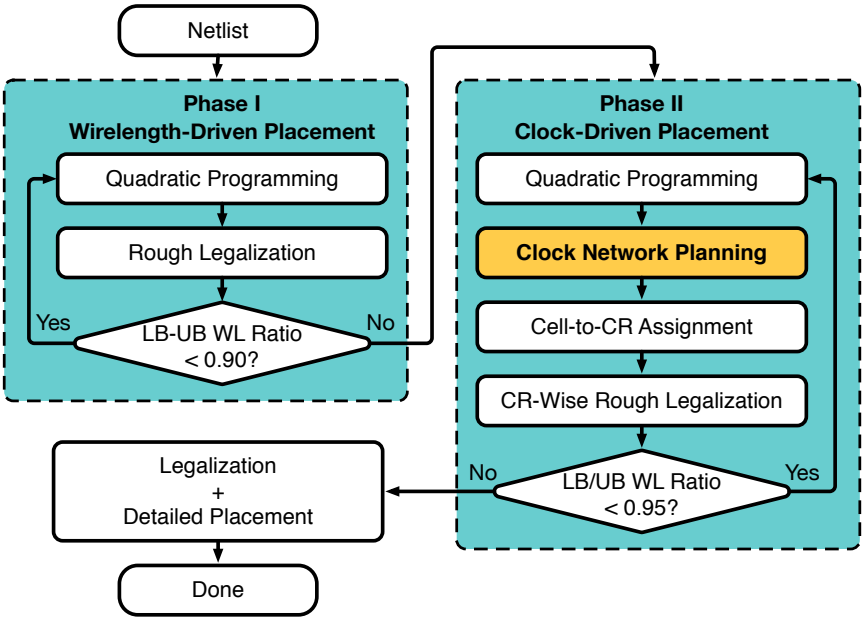


Figure 3.16: The proposed overall flow.

length and upper-bound wirelength ratio [34] (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 (Section 3.4.2). 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-wise rough legalization updates the anchor forces [34] 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 de-

tailed placement are performed to further optimize the placement result while honoring the previously achieved clock routing.

As the centerpiece of the proposed flow (Fig. 3.16), the *clock network planning* step will be elaborated later in this paper. 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 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 wirelength, 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, and (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 place-*

ment, find a movement-minimized cell-to-clock region assignment without logic resource overflow and a corresponding clock routing solution satisfying the targeting clocking architecture.

Table 3.6: 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.

Given the notations defined in Table 3.6, Problem 5 can be written as a binary optimization problem shown in Formulation (3.14). It is optimized over binary variables $\boldsymbol{x}_{v,r}$ to minimize the objective (3.14a) of total cell movement. If cell v is assigned to clock region r, then $\boldsymbol{x}_{v,r} = 1$, otherwise, $\boldsymbol{x}_{v,r} = 0$. Constraint (3.14c) guarantees that each cell is assigned to exactly one clock region. 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.

$$\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \sum_{v\in\mathcal{V}}\sum_{r\in\mathcal{R}} D_{v,r}\cdot \boldsymbol{x}_{v,r}, && \text{(3.14a)}\\
\text{subject to} \quad & \boldsymbol{x}_{v,r}\in\{0,1\}, \forall v\in\mathcal{V}, \forall r\in\mathcal{R}, && \text{(3.14b)}\\
& \sum_{r\in\mathcal{R}} \boldsymbol{x}_{v,r} = 1, \forall v\in\mathcal{V}, && \text{(3.14c)}\\
& \sum_{v\in\mathcal{V}} A_v^{(s)}\cdot \boldsymbol{x}_{v,r} \le C_r^{(s)}, \forall r\in\mathcal{R}, \forall s\in\mathcal{S}, && \text{(3.14d)}\\
& \text{Exist a legal clock routing w.r.t } \boldsymbol{x}. && \text{(3.14e)}
\end{aligned}$$

3.4.2.3 Branch-and-Bound Method

A general algorithm to solve binary optimization problems, like Formulation (3.14), is call *branch-and-bound method* [37]. 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 Fig. 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 Fig. 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 P_3, and (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 paper, we will frequently link our proposed techniques to the concepts introduced in this section for better explanation.

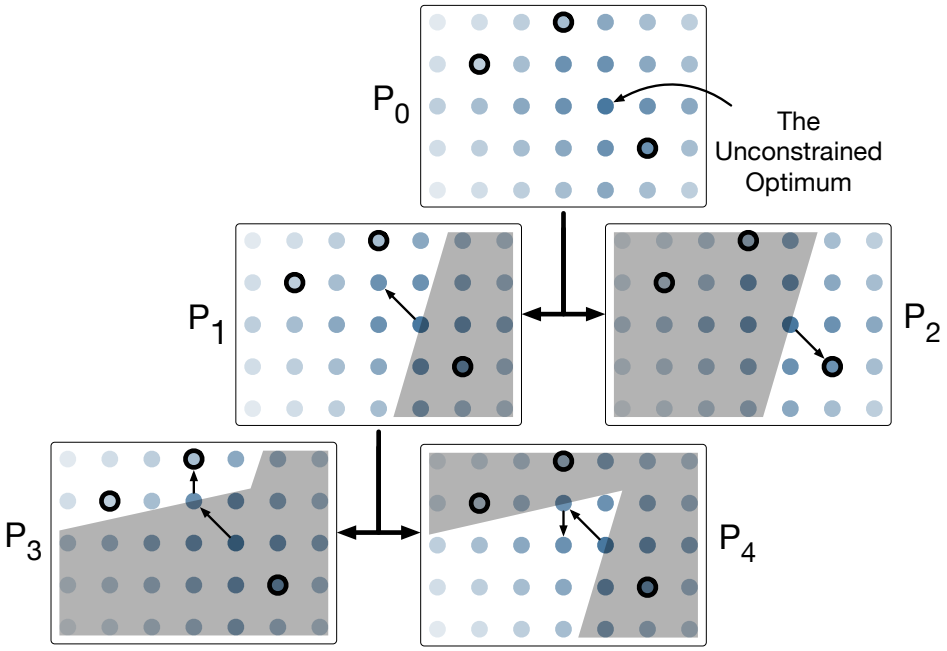


Figure 3.17: Illustration of the branch-and-bound method. Each circle denotes a solution and color intensity indicates its optimality. Feasible solutions are denoted by stroked circles.

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 3.6, 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.

In line 1 – 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. In line 3, we construct an initial clock-assignment constraint $\kappa^{(0)}$. For a clock-assignment constraint κ,

Algorithm 12: Clock Network Planning

Input : A placement with notations defined in Table 3.6. 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.

$(\mathsf{cost}^*, \boldsymbol{x}^*, \gamma^*) \leftarrow (+\infty, \mathsf{none}, \mathsf{none})$;
$\mathsf{n} \leftarrow 0$;
$\kappa_{e,r}^{(0)} \leftarrow 1, \forall e \in \mathcal{E}, \forall r \in \mathcal{R}$;
$\mathsf{stack.push}(\kappa^{(0)})$;
while stack *is not empty* ***and*** $\mathsf{n} < \mathsf{N}$ **do**
  $\kappa \leftarrow \mathsf{stack.pop}()$;
  Get the optimal cell-to-clock region assignment $\boldsymbol{x}^{(\kappa)}$ and its cost $\mathsf{cost}^{(\kappa)}$ under constraint κ (Section 3.4.2.5);
  if *no feasible* $\boldsymbol{x}^{(\kappa)}$ *exists* **then continue**;
  Get the clock routing solution $\gamma^{(\kappa)}$ corresponding to $\boldsymbol{x}^{(\kappa)}$ (Section 3.4.2.6);
  if $\gamma^{(\kappa)}$ *is overflow-free* **then**
    $\mathsf{n} \leftarrow \mathsf{n} + 1$;
    if $\mathsf{cost}^{(\kappa)} < \mathsf{cost}^*$ **then**
      $(\mathsf{cost}^*, \boldsymbol{x}^*, \gamma^*) \leftarrow (\mathsf{cost}^{(\kappa)}, \boldsymbol{x}^{(\kappa)}, \gamma^{(\kappa)})$;
    end
  end
  else if $\gamma^{(\kappa)}$ *has routing overflow* **then**
    Derive a set of more strict constraints K' from κ (Section 3.4.2.7);
    Remove $\kappa' \in K'$ that has lower-bound cost larger than cost^* (Section 3.4.2.8);
    Push $\kappa' \in K'$ into stack by their lower-bound costs from high to low;
  end
end
return $(\mathsf{cost}^*, \boldsymbol{x}^*, \gamma^*)$;

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 also 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 $\kappa^{(0)}$ 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 a depth-first order (DFS). The DFS starts with the constraint $\kappa^{(0)}$ (line 4) and repeated in line 5 – 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 κ are branched from the constraint tree rooted at $\kappa^{(0)}$, just like the branching procedure illustrated in Fig. 3.17. The best solution found during this DFS exploration is returned in line 22 as the final result.

In each execution of line 5 – 21, we first fetch the clock-assignment constraint κ on the top of the stack (line 6), then get the movement-minimized cell-to-clock region assignment constrained by logic resources and κ (line 7). This step can be interpreted as, within the sub-space κ, finding the optimal solution $\boldsymbol{x}^{(\kappa)}$ of Formulation (3.14) without considering the clock constraint (3.14e). If no such $\boldsymbol{x}^{(\kappa)}$ exists, this branch will be discarded (line 8). Otherwise, we continue to evaluate the clock feasibility of $\boldsymbol{x}^{(\kappa)}$ by constructing a clock routing solution $\gamma^{(\kappa)}$ (line 9). If $\gamma^{(\kappa)}$ is routing overflow-free, $(\mathsf{cost}^{(\kappa)}, \boldsymbol{x}^{(\kappa)}, \gamma^{(\kappa)})$ then forms a feasible solution, and we will update the best solution $(\mathsf{cost}^*, \boldsymbol{x}^*, \gamma^*)$ if needed (line 10 – 15). If $\gamma^{(\kappa)}$ still has routing overflows, we will branch new

clock-assignment constraints from κ to encourage more clock-friendly solutions (line 17). These new constraints $\kappa' \in K'$ can be interpreted as sub-spaces of κ, and some previously allowed clock assignments in κ can be blocked in $\kappa' \in K'$. 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 in line 7. Section 3.4.2.6 presents the clock routing in line 9. The clock-assignment constraint derivation in line 17 and the lower-bound cost calculation in line 18 – 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 κ. 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 3.6. Note that Formulation (3.14) and Formulation (3.15) only differ by the clock constraints (3.14e) and (3.15e).

$$
\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \sum_{v\in\mathcal{V}}\sum_{r\in\mathcal{R}} D_{v,r}\cdot \boldsymbol{x}_{v,r}, && \text{(3.15a)}\\
\text{subject to} \quad & \boldsymbol{x}_{v,r}\in\{0,1\}, \forall v\in\mathcal{V}, \forall r\in\mathcal{R}, && \text{(3.15b)}\\
& \sum_{r\in\mathcal{R}} \boldsymbol{x}_{v,r} = 1, \forall v\in\mathcal{V}, && \text{(3.15c)}\\
& \sum_{v\in\mathcal{V}} A_v^{(s)}\cdot \boldsymbol{x}_{v,r} \le C_r^{(s)}, \forall r\in\mathcal{R}, \forall s\in\mathcal{S}, && \text{(3.15d)}\\
& \boldsymbol{x}_{v,r} = 0, \forall (v,r)\in\{v\in\mathcal{V}, r\in\mathcal{R}\mid \\
& \qquad\qquad \exists e\in\mathcal{E}(v) \text{ s.t. } \kappa_{e,r}=0\}. && \text{(3.15e)}
\end{aligned}
$$

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 many mature algorithms [3]. 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 Fig. 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).

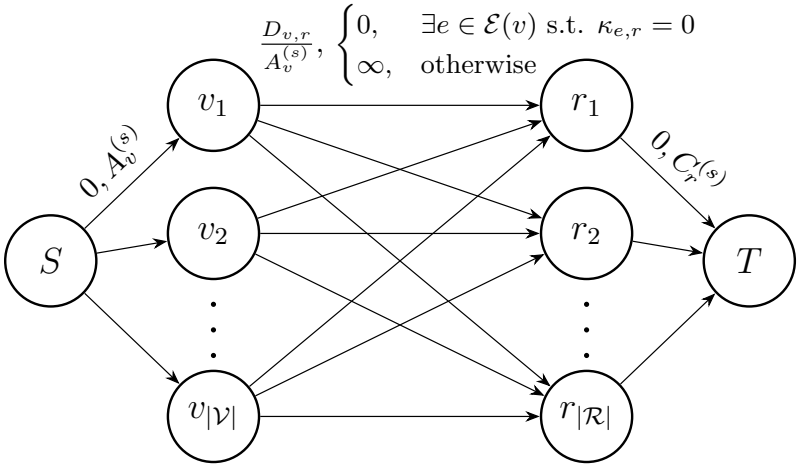


Figure 3.18: A graph representation of the minimum-cost flow for Formulation (3.15) with a single resource type. The pair of numbers on each edge denotes the unit flow cost and the flow capacity, respectively. For example, the edge between S and v_1 has a unit flow cost of 0 and a flow capacity of $A_{v_1}^{(s)}$.

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 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

[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.

(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 κ, and thus we can safely prune this branch as described in 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 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

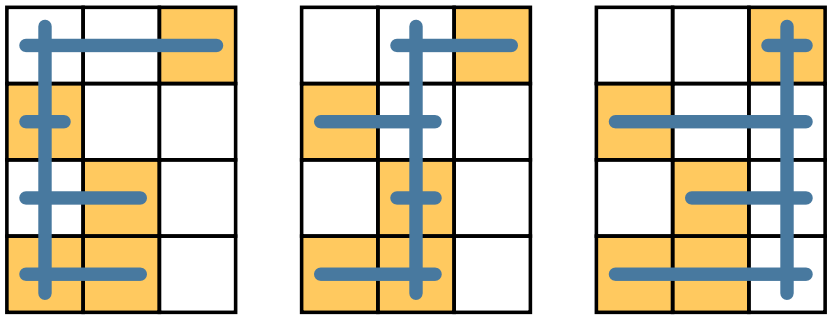


Figure 3.19: Three different D-layer clock tree topologies of the same clock load distribution on a 3×4 clock region grid. Each of them is a vertical trunk tree with horizontal branches connecting all the clock loads. Yellow shaded regions denote clock regions containing clock loads of the given clock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

As shown in Fig. 3.19, given a cell-to-clock region assignment on a clock region grid with m columns ($m = 3$ in Fig. 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 are no VD and HD overflows. Meanwhile, a topology-dependent objective (e.g., resource usage, clock skew, insertion delay, etc.) also needs to be optimized.

If we denote the set of m clock tree candidates of clock e (Fig. 3.19) 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).

$$\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \sum_{t \in \mathcal{T}} \phi_t \cdot \boldsymbol{z}_t, && \text{(3.16a)} \\
\text{subject to} \quad & \boldsymbol{z}_t \in \{0,1\}, \forall t \in \mathcal{T}, && \text{(3.16b)} \\
& \sum_{t \in \mathcal{T}(e)} \boldsymbol{z}_t = 1, \forall e \in \mathcal{E}, && \text{(3.16c)} \\
& \sum_{t \in \mathcal{T}} H_{t,r} \cdot \boldsymbol{z}_t \leq 24, \forall r \in \mathcal{R}, && \text{(3.16d)} \\
& \sum_{t \in \mathcal{T}} V_{t,r} \cdot \boldsymbol{z}_t \leq 24, \forall r \in \mathcal{R}. && \text{(3.16e)}
\end{aligned}$$

Formulation (3.16) is optimized 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 targeting 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 also be integrated in practice.

Although Formulation (3.16) can be optimally solved using integer lin-

ear 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 [21] 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.

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad \sum_{t \in \mathcal{T}} (\phi_t + \lambda_t) \cdot \boldsymbol{z}_t, \tag{3.17a}$$

$$\text{subject to} \quad \boldsymbol{z}_t \in \{0, 1\}, \forall t \in \mathcal{T}, \tag{3.17b}$$

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

Algorithm 13 summarizes our Lagrangian relaxation-based D-layer clock tree construction. In line 1 – 2, we create all the clock tree candidates $\mathcal{T}$ and initialize their penalties $\lambda^{(0)}$ to 0. In each Lagrangian iteration (line 4 – 8), we first get the optimal solution $\boldsymbol{z}^{(\mathrm{i})}$ of Formulation (3.17) with $\lambda^{(\mathrm{i})}$ (line 5). Then, $\lambda^{(\mathrm{i}+1)}$ can be derived from $\lambda^{(\mathrm{i})}$ by penalizing clock tree candidates that run through overflowed clock regions in the routing solution given by $\boldsymbol{z}^{(\mathrm{i})}$ (line 6). This iteration is repeated until one of the following holds: (1)

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.

Create all clock tree candidates $\mathcal{T}$ for $\boldsymbol{x}$ (Fig. 3.19);
$\lambda_t^{(0)} \leftarrow 0, \forall t \in \mathcal{T}$;
$\mathsf{i} \leftarrow 0$;
do
  $\boldsymbol{z}^{(\mathsf{i})} \leftarrow$ `solveLR(`$\lambda^{(\mathsf{i})}$`) // solve Formulation` (3.17)
  $\lambda^{(\mathsf{i}+1)} \leftarrow$ `updateLR(`$\boldsymbol{z}^{(\mathsf{i})}, \lambda^{(\mathsf{i})}$`) // update` λ
  $\mathsf{i} \leftarrow \mathsf{i} + 1$;
while $\boldsymbol{z}^{(\mathsf{i})}$ *has overflow* ***and*** $\lambda^{(\mathsf{i})} \neq \lambda^{(\mathsf{i}-1)}$ ***and*** $\mathsf{i} < \mathsf{I}_{\max}$;
$\boldsymbol{z}^* \leftarrow$ the $\boldsymbol{z}^{(\mathsf{i})}$ with the minimum overflow;
return $\{t \in \mathcal{T} \mid \boldsymbol{z}_t^* = 1\}$;
Function `solveLR(`λ`):`
  $\boldsymbol{z}_t \leftarrow 0, \forall t \in \mathcal{T}$;
  foreach $e \in \mathcal{E}$ **do**
    $t^* \leftarrow$ the $t \in \mathcal{T}(e)$ with the minimum $\phi_t + \lambda_t$;
    $z_{t^*} \leftarrow 1$;
  return $\boldsymbol{z}$;
Function `updateLR(`$\boldsymbol{z}^{(\mathsf{i})}, \lambda^{(\mathsf{i})}$`):`
  $\Delta\lambda_t \leftarrow 0, \forall t \in \mathcal{T}$;
  foreach $r \in \mathcal{R}$ *with HD/VD overflows of* O_H/O_V **do**
    $\mathcal{T}_H(r) \leftarrow \{t \in \mathcal{T} \mid H_{t,r} = 1\}$;
    $\mathcal{T}_V(r) \leftarrow \{t \in \mathcal{T} \mid V_{t,r} = 1\}$;
    foreach $t \in \mathcal{T}_H(r)$ **do** $\Delta\lambda_t \leftarrow \Delta\lambda_t + \frac{O_H}{|\mathcal{T}_H(r)|}$;
    foreach $t \in \mathcal{T}_V(r)$ **do** $\Delta\lambda_t \leftarrow \Delta\lambda_t + \frac{O_V}{|\mathcal{T}_V(r)|}$;
  $\alpha \leftarrow \infty$;
  foreach $e \in \mathcal{E}$ **do**
    $t^* \leftarrow$ the $t \in \mathcal{T}(e)$ being selected in iteration i;
    foreach $t \in \mathcal{T}(e)$ *that has* $\Delta\lambda_t < \Delta\lambda_{t^*}$ **do**
      $\alpha \leftarrow \mathtt{min}(\alpha, \frac{(\phi_t+\lambda_t)-(\phi_{t^*}+\lambda_{t^*})}{\Delta\lambda_{t^*}-\Delta\lambda_t})$;
  if $\alpha = \infty$ **then return** $\lambda^{(\mathsf{i})}$;
  $\lambda_t^{(\mathsf{i}+1)} \leftarrow \lambda_t^{(\mathsf{i})} + (1+\delta) \cdot \alpha \cdot \Delta\lambda_t, \forall t \in \mathcal{T}$;
  return $\lambda^{(\mathsf{i}+1)}$;

a routing overflow-free solution is found; (2) λ does not change anymore; (3) the maximum iteration count $\mathsf{I}_{\mathsf{max}}$ is reached; Finally, in line 9 – 10, we back-trace all the explored solutions and return the one with the minimum routing overflow as the final result.

The optimal solution of Formulation (3.17), as given in function `solveLR` (line 11 – 16), can be efficiently obtained by picking the candidate with the minimum $\phi_t + \lambda_t$ from each candidate pool $\mathcal{T}(e)$. Function `updateLR` (line 17 – 31) presents our λ updating scheme. In line 18 – 23, we first calculate the base penalty $\Delta\lambda_t$ for each candidate t. As shown in line 22 – 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, in line 24 – 28, we calculate the minimum scaling factor α that can change the optimal solution of Formulation (3.17) with $\alpha \cdot \Delta\lambda$ being added to current λ. If such an α does not exist, λ are kept unchanged (line 29). Otherwise, we add the extra penalty $(1 + \delta) \cdot \alpha \cdot \Delta\lambda$ to $\lambda^{(\mathsf{i})}$ ($\delta \ll 1$ is for tie-breaking) and return the result as $\lambda^{(\mathsf{i}+1)}$ (line 30 – 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 [27]-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 [54] is also applied to iteratively resolve routing overflows.

3.4.2.7 Clock-Assignment Constraint Derivation

Recall that, in 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 resource 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 current clock-assignment constraint κ to form a set of new constraints K' (line 3 – 9). If there are q clock nets occupying VD resource 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.

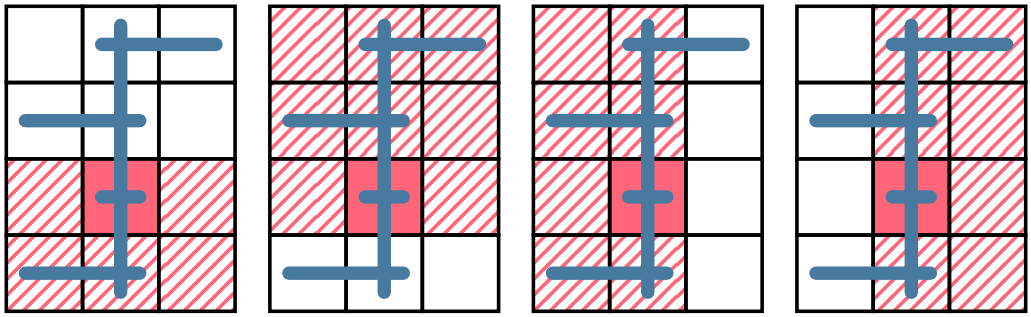


Figure 3.20: Four half-plane-based clock-assignment blockages (hatched/solid red regions) that are in the (a) south, (b) north, (c) west, and (d) east of a VD-overflowed clock region (solid red region).

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 Fig. 3.20 might be able to achieve even better results. For example, the corner-based (Fig. 3.21) and the row-based (Fig. 3.22) blockages can potentially resolve the overflow with less cell movement compared with the blockages shown in Fig. 3.20. Surprisingly, as will be shown in Section 3.4.3.3, the blockage schemes in Fig. 3.21 and Fig. 3.22 cannot outperform that in Fig. 3.20 within a limited amount of time in our experiments. This might be because the blockages in Fig. 3.21 and Fig. 3.22 tend

Algorithm 14: Clock-Assignment Constraint Derivation for VD Overflows

Input : A clock-assignment constraint κ and its clock routing solution γ.

Output: A set of new clock-assignment constraints K' derived from κ that can potentially alleviate the VD-overflow.

```
1  r ← the clock region with the most VD overflow in γ;
2  K' ← ∅;
3  foreach e ∈ E that occupies VD resource in r do
4  |   foreach blockage B in Fig. 3.20 do
5  |   |   κ' ← κ;
6  |   |   κ'_{e,b} ← 0, ∀b ∈ B;
7  |   |   K' ← K' ∪ κ';
8  |   end
9  end
10 return K';
```

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 Fig. 3.20, the placeable region of each clock is guaranteed to be a rectangle.

It is still an open question to find the best constraint derivation scheme, but we can see that our framework is generic and any other constraint derivation 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-bound cost is essential for: (1) the solution pruning in line 18 of Algorithm 12,

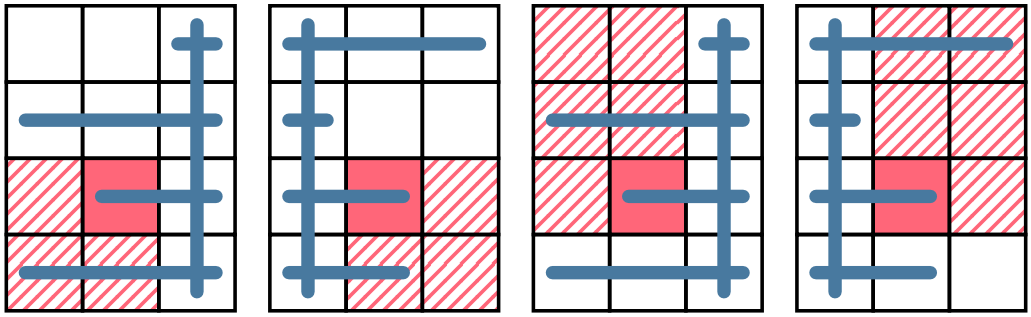


Figure 3.21: Corner-based clock-assignment blockages for an HD-overflowed clock region.

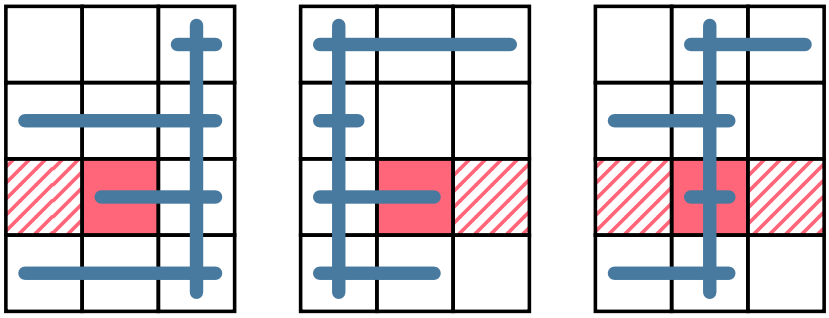


Figure 3.22: Row-based clock-assignment blockages for an HD-overflowed clock region.

and (2) the constraint sorting in 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:

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

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 computation-efficient.

3.4.3 Experimental Results

We implemented the proposed techniques in C++ based on the placement framework in [45] and performed 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 [82] released by Xilinx is used to demonstrate the effectiveness of the proposed approach. Routed wirelength reported by Xilinx Vivado v2016.4 [74] is used to evaluate the placement quality.

Table 3.7 lists the characteristics of the ISPD 2017 benchmark suite. To further demonstrate the effectiveness of the proposed approach, we also performed experiments under more strict clock constraints. Specifically, besides using all of the 24 clock routing tracks, we also conducted 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 this paper are all executed with only a single thread.

Table 3.7: 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	-

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

Table 3.8 compares our results with other state-of-the-art academic clock-aware placers, including UTPlaceF 2.0 [43], NTUfplace [35], RippleFPGA [66], and [45]. Among them, UTPlaceF 2.0, NTUfplace, and RippleFPGA are extensions of the top-3 winners of the ISPD 2017 Clock-Aware Placement Contest. Metrics "WL" and "RT" represent the routed wirelength and runtime, while "WLR" and "RTR" represent the wirelength and runtime ratios normalized to the proposed approach.

All the results of other placers are from their original publications. Since we are not able to get 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 [45] and our placer are executed with 16 and 10 threads, respectively. The runtime result of NTUfplace in Table 3.8 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 3.8 is not an apple-to-apple comparison from the clock legalization perspective. However, we still can see that our placer achieved 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 our approach in a fair way, we implemented a state-of-the-art clock legalization method UTPlaceF 2.0 [43] (it is also the 1st-place winner of the ISPD 2017 Clock-Aware Placement Contest) and replaced the proposed algorithms in our placer with it. This new placer is denoted as [43]-Impl. Since [43]-Impl and our placer 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 3.9 compares the proposed approach with [43]-Impl. It can be seen that the proposed approach achieves similar results compared with [43]-Impl under relatively loose clock constraints (CC = 24/12). As the clock capacity reduces, the proposed approach starts to outperform [43]-

Impl in both routed wirelength and runtime. On average, with CC = 8/7/6/5, [43]-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 (the proposed approach with CC = 24). However, by using the proposed approach, 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, our approach can find feasible solutions for 9 out of 13 designs within a runtime limit (1800 seconds), while only 6 designs can be successfully placed using [43]-Impl. Therefore, our approach is especially effective for cases with high clock utilization.

Table 3.8: Routed Wirelength ($\times 10^3$) and Runtime (Seconds) Comparison with Other State-of-the-Art Placers (CC = 24)

Designs	UTPlaceF 2.0 [43]				NTUfplace [35]				RippleFPGA [66]				[45]				Proposed			
	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

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

Table 3.9: Normalized Wirelength and Runtime Comparison with [43]-Impl Under Different Clock Capacities (CC)

Designs	CC = 24				CC = 12				CC = 8				CC = 7				CC = 6				CC = 5			
	[43]-Impl		Proposed		[43]-Impl		Proposed		[43]-Impl		Proposed		[43]-Impl		Proposed		[43]-Impl		Proposed		[43]-Impl		Proposed	
	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	1.002	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.

3.4.3.3 Comparison of Different Clock-Assignment Blockage Schemes

Table 3.10 compares the three clock-assignment blockage schemes described in Section 3.4.2.7: (1) half-plane-based blockages (Fig. 3.20), (2) corner-based blockages (Fig. 3.21), and (3) row-based blockages (Fig. 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 $\times$1.08 and $\times$1.33 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 observed 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 results in poor solution quality or even infeasible solution within a limited amount of time. It is still an open question to find blockage schemes better than the half-plane one (Fig. 3.20), but preserving the continuity and convexity of feasible regions should play a very important role here.

Table 3.10: 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.

3.4.3.4 Branch-and-Bound Tree Exploration

Figure 3.23 visualizes the lower-bound costs (Eq. (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 Fig. 3.23. However, due to the non-convexity of the clock network planning problem, the first solution is in general not the optimum. In this example, there are 7 solutions (in the first 30 feasible solutions) having less actual costs than the first solution (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 that simultaneously optimizes placement quality and ensures clock feasibility by explicit clock tree construction. The proposed framework significantly reduces the placement quality degradation while honoring the clock feasibility for designs with high clock utilization. To realize the proposed framework, a branch-and-bound-inspired clock network planning algorithm and a Lagrangian relaxation-based clock tree construction technique are proposed. Our experiments on ISPD 2017 benchmark suite demonstrate that the

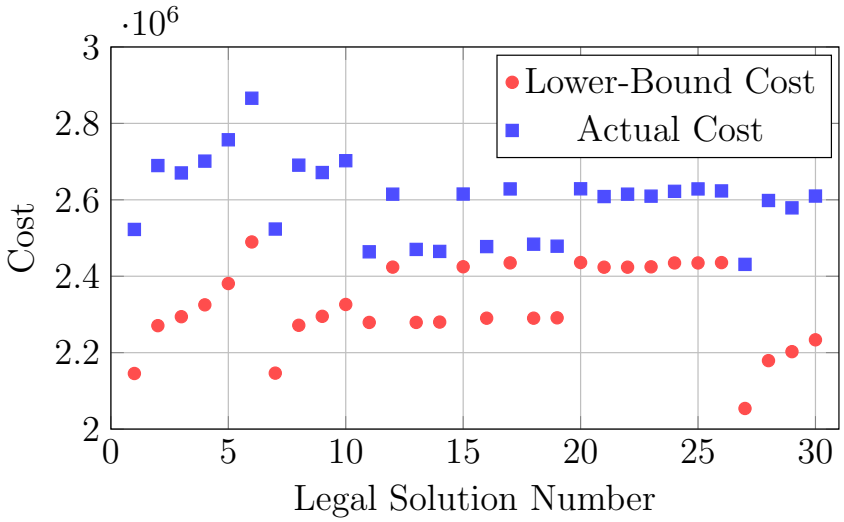


Figure 3.23: The lower-bound and actual costs of the first 30 feasible solutions found in a branch-and-bound procedure of *CLK-FPGA01* with CC = 6.

proposed approach 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 and endeavor 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 [17, 30]. 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,

---

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 developement, and experimental validations.

hence, good 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 [6,9,10], 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 min-cut-partitioning-based placers, e.g. [57, 58], which performed well for designs with tens of thousands gates. When the gate count of FPGAs arrived at the scale of millions, analytical placers, e.g. [13, 23, 24, 40, 51, 64, 65, 79, 80] started to steadily outperform min-cut-partitioning-based 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 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. SimPL [34], a quadratic placer

for application-specific integrated circuits (ASICs), achieved 1.89X 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, [48] showed that this technique cannot even achieve 2X speedup by using 16 threads in their experiments.

For nonlinear placers, wirelength and other constraints (e.g. cell density) are modeled into one single nonlinear objective, optimizing which becomes the runtime bottleneck. Unlike quadratic placers, the x and y directions in nonlinear placers cannot be decomposed into two independent components, therefore, parallelizing nonlinear placers is even harder.

A recent work, POLAR 3.0 [48], presented a quadratic placement parallelization framework for ASICs based on geometric partitioning. In particular, they divided cells into partitions based on their physical locations, then performed PCG and rough legalization [34] for each partition separately in parallel. In order to reduce solution quality loss, full-netlist placement was performed periodically to allow inter-partition cell moving. Although their approach achieved promising results (4X 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 leads to a stronger inter-partition connection and a larger quality degradation and 2) various FPGA architecture constraints (e.g. clock legalization rules [16]) impose difficulties

to the parallelization of rough legalization.

Table 4.1: Comparison of Recent FPGA and ASIC Placement Contest Benchmarks

Benchmark Suite		Avg. Num. Pins Per Mov. Cell	Avg. Num. Mov. Pins Per Net
FPGA	ISPD'16	4.99	4.95
	ISPD'17	4.16	4.15
ASIC	ICCAD'14	3.07	3.06
	ICCAD'15	3.13	2.99

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 also is 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 subproblems, which can be solved in parallel.
- We propose a parallelized incremental placement correction technique to reduce placement quality degradation.
- ISPD'16 benchmark suite demonstrates the effectiveness of our framework. On average, UTPlaceF 3.0 achieves 5X 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 $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, 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 $\mathbf{x} = \{x_1, x_2, ..., x_{|\mathcal{V}|}\}$ and $\mathbf{y} = \{y_1, y_2, ..., y_{|\mathcal{V}|}\}$ be the x and y coordinates of all cells. The wirelength-driven global placement problem is to determine cell position vectors $\mathbf{x}$ and $\mathbf{y}$ such that the total wirelength is minimized. Wirelength is measured by half-perimeter wirelength (HPWL) defined as follows.

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \sum_{e \in \mathcal{E}} \{\max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j|\}. \tag{4.1}$$

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 [78] and bound-to-bound (B2B) net model [72]. Therefore, the wirelength cost function in quadratic placers is defined as

$$W(\mathbf{x}, \mathbf{y}) = \frac{1}{2}\mathbf{x}^T Q_x \mathbf{x} + \mathbf{c}_x^T \mathbf{x} + \frac{1}{2}\mathbf{y}^T Q_y \mathbf{y} + \mathbf{c}_y^T \mathbf{y} + const. \tag{4.2}$$

Since both Hessian matrices Q_x and Q_y in Eq. (4.2) are SPD, minimiz-

ing $W(\mathbf{x}, \mathbf{y})$ is equivalent to solving the following two linear systems.

$$Q_x \mathbf{x} = -\mathbf{c}_x, \tag{4.3a}$$

$$Q_y \mathbf{y} = -\mathbf{c}_y. \tag{4.3b}$$

To eliminate cell overlapping, most state-of-the-art quadratic placers [40, 64, 65] adopted a cell-spreading technique, called rough legalization [34], 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 bottleneck of a state-of-the-art quadratic placer and evidence that achieving a highly scalable parallel placement is indeed a challenging problem.

A representative rough legalization-based quadratic placement flow is presented in Fig. 4.1. To show its runtime bottlenecks, we implemented a pure wirelength-driven UTPlaceF[1] [40] and performed experiments on ISPD'16 benchmark suites with single thread. As illustrated in Fig. 4.2, the three major runtime contributors, on average, are solving linear systems (85.1%), rough

[1]We removed routability optimization and global-move refinement in the original UTPlaceF.

legalization (7.5%), and linear system construction (7.3%), respectively.

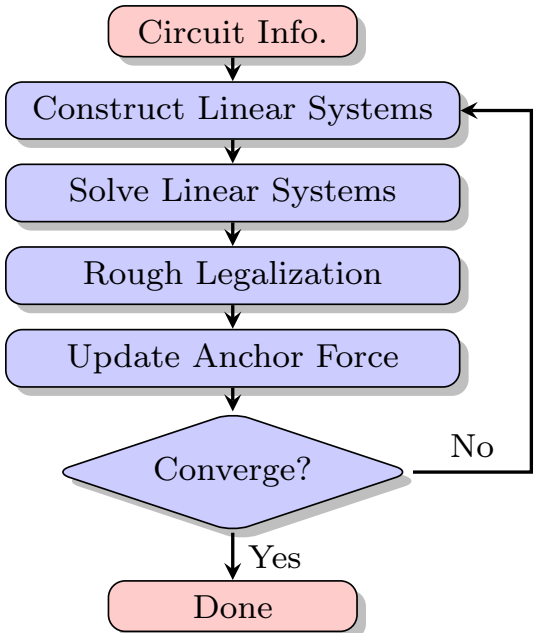


Figure 4.1: A representative rough legalization-based quadratic placement flow.

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 net-by-net manner. Considering multiple nets might contribute to the same entry in Q_x and Q_y (e.g. multiple nets share the same cell), there are two potential parallelization strategies: 1) processing nets

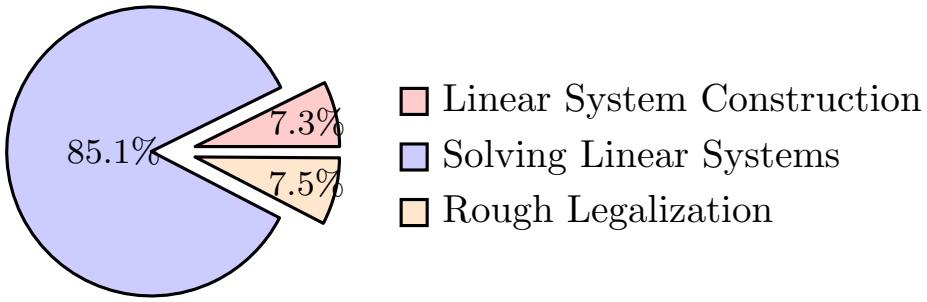


Figure 4.2: Normalized runtime of the three major runtime contributors in our pure wirelength-driven UTPlaceF on ISPD'16 benchmark suite.

in parallel and adding mutex lock[2] to each non-zero entry and 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, the most time-consuming step, is inherently hard to be parallelized as well due to the iterative nature of PCG. Experiments in [48] showed that only 1.8X speedup can be achieved by using the cutting-edge PCG solver in Intel MKL library [46] on an 8-core machine.

Unless special care is taken, rough legalization can only handle one cell density hotspots at a time, since spreading windows of multiple hotspots might intersect to each other. For each hotspot, the frequent cell sortings for horizontal and vertical cell spreading dominates the runtime. Although par-

[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.

allelism can be applied here, experiments in [34] only achieved 1.62X speedup by using 8 threads.

To manifest the placement parallelization crisis, we further parallelized our pure wirelength-driven UTPlaceF in the following way utilizing OpenMP [1]: 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, and 3) substituting sequential sortings in rough legalization with their equivalent parallel version in GNU libstdc++ parallel mode [47]. We then performed 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 ISPD'16 benchmark suite. 1, 2, 4, 8, and 16 threads were enabled respectively. Fig. 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 3X.

In summary, decomposing x and y and low-level parallelization (i.e., sortings and matrix-vector operations) alone can only achieve about 3X speedup. In this work, we will explore innovative high-level approaches to break this parallelism wall.

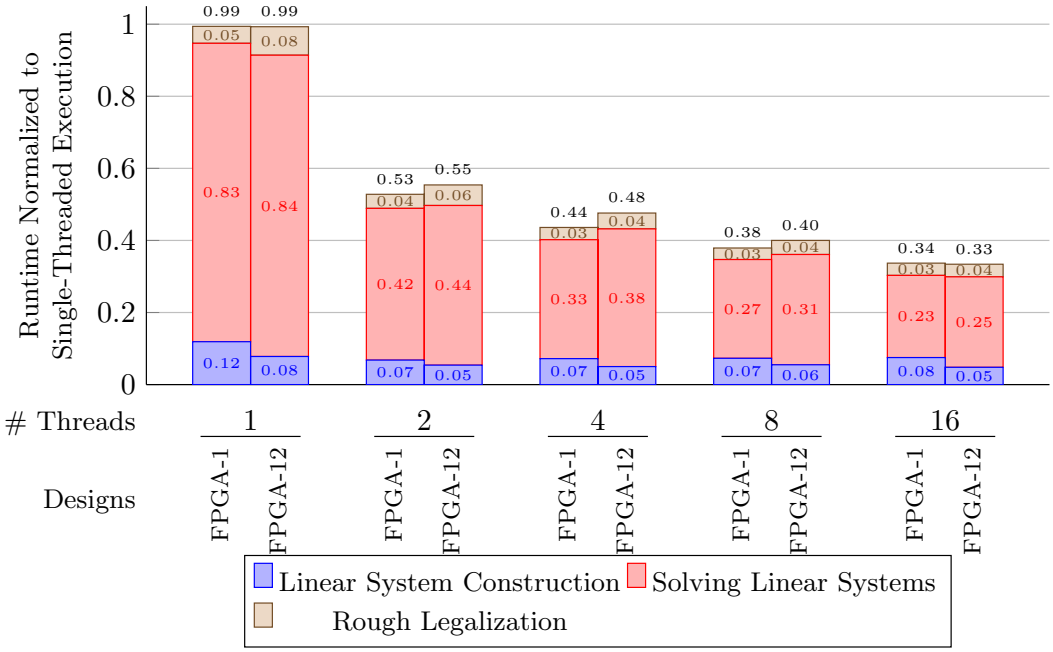


Figure 4.3: Runtime scaling of the three major runtime contributors in our pure wirelength-driven UTPlaceF by multi-threading. FPGA-1 and FPGA-12 contain 0.1M and 1.1M cells, respectively.

4.1.3 UTPlaceF 3.0 Algorithms

4.1.3.1 Overall Flow

The overall flow of UTPlaceF 3.0 is illustrated in Fig. 4.4. The whole flow starts with the initial placement consisting of one pass of linear system construction and solving followed by rough legalization and anchor force updating. After that, the netlist is divided into several sub-netlists utilizing hypergraph min-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 most likely mixed together. Once the parti-

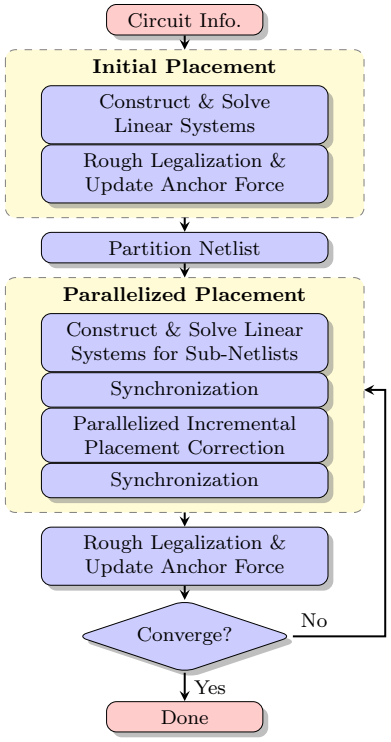


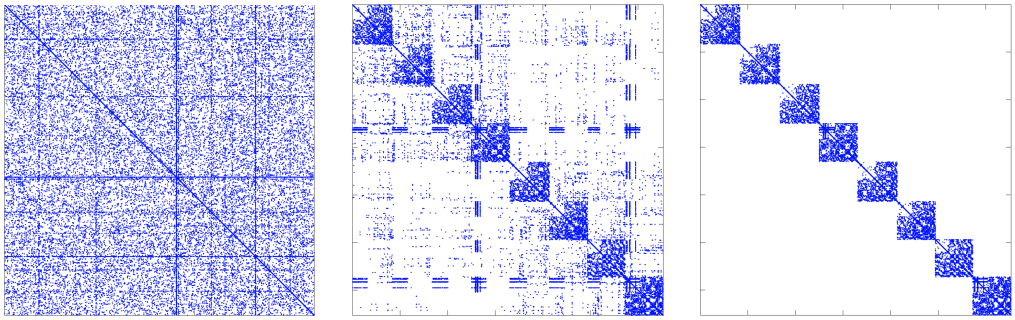
Figure 4.4: The overall flow of UTPlaceF 3.0.

tioning 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 parallelized placement, rough legalization, and anchor force updating is repeated until the wirelength con-

verges.

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. An effective PCG parallelization scheme is to leverage the strength of 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.



(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.

Fig. 4.5 shows an example of applying an 8-way block-Jacobi method on a sparse SPD matrix Q. The first step (from Fig. 4.5(a) to Fig. 4.5(b)) is to find a good row/column permutation such that the sum of off-block-diagonal non-zero entries are 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 min-cut partitioning of the induced UWG. It should be noted that the permutated Q (Fig. 4.5b) is mathematically equivalent to the original Q (Fig. 4.5a), thus, with proper permutation on decision vectors ($\mathbf{x}$ and $\mathbf{y}$) and right-hand sides ($\mathbf{c}_x$ and $\mathbf{c}_y$), a permutated linear system is also equivalent to the original one. After the permutation, the block-Jacobi preconditioner in Fig. 4.5(c) can be obtained by simply ignoring all off-block-diagonal non-zero entries in the permutated matrix in Fig. 4.5(b).

From placement point of view, each diagonal block in the block-Jacobi preconditioner corresponds to a partition (sub-netlist) and solving linear systems (4.3a) and (4.3b) with block-Jacobi preconditioner is physically equivalent to performing placement for each partition individually by assuming all off-partition cells are fixed at location $(x = 0, y = 0)$[3]. Although this approach scales almost linearly with the number of partitions created, the placement quality would 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 Fig. 4.6. We call this process

[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 $-2w_{i,j}x_ix_j$ would 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.

"placement-driven block-Jacobi preconditioning".

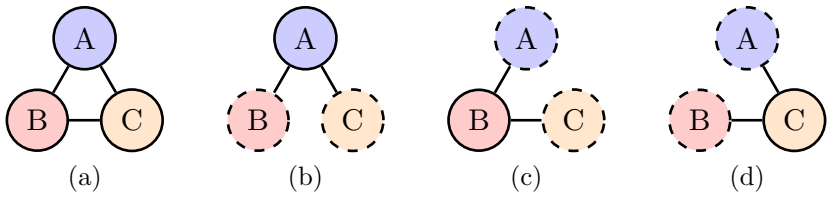


Figure 4.6: An inter-partition net spanning three partitions. (a) illustrates the net in the original netlist. (b), (c), and (d) show the net in the partition containing A, B, and C, respectively. Dashed circles represent off-partition cells that are fixed at their last locations.

Given a netlist consisting of the set of cell $\mathcal{V}$, a partitioning $\mathcal{P}$ on $\mathcal{V}$, and the $\mathcal{P}$-permutated linear systems (4.3a) and (4.3b), if we denote the belonging 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

$$\widehat{Q}_x \mathbf{x} = -\widehat{\mathbf{c}}_x, \tag{4.4a}$$

$$\widehat{Q}_y \mathbf{y} = -\widehat{\mathbf{c}}_y, \tag{4.4b}$$

then, $\widehat{Q}$ ($\widehat{Q}_x$ and $\widehat{Q}_y$) and $\widehat{\mathbf{c}}$ ($\widehat{\mathbf{c}}_x$ and $\widehat{\mathbf{c}}_y$) can be defined as

$$\widehat{Q}_{i,j} = \begin{cases} 0, & \text{if } \pi(i) \neq \pi(j), \forall i, j \in \mathcal{V}, \\ Q_{i,j}, & \text{otherwise,} \end{cases} \tag{4.5a}$$

$$\widehat{\mathbf{c}}_i = \mathbf{c}_i + \sum_{k \in \mathcal{V} \setminus \pi(i)} Q_{i,k} \cdot l_k, \forall i \in \mathcal{V}, \tag{4.5b}$$

where l_k denotes the (x or y) coordinate of cell $k \in \mathcal{V}$ in the previous placement iteration. Compared with normal 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 sys-

tems (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 quality wise, it is ideal to perform two (one each for x and y) UWG min-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 Fig. 4.4. Considering Q_x and Q_y change after every placement iteration, hypergraph min-cut partitioning is adopted to approximate the varied optimal matrix partitionings.

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 placement quality degradation introduced by partitioning, the idea of additive correction multigrid method [29] is adopted to incrementally correct the solutions of 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 are also applicable to y.

Let $\mathbf{x}^{(0)}$ be the solution of linear system (4.4a), that is

$$\widehat{Q}\mathbf{x}^{(0)} = -\widehat{\mathbf{c}}, \tag{4.6}$$

where $\widehat{Q}$ and $\widehat{\mathbf{c}}$ are defined in Eq. (4.5a) and Eq. (4.5b), respectively. If we can

find $\Delta\mathbf{x}$ satisfying

$$Q\Delta\mathbf{x} = -\mathbf{c} - Q\mathbf{x}^{(0)}, \tag{4.7}$$

the solution of the original linear system (4.3a) will be $\mathbf{x}^{(0)} + \Delta\mathbf{x}$. Intuitively, $\Delta\mathbf{x}$ represents the discrepancy between current placement $\mathbf{x}^{(0)}$ and the optimal placement $\mathbf{x}^\star$ and it is, physically, a very local and smooth perturbation around the placement $\mathbf{x}^{(0)}$.

Now we present the approach to find $\Delta\mathbf{x}$. Let $\mathbf{r}^{(0)} = -\mathbf{c} - Q\mathbf{x}^{(0)}$ and $N = \widehat{Q} - Q$. Then, solving linear system (4.7) is equivalent to solving

$$\widehat{Q}\Delta\mathbf{x} = N\Delta\mathbf{x} + \mathbf{r}^{(0)}. \tag{4.8}$$

The iterative method now becomes solving $\Delta\mathbf{x}^{(k)}$ for $k \geq 1$ with given $\Delta\mathbf{x}^{(0)}$ by

$$\widehat{Q}\Delta\mathbf{x}^{(k+1)} = N\Delta\mathbf{x}^{(k)} + \mathbf{r}^{(0)}, \tag{4.9}$$

which can be further written as

$$\Delta\mathbf{x}^{(k+1)} = \Delta\mathbf{x}^{(k)} + \widehat{Q}^{-1}(\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}). \tag{4.10}$$

Intuitively, if $\Delta\mathbf{x}^{(k)}$ and $\Delta\mathbf{x}^{(k+1)}$ are becoming closer and closer (i.e., $\Delta\mathbf{x}^{(k)}$ converges), we have $\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}$ approaches to 0. This indicates that once $\Delta\mathbf{x}^{(k)}$ converges, it will always converge to the same solution. To formally prove the convergence, consider the following lemma [67].

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*

Eq. (4.10) *is convergent for any choice of the initial* $\Delta\mathbf{x}^{(0)}$. *Moreover, the convergence of the iteration is monotone with respect to the norm* $||\cdot||_Q$ *(i.e.,* $||\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k+1)}|| < ||\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}||$, $k = 0, 1, ...$*).*

Now we can conclude the following proposition.

Proposition 4.1.2 *By picking any placement-driven block-Jacobi preconditioner of* Q *as* $\widehat{Q}$*, the iterative method defined in Eq.* (4.10) *is monotonically convergent for any choice of* $\Delta\mathbf{x}^{(0)}$.

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 Eq.* (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 Eq.* (4.10) *will converge monotonically.*

Proposition 4.1.2 reveals two important properties of the iterative method in Eq. (4.10): 1) with sufficient iterations, the exact solution, $\Delta\mathbf{x}$, of linear system (4.7) can be reached, hence, we can obtain the exact solution, $\mathbf{x}^{(0)} + \Delta\mathbf{x}$, of linear system (4.3a) and 2) since the convergence is monotone, more iterations guarantees 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 Eq. (4.10) is its strong parallelizability. Almost all the runtime of solving Eq. (4.10) is taken by computing $\widehat{Q}^{-1}(\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)})$, which is equivalent to solving

$$\widehat{Q}\mathbf{x} = \mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}. \tag{4.11}$$

Recall that $\widehat{Q}$ is the placement-driven block-Jacobi preconditioner of Q defined

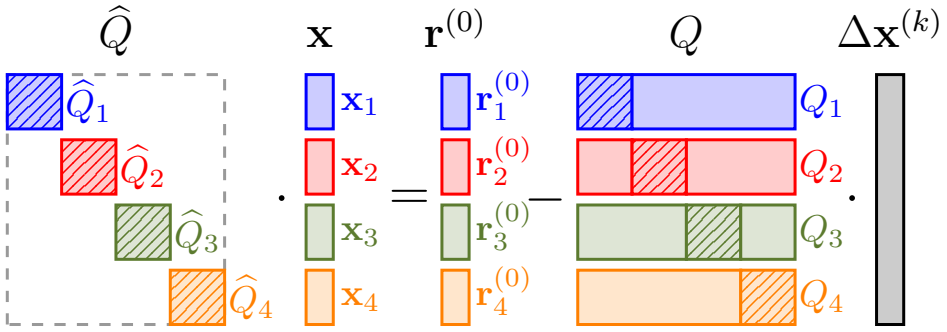


Figure 4.7: Parallelization scheme of solving linear system (4.11) with four partitions. Shaded regions represent diagonal blocks in $\widehat{Q}$ and Q. Different colors denote different partitions that can be solved in parallel.

in Eq. (4.5a). Fig. 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 ($\mathbf{x}_i$) together.

$$\widehat{Q}_i \mathbf{x}_i = \mathbf{r}_i^{(0)} - Q_i \Delta \mathbf{x}^{(k)}, \forall i \in \mathcal{P}, \tag{4.12}$$

where $\widehat{Q}_i$, x_i, $\mathbf{r}_i^{(0)}$, and Q_i are partial matrices and vectors corresponding to partition i as illustrated in Fig. 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 linear system (4.11) is obtained by assembling partial solutions together in line 10. The solution after each correction iteration is incrementally updated in line 11. The loop from line 6 to line 13 is repeated until the correction iteration limit I is reached.

4.1.3.4 Varied PIPC Configuration

In general, placement quality degrades as the number of partitions increases. So it is not wise to perform the same amount of PIPC for different number 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 accordingly.

In UTPlaceF 3.0, we configure three related variables: 1) the frequency of applying PIPC, 2) the number of correction iterations in each PIPC, and 3) the number of PCG iterations in PIPC. For small partition counts, good

Algorithm 15: Parallelized Incremental Placement Correction

Input : The set of partitions $\mathcal{P}$, the solution of linear system (4.4a) $\mathbf{x}^{(0)}$, and the number of correction iterations I.

Output : A better solution than $\mathbf{x}^{(0)}$.

parallel foreach $i \in \mathcal{P}$ **do**
  Construct Q_i (See Fig. (4.7));
end
$k \leftarrow 0$;
$\Delta\mathbf{x}^{(0)} \leftarrow 0$;
while $k < I$ **do**
  parallel foreach $i \in \mathcal{P}$ **do**
    Solve $\widehat{Q}_i \Delta\mathbf{x}_i^{(k+1)} = \mathbf{r}_i^{(0)} - Q_i \Delta\mathbf{x}^{(k)}$;
  end
  $\Delta\mathbf{x}^{(k+1)} \leftarrow (\Delta\mathbf{x}_1^{(k+1)}, \Delta\mathbf{x}_2^{(k+1)}, ..., \Delta\mathbf{x}_{|\mathcal{P}|}^{(k+1)})$;
  $\Delta\mathbf{x}^{(k+1)} \leftarrow \Delta\mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k+1)}$;
  $k \leftarrow k + 1$;
end
return $\mathbf{x}^{(0)} + \Delta\mathbf{x}^{(I)}$;

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, PIPC frequency and correction iterations need to be increased properly to control quality degradation. The third variable, PCG iteration count in PIPC, is tuned to further save runtime. Since the solution of PIPC ($\Delta\mathbf{x}$) is essentially a local perturbation of the existing placement ($\mathbf{x}^{(0)}$), its magnitude is typically much smaller than $\mathbf{x}^{(0)}$. 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 [1] is used to support multi-threading, GNU libstdc++ parallel mode [47] is used to parallelize critical sortings in rough legalization, hypergraph partitioning tool PaToh [63] is used to partition netlists, and the PCG in Eigen3 [25] 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'16 Placement Contest Benchmarks Statistics

Benchmark	#LUT	#FF	#RAM	#DSP
FPGA-1	50K	55K	0	0
FPGA-2	100K	66K	100	100
FPGA-3	250K	170K	600	500
FPGA-4	250K	172K	600	500
FPGA-5	250K	174K	600	500
FPGA-6	350K	352K	1000	600
FPGA-7	350K	355K	1000	600
FPGA-8	500K	216K	600	500
FPGA-9	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 ISPD'16 FPGA placement contest [81] 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 discarded 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

$$sHPWL = 0.5 \cdot HPWL_x + HPWL_y, \tag{4.13}$$

where $HPWL_x$ and $HPWL_y$ denote the x- and y-directed components of HPWL in Eq. (4.1), respectively.

4.1.4.1 Parallelization Configuration

Table 4.3: Parallelization Configuration of UTPlaceF 3.0 Under Different Number 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

Table 4.4: Comparison of UTPlaceF 3.0 without PIPC Under Different Number 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-1	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-2	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-3	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-4	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-5	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-6	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-7	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-8	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-9	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 compared to 1-thread execution.

† RTR: Runtime ratio compared to 1-thread execution.

Table 4.5: Comparison of UTPlaceF 3.0 with PIPC Under Different Number of Threads

Designs	8 Threads				16 Threads			
	WL	RT	WLR	RTR	WL	RT	WLR	RTR
FPGA-1	230	11	1.013	0.256	232	9	1.020	0.200
FPGA-2	448	19	1.024	0.261	453	15	1.037	0.207
FPGA-3	1719	57	1.034	0.247	1674	46	1.007	0.201
FPGA-4	3406	68	1.009	0.270	3379	51	1.001	0.204
FPGA-5	6413	85	0.986	0.290	6549	64	1.007	0.217
FPGA-6	3125	114	0.994	0.254	3203	91	1.019	0.203
FPGA-7	6160	115	1.053	0.258	6507	90	1.112	0.201
FPGA-8	5848	130	1.053	0.247	6151	103	1.107	0.197
FPGA-9	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 Number 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 Netlists	-	-	-	-	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.

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.

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) compared to corresponding 1-thread execution.

On average, without PIPC, UTPlaceF 3.0 achieves 1.7X, 2.8X, 4.4X, and 5.9X 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 5X 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 results 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.

4.1.4.3 Runtime Analysis

Table 4.6 presents the runtime breakdown of UTPlaceF 3.0 under different number of threads. We divide the total runtime into components of solving linear systems, linear system construction, rough legalization, netlists 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 speedup 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 scales poorly, 3) the runtime of netlist partitioning is almost linear to the number of partitions, and 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 [34] with our parallelization for critical sortings, and 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 for modern FPGA global placement, UTPlaceF 3.0. 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 elfPlace, 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 min-cost flow-based cell assignment algorithm. (b) Section 3.4 has described UTPlaceF 2.X, which is a generalization 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. UTPlaceF 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× speedup with 3% wirelength degradation on the industrial-strength ISPD 2016 benchmarks suite.

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 harware accelerators (e.g., FPGAs and GPUs), many conventional and emerging issues in FPGA implementations can be relieved. In particular, the following futher research directions are of great interest and potential:

- Most FPGA implementation flows are developed targeting to 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. Exisiting 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 limit. 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 methodolgy requires design automation tools

to leverage domain-specific knowledge and achieve fast compilation in a self-adaptive manner.

Bibliography

[1] OpenMP 4.0. `http://www.openmp.org/`, 2019. Accessed: 2019-4-1.

[2] 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 (TODAES)*, 23(5):66:1–66:33, 2018.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

[4] Larry C. Andrews. *Special Functions of Mathematics for Engineers*. SPIE Optical Engineering Press, 1992.

[5] 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.

[6] 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.

[7] 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 (ASPDAC), pages 629–634, 2001.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] T. C. Chen, Z. W. Jiang, T. C. Hsu, H. C. Chen, and Y. W. Chang. Ntuplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1228–1240, 2008.

[13] 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.

[14] 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.

[15] 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.

[16] ISPD 2017 clock-aware FPGA placement contest. `http://www.ispd.cc/contests/17/`. Accessed: 2017-8-1.

[17] Intel (Altera) Corp. `http://www.altera.com`. Accessed: 2017-8-1.

[18] Nima Karimpour Darav, Andrew Kennings, Kristofer Vorwerk, and Arun Kundu. Multi-commodity flow-based spreading in a commercial analytic placer. In *ACM Symposium on FPGAs*, pages 122–131, 2019.

[19] 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.

[20] 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.

[21] Marshall L Fisher. The lagrangian relaxation method for solving integer programming problems. *Management science*, 27(1):1–18, 1981.

[22] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[23] 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.

[24] 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.

[25] Gaël Guennebaud, Benoˆit Jacob, et al. Eigen3. http://eigen.tuxfamily.org.

[26] Xilinx UltraScale Architecture Clocking Resources User Guide. `https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf`. 2018.

[27] 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.

[28] Dwight Hill. Method and system for high speed detailed placement of cells within an integrated circuit design, 2002. US Patent 6,370,673.

[29] 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.

[30] Xilinx Inc. `http://www.xilinx.com`. 2019.

[31] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[32] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *VLSI Design*, 11(3):285–300, 2000.

[33] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[34] 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.

[35] 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.

[36] 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.

[37] Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[38] Claude Lemaréchal. Lagrangian relaxation. In *Computational combinatorial optimization*, pages 112–156. Springer, 2001.

[39] Wuxi Li, Mehrdad E. Dehkordi, Stephen Yang, and David Z. Pan. Simultaneous placement and clock tree construction for modern FPGAs. In *ACM Symposium on FPGAs*, pages 132–141, 2019.

[40] 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 (ICCAD)*, pages 66:1–66:7, 2016.

[41] 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.

[42] 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 (ICCAD), pages 908–914, 2017.

[43] 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 (TODAES)*, 23(4):42, 2018.

[44] 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. (Under Review).

[45] 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.

[46] Intel Math Kernel Library. `https://software.intel.com/en-us/mkl`. Accessed: 2017-8-1.

[47] GNU libstdc++ Parallel Mode. `https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html`. Accessed: 2017-8-1.

[48] Tao Lin, Chris 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] Hanyu Liu and Ali Akoglu. Timing-driven nonuniform depopulation-based clustering. Article 3, 2010.

[54] 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 (ICCAD)*, pages 713–719, 2012.

[55] 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 (TODAES)*, 20(2):17, 2015.

[56] 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.

[57] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. Fast timing-driven partitioning-based placement for island style fpgas. In *ACM/IEEE Design Automation Conference (DAC)*, pages 598–603, 2003.

[58] 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.

[59] Grzegorz Malewicz, Matthew H Austern, Aart JC 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.

[60] 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 Symposium on FPGAs*, pages 37–46, 1999.

[61] Zied Marrakchi, Hayder Mrabet, and Habib Mehrez. Hierarchical fpga clustering based on a multilevel partitioning approach to improve routability and reduce power dissapation. pages 25–28, 2005.

[62] 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.

[63] PaToH. `http://bmi.osu.edu/umit/software.html`. Accessed: 2017-8-1.

[64] 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.

[65] 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.

[66] 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.

[67] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*. Springer Science & Business Media, 2010.

[68] 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.

[69] 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.

[70] 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.

[71] Peter Spindler and Frank M Johannes. Fast and accurate routing demand estimation for efficient routability-driven placement. In *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, pages 1226–1231, 2007.

[72] 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.

[73] 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.

[74] Xilinx Vivado Design Suite. `https://www.xilinx.com/products/design-tools/vivado.html`. 2019.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] M Xu, Gary Gréwal, and Shawki Areibi. StarPlace: A new analytic method for FPGA placement. *Integration, the VLSI Journal*, 44(3):192–204, 2011.

[80] 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.

[81] 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.

[82] 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 physical design for VLSI. He has received the 1st-place awards in the FPGA placement contests of ISPD 2016 and 2017, and the Silver Medal in ACM Student Research Contest at ICCAD 2018.

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.