A New Paradigm for FPGA Placement without Explicit Packing

Wuxi Li, *Student Member, IEEE*, and David Z. Pan, *Fellow, IEEE*

***Abstract*—Placement and packing are two important but separated optimization steps in a conventional FPGA implementation flow. A packing engine clusters logic elements, like lookup tables (LUTs) and flip-flops (FFs), into configurable logic blocks (CLBs), while a placement engine determines their physical locations in FPGA layouts. This paper presents a new paradigm for FPGA placement without an explicit packing stage. In the proposed framework, the solution spaces of placement and packing are simultaneously explored in a smooth and elegant way. Our experiments on ISPD 2016 and 2017 benchmark suites demonstrate the effectiveness of the proposed framework.**

***Index Terms*—Placement, Legalization, Packing, FPGA, Parallel Algorithm**

I. Introduction

The *Field Programmable Gate Array (FPGA)*-based designs are becoming increasingly attractive for their reconfigurability, shorter time-to-market, and lower non-recurring engineering costs. Beyond the success of FPGAs in traditional applications like fast *Application Specific Integrated Circuits (ASICs)* prototyping, FPGAs have also demonstrated their applicability as hardware accelerators in modern applications, such as machine learning, deep learning, and data center.

The advance of FPGA-based designs and applications is inseparable from the support of FPGA CAD. In a traditional FPGA CAD flow, logic synthesis and technology mapping translate a design into a netlist consisting of lookup tables (LUTs), flip-flops (FFs), digital signal processors, block random access memories, and I/Os. Then, a packing engine clusters LUTs and FFs successively into architecture-legal basic logic elements (BLEs) and configurable logic blocks (CLBs). After that, placement determines the physical locations of these CLBs, followed by routing to finalize the implementation flow.

Given the significance of FPGA placement and packing in determining the overall implementation quality and efficiency, lots of research efforts have been devoted to them over the past two decades. Figure 1 summarizes several representative placement and packing flows in previous works. In the early age of FPGAs, *Pack-Place-Legalize* flows (the red path), such as [1]–[9], 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 [10]–[13]. 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 [14], [15]. 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.

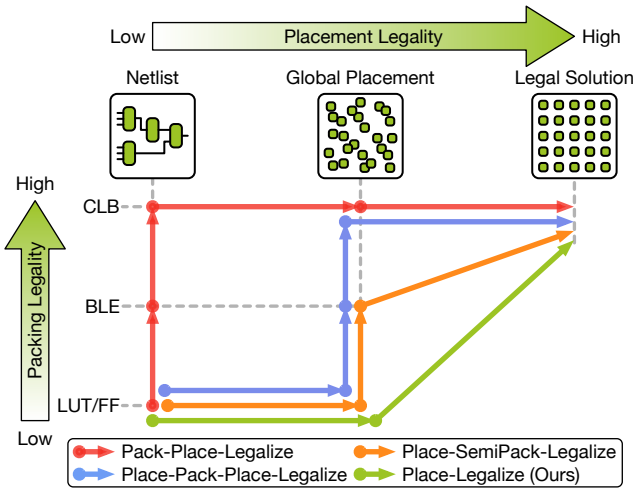


Fig. 1: Representative FPGA placement and packing flows.

Among all these methodologies, *Place-Pack-Place-Legalize* flows (the blue path) currently dominate the state-of-the-art industrial FPGA CAD tools [11]. However, large discrepancies between FIP solutions and final legal solutions can still be observed, which implies that metrics, like wirelength, timing, and routability, that are carefully optimized in FIPs can be ruined in final legal solutions. The reason is usually twofold. Firstly, most existing FIP techniques only seek to optimize the placement 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 a new paradigm for FPGA placement

This work was supported in part by Xilinx Inc.

The authors are with The Department of Electrical and Computer Engineering, The University of Texas at Austin, TX, USA. (e-mails: wuxi.li@utexas.edu; dpan@ece.utexas.edu)

without explicit packing. We call it *Place-Legalize* flow. As shown in Fig. 1, in the proposed flow (the green path), a final legal solution can be achieved directly from a FIP 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. The major contributions of this paper 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* technique that can produce legal solutions straightly from FIPs by simultaneously exploring the solution spaces of placement and packing.
- Our approach outperforms the winners of ISPD 2016 contest as well as three state-of-the-art academic placers UTPlaceF [12], RippleFPGA [14], and GPlace [16] in routed wirelength with competitive runtime on ISPD 2016 benchmark suite [17].

The rest of this paper is organized as follows. Section II reviews the FPGA architecture and quadratic placement, and formally defines the problem of direct legalization. In Section III, we point out the inherent challenges of the proposed *Place-Legalize* flow. Section IV details our proposed algorithms. Section V shows the experimental results, followed by the conclusion and future work in Section VI.

II. Preliminaries

A. FPGA Architecture

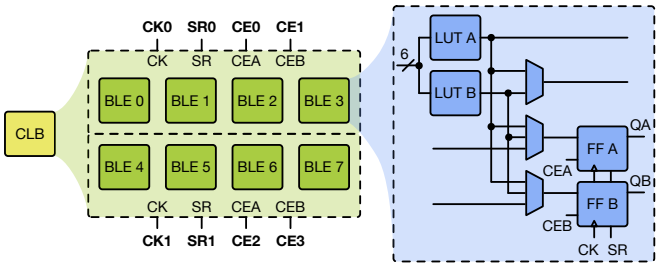


Fig. 2: CLB and BLE in Xilinx UltraScale architecture.

We adopt the Xilinx UltraScale VU095 [18] used in the ISPD 2016 FPGA placement contest [17] as the target FPGA device[1]. Its BLE and CLB architectures are shown in Fig. 2. In this particular architecture, a CLB slice consists of 8 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 2 smaller LUTs with a total number of distinct inputs no greater than 5. The 2 FFs in a BLE must share the same clock (CK) and set/reset (SR) signals, however, their clock enable (CEA and CEB) signals can be different. A CLB can be divided into two half CLBs, each of which consists of 4 BLEs that share the same set of CK, SR, CEA, and CEB.

In the rest of this paper, we will use "control set" to denote the tuple of (CK, SR, CE) for a FF (CE here is the CEA/CEB of FF A/B in Fig. 2) and the tuple of (CK, SR, CEA, CEB) for a CLB.

[1]We adopt this architecture because it is the only modern FPGA architecture that has public benchmarks, but our new paradigm can be easily adapted to other architecture as well.

B. Quadratic Placement

An FPGA netlist can be represented as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of cells, and $\mathcal{E}$ 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 placement problem is to determine cell locations such that the total half-perimeter wirelength (HPWL) defined in Eq. (1) is minimized.

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \sum_{e \in \mathcal{E}} [\max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j|]. \quad (1)$$

Quadratic placers approximate the HPWL by squared Euclidean distance between cells using various net models, such as hybrid net model [19] and bound-to-bound (B2B) net model [20]. Thus, quadratic placers minimize the wirelength cost function defined as

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

To eliminate cell overlapping, most state-of-the-art quadratic placers [12], [14], [21]–[23] adopted the idea of rough legalization [24]. The final placement solution can be obtained by iteratively solving the quadratic program in Eq. (2) and performing rough legalization until cells are fully spread out.

C. The FPGA Direct Legalization Problem

TABLE I: 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

In our *Place-Legalize* flow, the *direct legalization* (DL) is a step that produces a legal solution directly from a FIP. Given

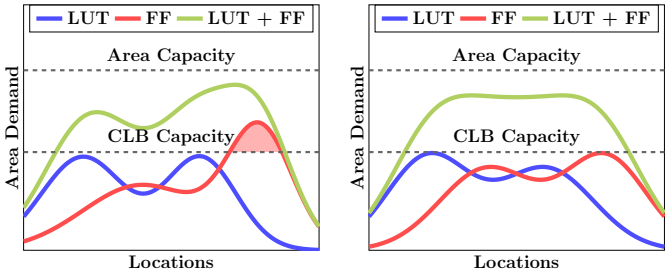


Fig. 3: (a) An area overflow-free placement but with FF demand overflow (the shaded red region). (b) A legal placement.

the notations defined in Table I, the FPGA DL problem can be defined as follows:

$$\max_{\mathbf{x},\mathbf{y}} \quad \sum_{s\in\mathcal{S}} \phi(\{v \in \mathcal{V} \mid z_{v,s} = 1\}) - \lambda \cdot \text{HPWL}(\mathbf{x},\mathbf{y}), \tag{3a}$$

$$\text{s.t.} \quad \sum_{s\in\mathcal{S}} z_{v,s} = 1, \forall v \in \mathcal{V}, \tag{3b}$$

$$\{v \mid v \in \mathcal{V}, z_{v,s} = 1\} \text{ is arch. legal}, \forall s \in \mathcal{S}, \tag{3c}$$

$$|x_v - x'_v| + |y_v - y'_v| \leq D, \forall v \in \mathcal{V}. \tag{3d}$$

The objective (3a) 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 [3], [5], [8]. 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 IV-C4. The constraint (3b) guarantees that each LUT and FF is assigned to one and only one CLB slice. The constraint (3c) assures that all the architecture rules stated in Section II-A are satisfied. The maximum displacement constraint (3d) is introduced to better preserve the FIP. We treat (3d) 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 (3a). Besides, the maximum displacement is explicitly considered. It is, however, much harder to be dealt with in traditional methods (e.g., *Place-Pack-Place-Legalize* flow).

III. Challenges of Place-Legalize Flow

Although *Place-Legalize* flows have lots of potential merits, there are several new challenges come into the field with it.

To achieve a smooth DL process, the 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 [14], [16]. 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 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. 3. 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. 3(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 3(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 [25]. 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 *Place-Legalize* 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 *Place-Legalize* flow.

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

IV. Proposed Algorithms

A. Overall Flow

The proposed overall flow is illustrated in Fig. 4. 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 adjustment step is performed after each placement iteration. This new step is aiming at resolving the two FIP-related issues pointed out in Section III. 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

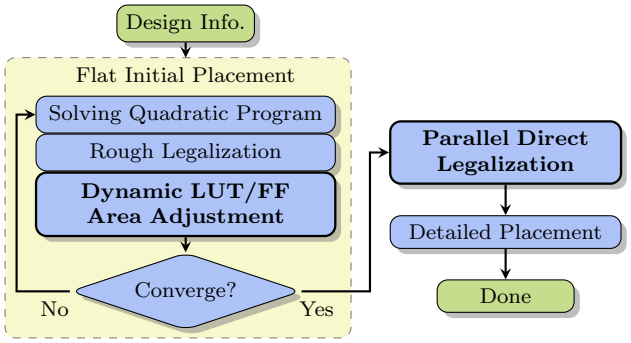


Fig. 4: The proposed overall flow.

legalization, in which the Formulation (3a) – (3d) 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 IV-B and Section IV-C, respectively.

B. 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 $\{v\} \cup \mathcal{N}_v$.

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 III, 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 III, 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:

$$U_v = \frac{\sum_{i \in \mathcal{N}_v^+} A_i}{C_v}, \tag{4}$$

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 IV-B2 (Eq. (6)) and Section IV-B3 (Eq. (7)), respectively.

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

$$a_v = \begin{cases} \min(a'_v \beta_+,\ A_v U_v \gamma_v), & \text{if } A_v U_v \gamma_v > a'_v, \\ \max(a'_v \beta_-,\ A_v U_v \gamma_v), & \text{if } A_v U_v \gamma_v < a'_v \text{ and} \\ & R_v < R_{\max}, \\ a'_v, & \text{otherwise}, \end{cases} \tag{5}$$

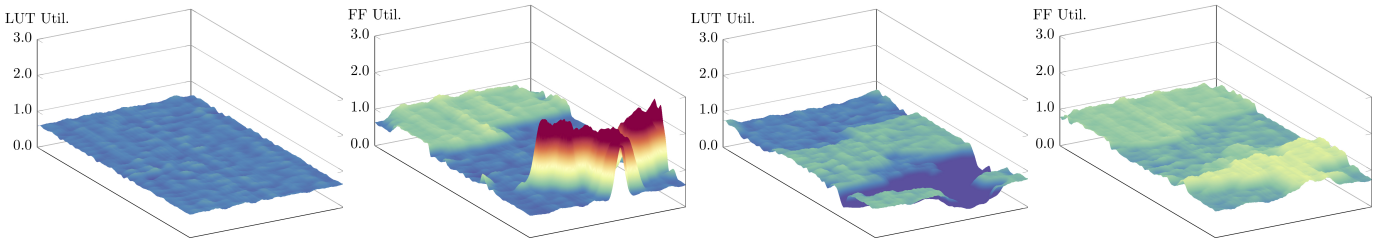
where a'_v denotes the current area of v, $\gamma_v \geq 1$ denotes the cumulative inflation ratio of v for routability optimization, R_v denotes the local routing utilization at v, and $R_{\max}$ is an empirically determined constant representing the maximum routing utilization for area shrinking. Besides,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

Intuitively, if a LUT (FF) v is hard to pack (large A_v) and is located in a region with high LUT (FF) utilization (large U_v), as well as high routing congestion (large γ_v), its area will be inflated. Otherwise, we will shrink its area to allow other cells to get in, but only when the region is not routing-congested ($R_v < R_{\max}$). To achieve a smooth area adjustment, β_+ and β_- are set to 1.1 and 0.95, respectively. The γ_v is cumulatively updated using a history-based cell inflation technique similar to [12], [14]. The routing congestion is estimated by a fast global router NCTUgr [26], 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 V-B.

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. (6) and Eq. (7)) can result in low resource usage. This issue is prevented by scaling A_v using the local resource utilization U_v in Eq. (5). 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. 3, and resolve "packing hotspots" that are harmful to the smoothness of the subsequent direct legalization process. Figure 5 shows the LUT/FF utilization (Eq. (4)) maps of a design with/without this technique when area constraint is satisfied (see Fig. 3). It can be seen that, with this technique, a huge FF hotspot (Fig. 5(b)) is resolved (Fig. 5(d)). More interestingly, the LUTs that are original in the FF hotspot (Fig. 5(a)) tend to follow the movement of their connected FFs, which results in a "valley" (Fig. 5(c)). This is actually an expected behavior, since it implicitly preserves the relative cell ordering, which is important for the placement quality.

2) LUT Resource Demand Estimation: Now we present the method to estimate the resource demand (the A_i in Eq. (4)) 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



(a) (b) (c) (d)

Fig. 5: (a) and (b) are the 3-D LUT and FF utilization (Eq. (4)) 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. 3. The experiments are based on design *FPGA-10* in ISPD 2016 benchmark suite [17].

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. (6) is used instead for a LUT v with its neighbors $\mathcal{N}_v$.

$$A_v^{(\text{LUT})} = \frac{|\widehat{\mathcal{N}}_v|}{|\mathcal{N}_v|} \cdot \frac{1}{16} + \frac{|\mathcal{N}_v| - |\widehat{\mathcal{N}}_v|}{|\mathcal{N}_v|} \cdot \frac{1}{8}, \tag{6}$$

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.

3) FF Resource Demand Estimation: The resource demand (the A_i in Eq. (4)) 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. (6) 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*[2], and define every 8 FFs that share the same (CK, SR) in a CLB slice as a *half CLB*[2]. For instance, in Fig. 2, 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. 6. Using the (CK, SR) group of (B, P) in Fig. 6 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 $\{\text{CE}_0, \text{CE}_1, \ldots, \text{CE}_m\}$ be the set of CE nets in $\mathcal{N}_v^+$. If we denote the number of FFs in $\mathcal{N}_v^+$ with the control set (CK_0, SR_0, CE_i) as n_i, for $0 \le i \le m$, the estimated FF demand of v can be expressed as follows:

$$A_v^{(\text{FF})} = \frac{1}{2} \cdot \frac{\lceil \frac{n_0}{4} \rceil}{\sum_{i=0}^{m} \lceil \frac{n_i}{4} \rceil} \cdot \left\lceil \frac{\sum_{i=0}^{m} \lceil \frac{n_i}{4} \rceil}{2} \right\rceil \cdot \frac{1}{n_0} \cdot \alpha^{(\text{FF})}. \tag{7}$$

Note that our analysis in Fig. 6 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 $\alpha^{(\text{FF})} \ge 1$ in Eq. (7) to give more flexibility to the packing. In our framework, we empirically set $\alpha^{(\text{FF})}$ to 1.1.

If we ignore the $\alpha^{(\text{FF})}$ term, the FF demand estimated by Eq. (7) is in the range $[1/16, 1/2]$, which implies that FF demands can vary up to 8× (e.g., the FF demands of (A, P, Z) and (B, Q, Y) in Fig. 6). 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. 5. 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 IV-B2 and Section IV-B3 are specific to the architecture detailed in Section II-A, the same idea is also applicable to other FPGA devices with different architectures.

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. (6) and Eq. (7)) and the required area adjustment rates. In

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

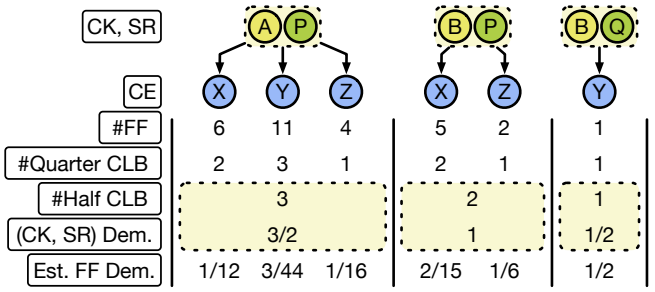


Fig. 6: Illustration of FF resource demand estimation.

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 V-B, 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 $\beta_- = 0.95$.

C. Fully Parallelizable Direct Legalization

Our direct legalization (DL) takes a rough-legal FIP and produces a legal solution by solving the Formulation (3a) – (3d). The idea is partially similar to the clustering algorithms in [11], [27], 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.

1) A College Admission Analogy: The proposed DL algorithm is inspired by the *College Admission Problem* [28]. 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) 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.

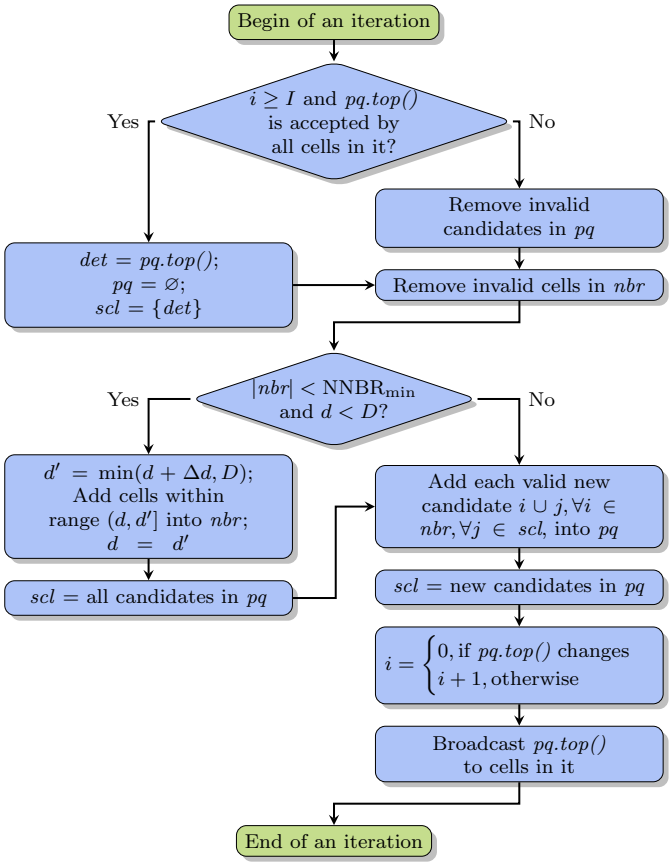


Fig. 7: The node-centric DL algorithm flow at each computation node (slice).

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. 7. Each computation node maintains a set of cells that have been determined in the slice (*det*), a priority queue of cluster candidates (*pq*), a set of neighbor cells (*nbr*), and a list of seed clusters (*scl*) for new candidate generation. The *pq* stores the K best clusters found so far at each computation node and we use K = 10. Clusters are created by adding cells in *nbr* into seed clusters in *scl* in our algorithm. Besides, a variable i is also maintained by each computation node to record the number of iterations since the last change of the best candidate in *pq*. We will use *pq.top()* to denote the best candidate in a *pq*. Before the DL starts, for each computation node, we initialize i = 0, *det* = ∅, *pq* = ∅, *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. 7, 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

$$\begin{aligned}&\Delta\text{SCORE}(s) = \\ &\quad \text{SCORE}(\textit{pq.top()} \text{ of } s, s) - \text{SCORE}(\textit{det} \text{ of } s, s),\end{aligned} \tag{8}$$

where SCORE(c, s) represents the score of cluster c in slice s, and it will be given in Section IV-C4. Intuitively, ΔSCORE(s) is the amount of score improvement by committing the *pq.top()* in slice s.

Algorithm 1 summarizes the whole fully parallelizable DL process. All computation nodes are initialized in parallel in line 1 – 7. In each DL iteration, the node-centric algorithm presented in Fig. 7 is executed by each slice individually in line 9 – 11. 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. (8)) and inform its acceptance to the selected slice in line 12 – 16. The loop from line 8 to line 17 is repeated until no more valid candidate exists.

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 1.

Algorithm 1 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 for each 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:  end parallel for
8:  while exists valid pq.top() for s ∈ S do
9:      parallel for each s ∈ S do
10:         Run the node-centric algorithm shown in Fig. 7;
11:     end parallel for
12:     parallel for each c ∈ C do
13:         Among {s ∈ S | c ∈ s.pq.top()}, pick s* that
14:         has the highest ΔSCORE(s*) defined in Eq. (8)
15:         and inform s* that c accepts its pq.top();
16:     end parallel for
17: end while
```

Lemma 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 1 basically requires a tie-breaking mechanism for the case that multiple computation nodes offer the same score improvement (Eq. (8)) 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.

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:

$$\begin{aligned}&\text{SCORE}(c, s) = \\ &\quad \sum_{e \in \text{Net}(c)} \frac{\text{InternalPins}(e, c) - 1}{\text{TotalPins}(e) - 1} - \lambda \cdot \Delta\text{HPWL}(c, s),\end{aligned} \tag{9}$$

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. (3a), 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.

5) Parallelization Scheme: As illustrated in Algorithm 1, our DL algorithm is massively parallelizable. Like colleges independently making decisions in the analogy of college admission (Section IV-C1), different computation nodes (slices) can execute the flow illustrated in Fig. 7 perfectly in parallel in each DL iteration (Algorithm 1 line 9 – 11). Moreover, cells can also process the results generated by slices and send back their decisions individually (Algorithm 1 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 V-C 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.

6) Post-DL Exception Handling: Although the convergence of our algorithm can be guaranteed by Lemma 1, after the regular DL process described in Algorithm 1, 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

$$\begin{aligned}\text{SCORE}_{\text{ripup}}(v, s, c) = \\ -\lambda_1 \cdot \Delta\text{HPWL}(v, s) - \lambda_2 \cdot \text{SCORE}(c, s) \\ -\lambda_3 \cdot \text{Area}(c),\end{aligned} \tag{10}$$

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. (9), 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. (6) and Eq. (7)). For both cases, we tend to not break it.

Algorithm 2 Post-DL Exception Handling

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

```
1:  for each c ∈ C' do
2:      D^(c) ← D;
3:      while Legalize(c, D^(c)) is fail do
4:          D^(c) ← D^(c) + 1;
5:      end while
6:  end for
7:
8:  function Legalize(c, D)
9:      ls^(c) ← {s ∈ S | dist(c, s) ≤ D};
10:     Sort slices in ls^(c) by their SCORE_ripup defined in
11:      Eq. (10) in descending order;
12:     for each s ∈ ls^(c) do
13:         if RipUpAndLegalize(s, c, D) is success then
14:             return success;
15:         end if
16:     end for
17:     return fail;
18: end function
19:
20: function RipUpAndLegalize(s, c, D)
21:     lv ← {v ∈ C | v ∈ s.det};
22:     s.det ← {c};
23:     for each v ∈ lv do
24:         ls^(v) ← {s ∈ S | dist(v, s) ≤ D and s.det ∪ v
25:                   is a legal cluster};
26:         if ls^(v) is ∅ then
27:             Remove c from s.det;
28:             Put all v ∈ lv back to s.det;
29:             return fail;
30:         else
31:             Pick s* ∈ ls^(v) with the highest
32:             SCORE(s*.det ∪ v, s*) − SCORE(s*.det, s*);
33:             s*.det ← s*.det ∪ v;
34:         end if
35:     end for
36:     return success;
37: end function
```

Algorithm 2 summarizes our post-DL exception handling technique that legalizes illegal cells after the regular DL process (Algorithm 1). In the main loop (line 1 – 6), 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 8 – 18), we first collect the set of slices $ls^{(c)}$ that are within distance D w.r.t the location of c in FIP (line 9). Then we sort all slices in $ls^{(c)}$ by their SCORE$_{\text{ripup}}$ defined in Eq. (10) in descending order (line 10 – 11). 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 12 – 16). If c cannot be legalized by breaking any slice in $ls^{(c)}$, `Legalize`(c, D) will return *fail* (line 17) and the displacement constraint will be relaxed in the main loop (line 3 – 5).

The details of function `RipUpAndLegalize`(s, c, D) are given in line 20 – 37. 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 22). Then, the cells that are originally in $s.det$ are legalized sequentially in the loop from line 23 to line 35. For each such a cell v, we first collect the set of slices ($ls^{(v)}$) that are within distance D (w.r.t the location of v in FIP) and have their *det* compatible with with v (line 24 – 25). If no such slice exists, we discard all the changes that have been made in this function call (line 27 – 28) and return *fail* (line 29). 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 31 – 33). The function call successes only when all cells that are originally in $s.det$ can find legal positions within displacement D (line 36).

7) Extension to Clock-Aware Placement: Clock networks in modern FPGAs can impose extra layout constraints during placement stage. The most common one is "clock region constraint" [29], which only allows a limited number of clock nets occupying each clock region (each clock region is a predefined rectangular region in the layout). Some previous works [15], [30] honor this constraint by finding a legal clock-to-clock region assignment and restricting clock sinks (e.g., Flip-Flops, DSPs, and RAMs) to follow it.

Here we assume such a legal clock-to-clock region assignment is given, that is, we know which cell can be placed into which slice. Then, to honor the clock region constraint, we only need to perform an extra checking step to reject cell-to-slice assignments that violate the given clock assignment in our DL algorithm.

V. Experimental Results

We implemented the proposed framework in C++ based on UTPlaceF [12] 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 [31] is used to support multi-threading. The benchmark suite released by Xilinx for ISPD 2016 FPGA placement contest [17] is used to validate the the effectiveness of the proposed approaches. All the routings are conducted by Xilinx Vivado v2015.4 [32]. The characteristics of ISPD 2016 benchmark suite are listed in Table II.

To study the impact of clock constraints, we also integrate the proposed approaches to a clock-aware placer UTPlaceF 2.0 [30], and perform experiments on the benchmark suite released by Xilinx for ISPD 2017 FPGA clock-aware placement contest [17]. The routings of this benchmark suite are conducted by Xilinx Vivado v2016.4. The characteristics of ISPD 2017 benchmark suite are listed in Table III.

TABLE II: ISPD 2016 Contest Benchmarks Statistics

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

TABLE III: ISPD 2017 Contest Benchmarks Statistics

Benchmark	#LUT	#FF	#RAM	#DSP	#Clocks
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	-

A. Effectiveness Validation of Proposed Techniques

Table IV 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 [12] flow. Column "UTPlaceF + DAA" applys DAA on top of the original UTPlaceF. Column "Proposed" 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. (9) used by the proposed DL, and it legalizes one cell at a time to maximize the score improvement defined in Eq. (8). The results of substituting the proposed DL in "Proposed" 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 "Proposed" 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

TABLE IV: Routed Wirelength (in 10^3) and Runtime (in Seconds) Comparison with UTPlaceF† [12]

Designs	UTPlaceF† [12]				UTPlaceF† + DAA				DAA + Greedy DL				Proposed (DAA + 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 [12].

portion of cells can fail to find nearby legal positions. Therefore, we always employ the proposed DL together with the DAA technique in our experiments.

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

We first compare 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 V, which we will discuss in details later in this section.

We then compare columns "UTPlaceF + DAA" and "DAA + Greedy DL" with "Proposed" 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, "Proposed" outperforms "UTPlaceF + DAA" and "DAA + Greedy" by 3.0% and 6.0%, respectively, in routed wirelength. It should be noted that "Proposed" 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 "Proposed" 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 V 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 IV. 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.

TABLE V: Displacement Comparison with UTPlaceF

Designs	UTPlaceF [12]		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

Figure 8 visualizes the distributions of cell displacement between the FIPs and the post-legalization/DL solutions based on design *FPGA-03*. Compared with UTPlaceF, most cells in the proposed methodology are much closer to their original locations in the FIP. However, tremendously large displacements can be observed in UTPlaceF.

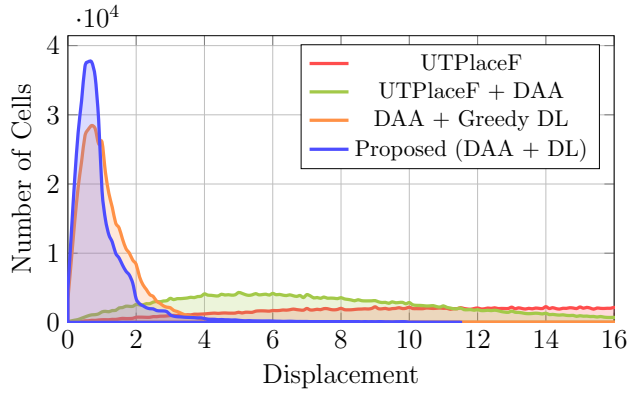


Fig. 8: The cell displacement distributions of UTPlaceF, UTPlaceF + DAA, DAA + Greedy DL, and the proposed flow (DAA + DL) based on design *FPGA-03*. The displacement of a cell is defined as the Manhattan distance between its locations in the FIP and the post-legalization/DL placement. The average/maximum 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.

Table VI shows the impact of DAA on HPWL and the maximum LUT and FF resource utilization (Eq. (4)) 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. 9. It shows the HPWL after the flat initial placement (FIP), legalization/DL (LG), and detailed placement (DP) stages in the four previously described flows. In spite of the larger HPWL after FIP, flows with DAA applied finally achieve better solutions with smoother wirelength convergence. Besides, we can also see that the proposed DL technique largely preserves the FIP solution in the LG stage, and this is the main reason that the proposed flow can significantly outperform the other three methodologies.

B. 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. (5), should be set.

Figure 10 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. 10

TABLE VI: HPWL and the Maximum LUT and FF Utilizations (Eq. (4)) w/ and w/o DAA after FIP

Designs	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

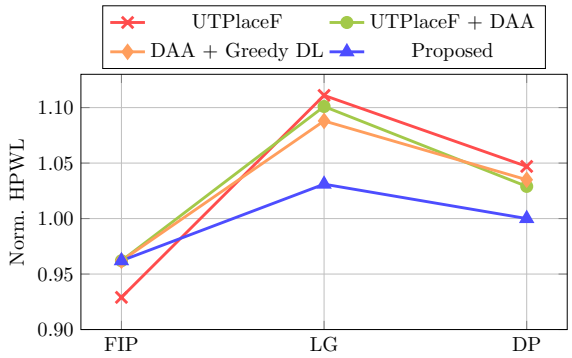


Fig. 9: The normalized HPWL after the flat initial placement (FIP), legalization/DL (LG), and detailed placement (DP) in the four methodologies listed in Table IV. All HPWL values are normalized to the post-DP HPWL of the proposed flow.

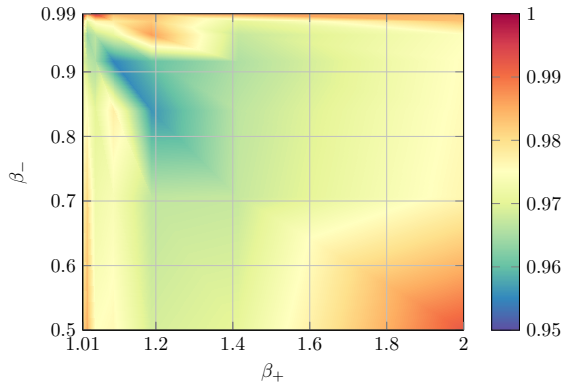


Fig. 10: Normalized HPWL under different β_+ and β_- in Eq. (5) based on design *FPGA-01*. All wirelengths are normalized to the solution without applying dynamic area adjustment.

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

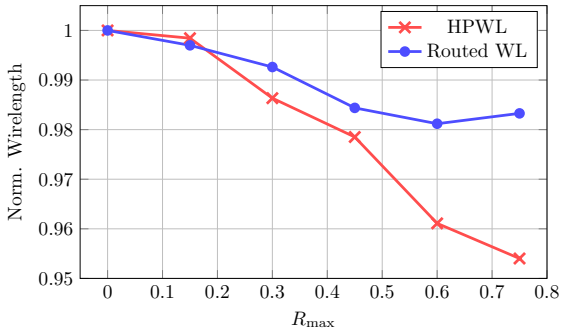


Fig. 11: Normalized HPWL and routed wirelengths under different $R_{\max}$ in Eq. (5) based on design *FPGA-05*. All wirelengths are normalized to the solution with $R_{\max} = 0$ (i.e. area shrinking is disallowed).

Figure 11 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 $R_{max} = 0.65$.

C. Runtime Scaling of the Direct Legalization

Figure 12 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.

D. 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 [14], GPlace [16] as well as the top-3 winners of ISPD 2016 contest, on ISPD

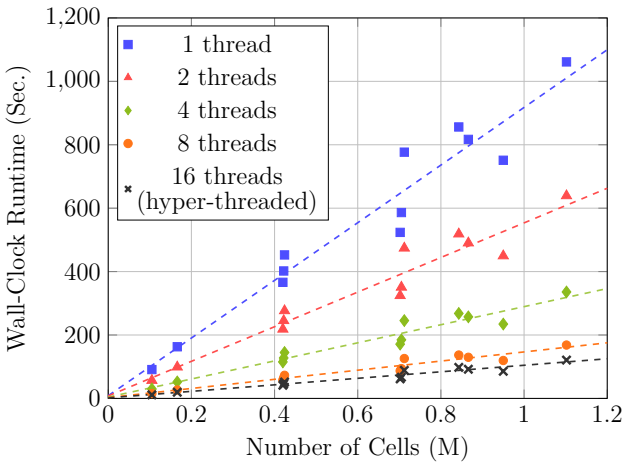


Fig. 12: Runtime scaling of the direct legalization.

2016 benchmark suite. The comparisons of routed wirelength and runtime are presented in Table VII.

As the experiment setup in Section V-A, 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 [14].

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.

To demonstrate the capability of tackling clock constraint in the proposed approaches, we also integrate them into a clock-aware placer UTPlaceF 2.0 [30]. Then, we compare it with the original UTPlaceF 2.0, the clock-aware version of RippleFPGA [15], [13] as well as the top-3 winners of ISPD 2017 contest (the original UTPlaceF 2.0 is also the 1st-place contest winner), on ISPD 2017 benchmark suite. The comparisons of routed wirelength and runtime are shown in Table VIII.

We first compare the original UTPlaceF 2.0 and the proposed flow, which is the only apple-to-apple comparison here that decouples other irrelevant differences (e.g., global/detailed placement and clock legalization algorithms). On average, the proposed approaches achieve 2.8% better routed wirelength with slightly shorter runtime. Compared with the results on ISPD 2016 benchmark suite shown in Table IV, both wirelength and runtime improvements in ISPD

TABLE VII: Routed Wirelength (in 10^3) and Runtime (in Seconds) Comparison with Other State-of-the-Art Academic Placers on ISPD 2016 Benchmark Suite

Designs	1st Place				2nd Place				3rd Place				GPlace [16]				RippleFPGA [14]				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.

TABLE VIII: Routed Wirelength (in 10^3) and Runtime (in Seconds) Comparison with Other State-of-the-Art Academic Placers on ISPD 2017 Benchmark Suite

Designs	UTPlaceF 2.0 [30] (1st Place)				2nd Place				3rd Place				RippleFPGA [15]				[13]				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
CLK-FPGA01	2208	422	1.051	0.89	2209	3023	1.052	6.35	2269	354	1.080	0.74	2011	288	0.958	0.61	2098	3524	0.999	-	2101	476	1.000	1.00
CLK-FPGA02	2279	407	1.007	0.90	2274	3153	1.005	6.94	2504	333	1.107	0.73	2168	266	0.958	0.59	2173	3351	0.960	-	2263	454	1.000	1.00
CLK-FPGA03	5353	824	1.033	0.89	6229	4066	1.202	4.37	5803	666	1.120	0.72	5265	583	1.016	0.63	5049	6722	0.974	-	5181	930	1.000	1.00
CLK-FPGA04	3698	564	1.012	0.86	3817	3077	1.045	4.69	4086	464	1.118	0.71	3607	380	0.987	0.58	3710	5101	1.015	-	3654	656	1.000	1.00
CLK-FPGA05	4692	744	1.023	0.88	4995	3631	1.089	4.29	5181	680	1.129	0.80	4660	569	1.016	0.67	4523	6336	0.986	-	4589	846	1.000	1.00
CLK-FPGA06	5589	845	1.040	0.88	5606	3836	1.043	3.98	6217	695	1.157	0.72	5737	591	1.067	0.61	5169	7932	0.962	-	5375	963	1.000	1.00
CLK-FPGA07	2445	670	0.999	1.30	2505	3953	1.023	7.68	2676	410	1.093	0.80	2326	304	0.950	0.59	2380	4071	0.972	-	2448	515	1.000	1.00
CLK-FPGA08	1886	419	1.031	0.96	1990	4395	1.088	10.08	2057	277	1.125	0.64	1778	247	0.972	0.57	1843	3109	1.008	-	1829	436	1.000	1.00
CLK-FPGA09	2597	668	1.016	1.28	2583	5428	1.011	10.38	2814	414	1.101	0.79	2530	327	0.990	0.63	2499	4423	0.978	-	2556	523	1.000	1.00
CLK-FPGA10	4464	772	1.049	0.96	4770	3305	1.121	4.13	4840	516	1.137	0.64	4496	512	1.057	0.64	4294	6569	1.009	-	4255	801	1.000	1.00
CLK-FPGA11	4184	847	1.042	1.25	4208	4341	1.048	6.39	4777	548	1.190	0.81	4190	455	1.044	0.67	4031	6538	1.004	-	4014	679	1.000	1.00
CLK-FPGA12	3369	614	1.036	0.95	3377	4949	1.038	7.65	3740	413	1.150	0.64	3388	409	1.041	0.63	3244	5300	0.997	-	3253	647	1.000	1.00
CLK-FPGA13	3848	929	1.031	1.25	3921	3748	1.051	5.04	4320	548	1.158	0.74	3833	441	1.027	0.59	3818	5639	1.023	-	3731	743	1.000	1.00
Norm.	-	-	1.028	1.02	-	-	1.063	6.31	-	-	1.128	0.73	-	-	1.006	0.62	-	-	0.991	-	-	-	1.000	1.00

*: [13] only reported the total runtime of placement and routing, so we only list their total runtime (RT) here for reference but do not show the runtime ratio (RTR).

2017 benchmark suite turn out to be less. This is mainly because that, in the ISPD 2017 benchmark suite, no severely unbalanced resource distributions, like shown in Fig. 5, are observed even without our DAA technique. Hence, the overall improvement from DAA in this benchmark suite is reduced.

On average, the proposed approach outperforms the top-3 contest winners and RippleFPGA in routed wirelength by 2.8%, 6.3%, 12.8%, and 0.6%, respectively. It is also 1.02× and 6.31× faster than the top-2 contest winners, while runs 1.37× and 1.61× slower than the 3rd-place contest winner and RippleFPGA. Compared with [13], the proposed approach is 0.9% worse in routed wirelength. However, [13] does not report their placement runtime, so we cannot quantitativly compare the runtime. Considering [13] uses a NTUplace [33]-like non-linear placement engine, we can expect that it runs substantially slower than our quadratic placement based flow.

On both ISPD 2016 and 2017 benchmark suites, RippleFPGA achieves very competitive results compared with our approaches. It turns out that RippleFPGA adopts a methodology that is very similar to us, but there are following two key differences: 1) RippleFPGA empirically and statically set cell areas in their flat initial placement, while we dynamically adjust them using the proposed DAA technique; 2) RippleFPGA legalizes cells in a greedy Tetris-based manner, while we explore a much larger solution space and legalize cells using the proposed DL algorithm. As a result, compared with RippleFPGA, our approach can potentially achieve better solution quality especially on challenging designs, but with slower runtime.

E. Runtime Breakdown

The runtime breakdown of our approach based on all twelve designs in ISPD 2016 benchmark suite is shown in Figure 13. 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.

VI. Conclusion And Future Work

In this paper, 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 and 2017 benchmark suites demonstrate the effectiveness of the proposed approach.

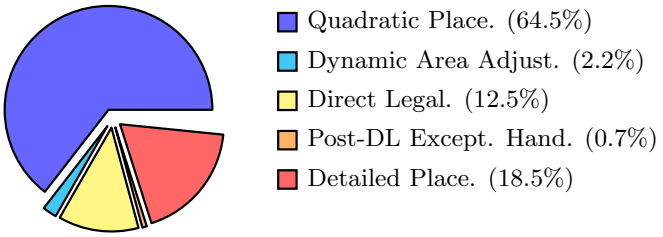


Fig. 13: The runtime breakdown of our approach based on designs in ISPD 2016 benchmark suite.

In the future, we plan to further improve the resource demand estimation models and implement the fully parallelizable direct legalization algorithm on GPU/FPGA platforms. 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.

REFERENCES

[1] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL*, 1997, pp. 213–222.

[2] ——, "Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size," in *IEEE Custom Integrated Circuits Conference (CICC)*, 1997, pp. 551–554.

[3] A. S. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in *FPGA*, 1999, pp. 37–46.

[4] E. Bozorgzadeh, S. Ogrenci-Memik, and M. Sarrafzadeh, "RPack: routability-driven packing for cluster-based FPGAs," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2001, pp. 629–634.

[5] A. Singh, G. Parthasarathy, and M. Marek-Sadowska, "Efficient circuit clustering for area and power reduction in FPGAs," *ACM TODAES*, vol. 7, no. 4, pp. 643–663, 2002.

[6] Z. Marrakchi, H. Mrabet, and H. Mehrez, "Hierarchical FPGA clustering based on a multilevel partitioning approach to improve routability and reduce power dissapation," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2005, pp. 25–28.

[7] H. Liu and A. Akoglu, "Timing-driven Nonuniform Depopulation-based Clustering," *International Journal of Reconfigurable Computing*, Article 3, 2010.

[8] S. T. Rajavel and A. Akoglu, "MO-Pack: Many-objective clustering for FPGA CAD," in *DAC*, 2011, pp. 818–823.

[9] W. Feng, "K-way partitioning based packing for FPGA logic blocks without input bandwidth constraint," in *IEEE International Conference on Field-Programmable Technology (FPT)*, 2012, pp. 8–15.

[10] D. T. Chen, K. Vorwerk, and A. Kennings, "Improving timing-driven FPGA packing with physical information," in *FPL*, 2007, pp. 117–123.

[11] L. Singhal, M. A. Iyer, and S. Adya, "LSC: A Large-Scale Consensus-Based Clustering Algorithm for High-Performance FPGAs," in *DAC*, 2017, pp. 30:1–30:6.

[12] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," *IEEE TCAD*, 2017.

[13] Y.-C. Kuo, C.-C. Huang, S.-C. Chen, C.-H. Chiang, Y.-W. Chang, and S.-Y. Kuo, "Clock-aware placement for large-scale heterogeneous fpgas," in *ICCAD*, 2017, pp. 519–526.

[14] G. Chen, C. W. Pui, W. K. Chow, K. C. Lam, J. Kuang, E. F. Y. Young, and B. Yu, "RippleFPGA: Routability-driven simultaneous packing and placement for modern FPGAs," *IEEE TCAD*, 2017.

[15] C.-W. Pui, G. Chen, Y. Ma, E. F. Y. Young, and B. Yu, "Clock-aware ultrascale fpga placement with machine learning routability prediction," in *ICCAD*, 2017, pp. 915–922.

[16] R. Pattison, Z. Abuowaimer, S. Areibi, G. Grewal, and A. Vannelli, "GPlace: A congestion-aware placement tool for ultrascale FPGAs," in *ICCAD*, 2016, pp. 68:1–68:7.

[17] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal, "Routability-driven fpga placement contest," in *ISPD*, 2016, pp. 139–143.

[18] Xilinx Virtex UltraScale Product Table, https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html#productTable, accessed: 2018-2-21.

[19] N. Viswanathan and C. C. Chu, "FastPlace: efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model," *IEEE TCAD*, vol. 24, no. 5, pp. 722–733, 2005.

[20] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Kraftwerk2-A fast force-directed quadratic placement approach using an accurate net model," *IEEE TCAD*, vol. 27, no. 8, pp. 1398–1411, 2008.

[21] T. Lin, C. C. Chu, J. R. Shinnerl, I. Bustany, and I. Nedelchev, "POLAR: Placement based on novel rough legalization and refinement," in *ICCAD*, 2013, pp. 357–362.

[22] M.-C. Kim and I. L. Markov, "ComPLx: A Competitive Primal-dual Lagrange Optimization for Global Placement," in *DAC*, 2012, pp. 747–752.

[23] W. Li, M. Li, J. Wang, and D. Z. Pan, "UTPlaceF 3.0: A Parallelization Framework for Modern FPGA Global Placement," in *ICCAD*, 2017, pp. 908–914.

[24] M.-C. Kim, D.-J. Lee, and I. L. Markov, "SimPL: An Effective Placement Algorithm," *IEEE TCAD*, vol. 31, no. 1, pp. 50–60, 2012.

[25] D. Hill, "Method and system for high speed detailed placement of cells within an integrated circuit design," 2002, uS Patent 6,370,673.

[26] W.-H. Liu, Y.-L. Li, and C.-K. Koh, "A fast maze-free routing congestion estimator with hybrid unilateral monotonic routing," in *ICCAD*, 2012, pp. 713–719.

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM Conference on Management of Data (SIGMOD)*, 2010, pp. 135–146.

[28] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, 1962.

[29] S. Yang, C. Mulpuri, S. Reddy, M. Kalase, S. Dasasathyan, M. E. Dehkordi, M. Tom, and R. Aggarwal, "Clock-Aware FPGA Placement Contest," in *ISPD*, 2017, pp. 159–164.

[30] W. Li, Y. Lin, M. Li, S. Dhar, and D. Z. Pan, "UTPlaceF 2.0: A high-performance clock-aware FPGA placement engine," *ACM TODAES*, 2017.

[31] OpenMP 4.0, http://www.openmp.org/, accessed: 2018-2-21.

[32] Xilinx Vivado Design Suite, https://www.xilinx.com/products/design-tools/vivado.html, accessed: 2018-2-21.

[33] 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 TCAD*, vol. 27, no. 7, pp. 1228–1240, 2008.

Wuxi Li (S'18) received the B.S. degree in microelectronics from Shanghai Jiao Tong University, Shanghai, China, in 2013. He is currently pursuing the Ph.D. degree at the Department of Electrical and Computer Engineering, University of Texas at Austin. His research interests include physical design automation for FPGAs. He was a recipient of the 1st-place awards in the FPGA placement contests of ISPD 2016 and 2017.

David Z. Pan David Z. Pan (IEEE Fellow, SPIE Fellow) is currently Engineering Foundation Professor at The University of Texas at Austin. His research interests include cross-layer IC design for manufacturing, reliability, security, machine learning in EDA, CAD for VLSI and emerging technologies. He has published over 300 refereed papers and 8 US patents. He has served in many journal editorial boards and conference committees. He has received many awards, including SRC Technical Excellence Award, 16 Best Paper Awards, DAC Top 10 Author Award in Fifth Decade, NSF CAREER Award, IBM Faculty Award, etc.