Simultaneous Placement and Clock Tree Construction for Modern FPGAs

Wuxi Li
Univ. of Texas at Austin
wuxi.li@utexas.edu

Mehrdad E. Dehkordi
Xilinx Inc.
mehrdad@xilinx.com

Stephen Yang
Xilinx Inc.
stepheny@xilinx.com

David Z. Pan
Univ. of Texas at Austin
dpan@ece.utexas.edu

ABSTRACT

Modern field-programmable gate array (FPGA) devices often contain complex clocking architectures to achieve high-performance and flexible clock networks. The physical structure of these clock networks, however, are pre-manufactured, unadjustable, and with only limited routing resources. Most conventional FPGA placement algorithms rarely consider clock feasibility, and therefore lead to clock routing failures. Some recent works adopt simplified clock routing models (e.g., the bounding box model) to force clock legality during placement, which, however, can often overestimate clock routing demands and results in unnecessary placement quality degradation. To address these limitations, in this paper, we propose a generic FPGA placement framework that can simultaneously optimize placement quality and ensure clock feasibility by explicit clock tree construction. We demonstrate the effectiveness and efficiency of the proposed approach using the ISPD 2017 Clock-Aware Placement Contest benchmark suite. Compared with other state-of-the-art clock legalization algorithms, the proposed approach can achieve the best routed wirelength with competitive runtime.

ACM Reference Format:
Wuxi Li, Mehrdad E. Dehkordi, Stephen Yang, and David Z. Pan. 2019. Simultaneous Placement and Clock Tree Construction for Modern FPGAs. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293897

1 INTRODUCTION

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 of FPGA devices brings ever more challenges to FPGA CAD tools, especially placement engines.

Figure 1 illustrates a representative column-based FPGA architecture that has been widely adopted by many state-of-the-art commercial FPGA devices [7] (e.g., Xilinx *Virtex UltraScale* and *UltraScale+* series). In this specific architecture, each column provides one type of logic resource among *digital signal processor* (DSP), *random access memory* (RAM), I/O, and *configurable logic block* (CLB), where each CLB site further consists of multiple *lookup table* (LUT) and *flip-flop* (FF) slots. In a modern FPGA CAD flow, a design is first translated into a netlist composed of LUTs, FFs, and other heterogeneous blocks (e.g., DSPs, RAMs, and I/Os) by logic synthesis and technology mapping. Then, a placement engine determines the physical locations of all the cells in an FPGA layout shown in Fig. 1. Finally, routing is conducted to realize the interconnects among these placed cells.

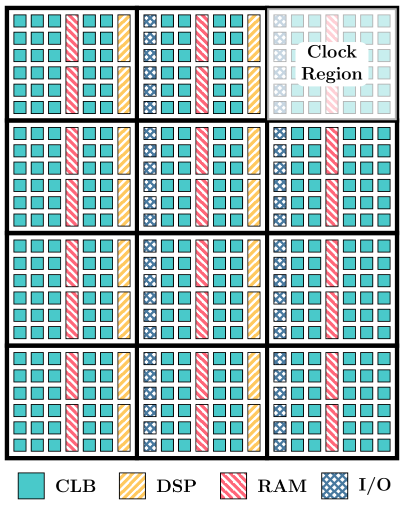

Figure 1: A representative column-based FPGA architecture adopted in state-of-the-art Xilinx UltraScale and UltraScale+ series. This device is composed of 3 $\times$ 4 clock regions.

Given the significance of FPGA placement in determining the overall implementation quality and efficiency, lots of research efforts have been previously devoted to optimizing conventional metrics like wirelength, routability, timing, and power [2, 3, 12, 13, 16, 18–20, 22–24, 26]. However, there are still limited works in the literature considering clock feasibility during placement. With the recent increase in design complexity, there can be tens to even hundreds of global clocks in a single FPGA design. Given the limited clock routing resources on today's FPGA devices, placement without careful clock network planning can easily fail the whole implementation flow.

The clocking architecture of an FPGA device, as shown in Fig. 1, typically consists of a grid of *clock regions*. One of the most important clocking constraints in such an architecture is that only a limited number of clock networks can route through each clock region. This constraint is imposed by the pre-determined number of pre-manufactured clock routing tracks in each clock region. Considering the clock loads (e.g., FFs, DSPs, and RAMs) of a clock network can often scatter over a considerable portion of the FPGA device, it is common for a clock network to span multiple clock regions. Therefore, for clock-intensive designs, clock routing congestions can be a headache, and techniques that are capable of ensuring clock feasibility during placement becomes extremely imperative.

Several previous works have tentatively explored placement techniques with the awareness of clock feasibility for FPGAs. In [10], a cost function that penalizes high-clock-usage placements is proposed

FPGA '19, February 24–26, 2019, Seaside, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293897

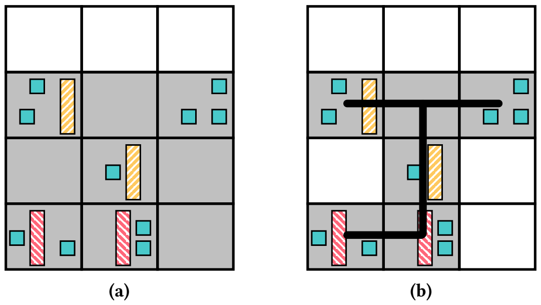

Figure 2: Illustration of the clock routing demand calculation using (a) the bounding box of clock loads (adopted in UTPlaceF 2.0 [14], NTUfplace [3], and RippleFPGA [21]) and (b) the actual clock tree. Both figures show the same clock network with the same load distribution. Shaded areas denote the 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).

and integrated into the simulated annealing-based *VPR* framework [2] to produce clock-friendly solutions. In the more recent *ISPD 2017 Clock-Aware Placement Contest* [28] hold by Xilinx, the top-3 winners UTPlaceF 2.0 [14], NTUfplace [3], and RippleFPGA [21] adopt a more realistic commercial FPGA clocking architecture (similar to Fig. 1). 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 2(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. 2(b). It shows the same clock loads distribution as that in Fig. 2(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. 2(a), which consumes 9 clock regions, the same clock network only spans 6 clock regions with tree construction in Fig. 2(b). Apart from the clock routing modeling inaccuracy, all these three works resolve clock routing congestions in a greedy and iterative manner. Specifically, they repeatedly push clock loads away from overflowed clock regions while greedily minimizing the placement disturbance in each step. Such a method, however, can only explore a very narrow solution space and always follows the decisions made previously, which can lead to very suboptimal or even infeasible solutions.

To remedy the aforementioned deficiencies in previous works, this paper presents a generic framework that simultaneously optimizes placement and ensures clock feasibility by explicit clock tree construction. Inspired by the branch-and-bound idea [11], 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. Besides, a Lagrangian relaxation [4]-based clock tree construction technique is also proposed to accurately reflect actual clock routing demands. The major contributions of this paper are highlighted as follows:

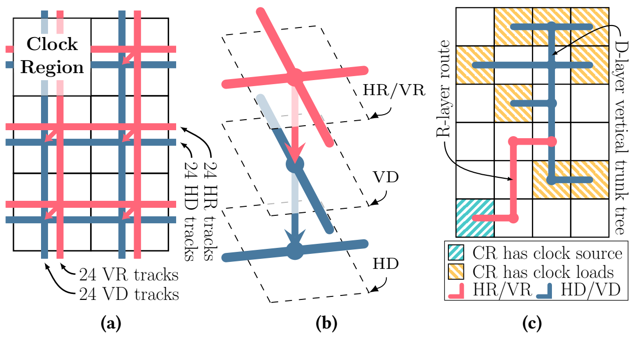

Figure 3: Illustration of the targeting clocking architecture. (a) A global view of 2 × 3 clock regions with R-layer (red) and D-layer (blue). (b) A detailed view of HR/VR/HD/VD within a single clock region. (c) The required routing pattern of a clock net.

(1) 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.
(2) We propose a novel Lagrangian relaxation-based clock tree construction technique to accurately model the clock routing demands during FPGA placement.
(3) 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.
(4) We perform experiments on the *ISPD 2017 Clock-Aware Placement Contest* [28] 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 paper is organized as follows. Section 2 reviews the targeting FPGA clocking architecture and gives the problem definition. Section 3 overviews our overall flow and details the proposed algorithms. Section 4 shows the experimental results, followed by the conclusion and future work in Section 5.

2 PRELIMINARIES

2.1 Clocking Architecture

Our targeting FPGA device is Xilinx *UltraScale VU095*, which was also adopted in both ISPD 2016 and ISPD 2017 FPGA placement contests [27, 28]. Its clocking architecture is illustrated in Fig. 3(a) – (c). The global clocking architecture, as shown in Fig. 3(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(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(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 [5].

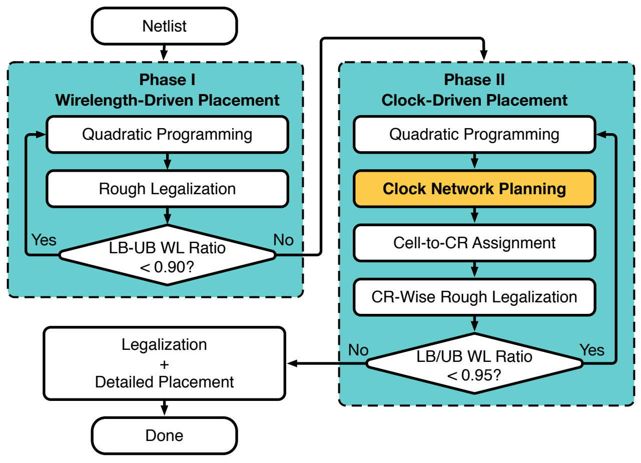

Figure 4: The proposed overall flow.

2.2 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 2.1) and the optimization objective, we now define our *simultaneous FPGA placement and clock tree construction* problem as follows.

Problem 1 (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 PROPOSED ALGORITHMS

3.1 Overview of the Proposed Flow

The proposed overall flow is shown in Fig. 4. Our framework is built on top of a state-of-the-art academic FPGA placer presented in [15], 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* [8]. In each iteration of this phase, a quadratic program is solved to minimize the wirelength, and the rough legalization [8] technique is conducted to eliminate cell overlapping. This loop is repeated until the lower-bound wirelength and upper-bound wirelength ratio [8] (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). 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 [8] of cells to point to their feasible locations found in the current placement iteration and pull them to form a more clock-feasible solution in the next iteration. The clock-driven placement phase stops when the wirelength fully converges. Finally, legalization and detailed placement are performed to further optimize the placement result while honoring the previously achieved clock routing.

As the centerpiece of the proposed flow (Fig. 4), 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 3.2. Then, in Section 3.3, 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 – 3.8.

3.2 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 2 (Clock Network Planning). *Given an optimized FPGA placement, find a movement-minimized cell-to-clock region assignment without logic resource overflow and a corresponding clock routing solution satisfying the targeting clocking architecture.*

Table 1: Notations Used in Clock Network Planning

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

Given the notations defined in Table 1, Problem 2 can be written as a binary optimization problem shown in Formulation (1). It is optimized over binary variables $\boldsymbol{x}_{v,r}$ to minimize the objective (1a) 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 (1c) guarantees that each cell is assigned to exactly one clock region. Constraint (1d) ensures that the total demand of each resource type is no more than the corresponding capacity in each clock region. Constraint (1e) 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.

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

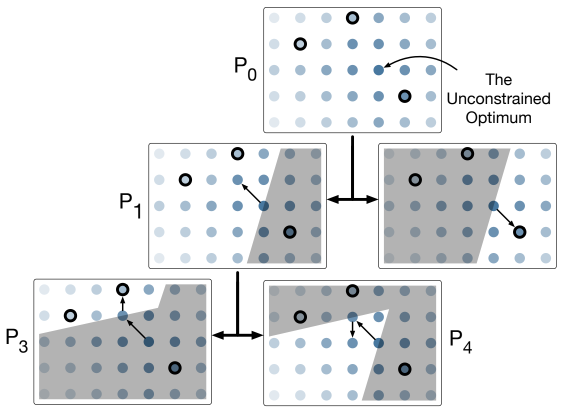

Figure 5: Illustration of the branch-and-bound method. Each circle denotes a solution and color intensity indicates its optimality. Feasible solutions are denoted by stroked circles.

3.3 Branch-and-Bound Method

A general algorithm to solve binary optimization problems, like Formulation (1), is call *branch-and-bound method* [11]. 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. 5. 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. 5 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.

3.4 The Clock Network Planning Algorithm

Algorithm 1 gives our proposed clock network planning algorithm to solve Formulation (1). Besides the inputs/outputs and notations described in Problem 2 and Table 1, 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.

Algorithm 1: Clock Network Planning

```
Input  : A placement with notations defined in Table 1. The
         maximum number of legal solutions N.
Output : A movement-minimized cell-to-clock region
         assignment without logic resource overflow, and a
         corresponding legal clock routing solution.
1  $(\text{cost}^*, \boldsymbol{x}^*, \gamma^*) \leftarrow (+\infty, \text{none}, \text{none})$;
2  $\mathsf{n} \leftarrow 0$;
3  $\kappa^{(0)}_{e,r} \leftarrow 1, \forall e \in \mathcal{E}, \forall r \in \mathcal{R}$;
4  stack.push($\kappa^{(0)}$);
5  while stack is not empty and $\mathsf{n} < \mathsf{N}$ do
6      $\kappa \leftarrow$ stack.pop();
7      Get the optimal cell-to-clock region assignment $\boldsymbol{x}^{(\kappa)}$ and
         its cost $\text{cost}^{(\kappa)}$ under constraint $\kappa$ (Section 3.5);
8      if no feasible $\boldsymbol{x}^{(\kappa)}$ exists then continue;
9      Get the clock routing solution $\gamma^{(\kappa)}$ corresponding to $\boldsymbol{x}^{(\kappa)}$
         (Section 3.6);
10     if $\gamma^{(\kappa)}$ is overflow-free then
11         $\mathsf{n} \leftarrow \mathsf{n} + 1$;
12         if $\text{cost}^{(\kappa)} < \text{cost}^*$ then
13             $(\text{cost}^*, \boldsymbol{x}^*, \gamma^*) \leftarrow (\text{cost}^{(\kappa)}, \boldsymbol{x}^{(\kappa)}, \gamma^{(\kappa)})$;
14         end
15     end
16     else if $\gamma^{(\kappa)}$ has routing overflow then
17         Derive a set of more strict constraints $K'$ from $\kappa$
             (Section 3.7);
18         Remove $\kappa' \in K'$ that has lower-bound cost larger than
             $\text{cost}^*$ (Section 3.8);
19         Push $\kappa' \in K'$ into stack by their lower-bound costs
             from high to low;
20     end
21 end
22 return $(\text{cost}^*, \boldsymbol{x}^*, \gamma^*)$;
```

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, 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 3.3), 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. 5. 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 (1) without considering the clock constraint (1e). 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, $(\text{cost}^{(\kappa)}, \boldsymbol{x}^{(\kappa)}, \gamma^{(\kappa)})$ then forms a feasible solution, and we will update the best solution $(\text{cost}^*, \boldsymbol{x}^*, \gamma^*)$ 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 1 will be further elaborated in the later sections. Section 3.5 describes the cell-to-clock region assignment in line 7. Section 3.6 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 3.7 and Section 3.8, respectively.

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

The cell-to-clock region assignment (line 7 in Algorithm 1) essentially is solving the clock-unconstrained version of Formulation (1) within the sub-space of a given clock-assignment constraint κ. It can be written as a binary optimization problem shown in Formulation (2), 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 1. Note that Formulation (1) and Formulation (2) only differ by the clock constraints (1e) and (2e).

$$
\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \sum_{v\in\mathcal{V}}\sum_{r\in\mathcal{R}} D_{v,r}\cdot \boldsymbol{x}_{v,r}, && \text{(2a)}\\
\text{subject to} \quad & \boldsymbol{x}_{v,r} \in \{0,1\}, \forall v\in\mathcal{V}, \forall r\in\mathcal{R}, && \text{(2b)}\\
& \sum_{r\in\mathcal{R}} \boldsymbol{x}_{v,r} = 1, \forall v \in \mathcal{V}, && \text{(2c)}\\
& \sum_{v\in\mathcal{V}} A_v^{(s)}\cdot \boldsymbol{x}_{v,r} \le C_r^{(s)}, \forall r\in\mathcal{R}, \forall s \in \S, && \text{(2d)}\\
& \boldsymbol{x}_{v,r} = 0, \forall (v,r) \in \{v\in\mathcal{V}, r\in\mathcal{R} \mid \\
& \qquad \exists e \in \mathcal{E}(v) \text{ s.t. } \kappa_{e,r} = 0\}. && \text{(2e)}
\end{aligned}
$$

The motivation of formulating Formulation (2) 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 [1]. Figure 6 gives a graph representation of the minimum-cost flow corresponding to Formulation (2) 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. 6, 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 (2).

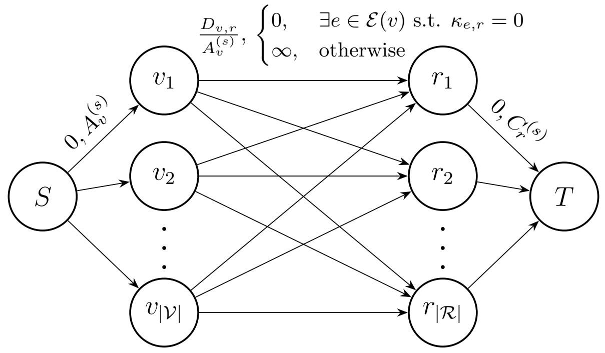
$\frac{D_{v,r}}{A_v^{(s)}}, \begin{cases} 0, & \exists e \in \mathcal{E}(v) \text{ s.t. } \kappa_{e,r} = 0 \\ \infty, & \text{otherwise} \end{cases}$
$0, A_v^{(s)}$
$0, C_r^{(s)}$
S
v_1
v_2
$v_{|\mathcal{V}|}$
r_1
r_2
$r_{|\mathcal{R}|}$
T

Figure 6: A graph representation of the minimum-cost flow for Formulation (2) 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 (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 (2). 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 1.

3.6 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 2.1, 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

[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.

relies on the D-layer trunk location, we perform D-layer routing first, then followed by R-layer routing.

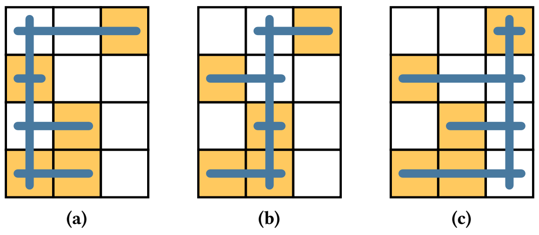

Figure 7: Three different D-layer clock tree topologies of the same clock load distribution on a 3 × 4 clock region grid. Each of them is a vertical trunk tree with horizontal branches connecting all the clock loads. Yellow shaded regions denote clock regions containing clock loads of the given clock net.

3.6.1 Lagrangian Relaxation-Based D-Layer Clock Tree Construction. As shown in Fig. 7, given a cell-to-clock region assignment on a clock region grid with m columns ($m = 3$ in Fig. 7), 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. 7) 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).

$$\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \sum_{t \in \mathcal{T}} \phi_t \cdot z_t, && \text{(3a)}\\
\text{subject to} \quad & z_t \in \{0,1\}, \forall t \in \mathcal{T}, && \text{(3b)}\\
& \sum_{t \in \mathcal{T}(e)} z_t = 1, \forall e \in \mathcal{E}, && \text{(3c)}\\
& \sum_{t \in \mathcal{T}} H_{t,r} \cdot z_t \le 24, \forall r \in \mathcal{R}, && \text{(3d)}\\
& \sum_{t \in \mathcal{T}} V_{t,r} \cdot z_t \le 24, \forall r \in \mathcal{R}. && \text{(3e)}
\end{aligned}$$

Formulation (3) is optimized over binary variables z_t to minimize the objective of topology cost (3a). If the clock tree candidate t is selected in the routing solution, then $z_t = 1$, otherwise, $z_t = 0$. Constraint (3c) ensures that exactly one candidate is selected for each clock net. Constraints (3d) and (3e) 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 2.1). 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) can be optimally solved using integer linear programming techniques, they are too computationally expensive and unaffordable in our application. Therefore, we relax Formulation (3) to a much easier problem, as shown in Formulation (4). Here, we remove the two clock resource constraints (3d) and (3e), and add a set of Lagrangian multipliers [4] λ_t in the objective (4a). 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 (4), overflow-free or overflow-minimized clock routing solutions can be achieved.

$$\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \sum_{t \in \mathcal{T}} (\phi_t + \lambda_t) \cdot z_t, && \text{(4a)}\\
\text{subject to} \quad & z_t \in \{0,1\}, \forall t \in \mathcal{T}, && \text{(4b)}\\
& \sum_{t \in \mathcal{T}(e)} z_t = 1, \forall e \in \mathcal{E}. && \text{(4c)}
\end{aligned}$$

Algorithm 2 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 $\boldsymbol{z}^{(i)}$ of Formulation (4) with $\lambda^{(i)}$ (line 5). Then, $\lambda^{(i+1)}$ can be derived from $\lambda^{(i)}$ by penalizing clock tree candidates that run through overflowed clock regions in the routing solution given by $\boldsymbol{z}^{(i)}$ (line 6). This iteration is repeated until one of the following holds: (1) a routing overflow-free solution is found; (2) λ does not change anymore; (3) the maximum iteration count $\mathrm{I}_{\max}$ is reached; Finally, in line 9 – 10, we backtrace all the explored solutions and return the one with the minimum routing overflow as the final result.

The optimal solution of Formulation (4), as given in function `solveLR` (line 11 – 18), 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 19 – 37) presents our λ updating scheme. In line 20 – 26, we first calculate the base penalty $\Delta\lambda_t$ for each candidate t. As shown in line 24 – 25, 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 27 – 33, we calculate the minimum scaling factor α that can change the optimal solution of Formulation (4) with $\alpha \cdot \Delta\lambda$ being added to current λ. If such an α does not exist, λ are kept unchanged (line 34). Otherwise, we add the extra penalty $(1 + \delta) \cdot \alpha \cdot \Delta\lambda$ to $\lambda^{(i)}$ ($\delta \ll 1$ is for tie-breaking) and return the result as $\lambda^{(i+1)}$ (line 35 – 36).

3.6.2 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 2.1) instead of a single terminal. Therefore, we extend the conventional A* search [6]-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 [17] is also applied to iteratively resolve routing overflows.

3.7 Clock-Assignment Constraint Derivation

Recall that, in line 17 of Algorithm 1, 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

Algorithm 2: LAGRANGIAN RELAXATION-BASED D-LAYER CLOCK TREE CONSTRUCTION

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

```
1  Create all clock tree candidates $\mathcal{T}$ for $\boldsymbol{x}$ (Fig. 7);
2  $\lambda_t^{(0)} \leftarrow 0, \forall t \in \mathcal{T}$;
3  $\mathsf{i} \leftarrow 0$;
4  do
5      $z^{(\mathsf{i})} \leftarrow$ solveLR($\lambda^{(\mathsf{i})}$) // solve Formulation (4)
6      $\lambda^{(\mathsf{i}+1)} \leftarrow$ updateLR($z^{(\mathsf{i})}, \lambda^{(\mathsf{i})}$) // update $\lambda$
7      $\mathsf{i} \leftarrow \mathsf{i} + 1$;
8  while $z^{(\mathsf{i})}$ has overflow and $\lambda^{(\mathsf{i})} \neq \lambda^{(\mathsf{i}-1)}$ and $\mathsf{i} < \mathsf{I}_{\max}$;
9  $z^* \leftarrow$ the $z^{(\mathsf{i})}$ with the minimum overflow;
10 return $\{t \in \mathcal{T} \mid z_t^* = 1\}$;

11 Function solveLR($\lambda$):
12     $z_t \leftarrow 0, \forall t \in \mathcal{T}$;
13     foreach $e \in \mathcal{E}$ do
14         $t^* \leftarrow$ the $t \in \mathcal{T}(e)$ with the minimum $\phi_t + \lambda_t$;
15         $z_{t^*} \leftarrow 1$;
16     end
17     return $z$;
18 end

19 Function updateLR($z^{(\mathsf{i})}, \lambda^{(\mathsf{i})}$):
20     $\Delta\lambda_t \leftarrow 0, \forall t \in \mathcal{T}$;
21     foreach $r \in \mathcal{R}$ with HD/VD overflows of $O_H/O_V$ do
22         $\mathcal{T}_H(r) \leftarrow \{t \in \mathcal{T} \mid H_{t,r} = 1\}$;
23         $\mathcal{T}_V(r) \leftarrow \{t \in \mathcal{T} \mid V_{t,r} = 1\}$;
24         foreach $t \in \mathcal{T}_H(r)$ do $\Delta\lambda_t \leftarrow \Delta\lambda_t + \frac{O_H}{|\mathcal{T}_H(r)|}$;
25         foreach $t \in \mathcal{T}_V(r)$ do $\Delta\lambda_t \leftarrow \Delta\lambda_t + \frac{O_V}{|\mathcal{T}_V(r)|}$;
26     end
27     $\alpha \leftarrow \infty$;
28     foreach $e \in \mathcal{E}$ do
29         $t^* \leftarrow$ the $t \in \mathcal{T}(e)$ being selected in iteration $\mathsf{i}$;
30         foreach $t \in \mathcal{T}(e)$ that has $\Delta\lambda_t < \Delta\lambda_{t^*}$ do
31             $\alpha \leftarrow$ min($\alpha$, $\frac{(\phi_t+\lambda_t)-(\phi_{t^*}+\lambda_{t^*})}{\Delta\lambda_{t^*}-\Delta\lambda_t}$);
32         end
33     end
34     if $\alpha = \infty$ then return $\lambda^{(\mathsf{i})}$;
35     $\lambda_t^{(\mathsf{i}+1)} \leftarrow \lambda_t^{(\mathsf{i})} + (1+\delta) \cdot \alpha \cdot \Delta\lambda_t, \forall t \in \mathcal{T}$;
36     return $\lambda^{(\mathsf{i}+1)}$;
37 end
```

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.

3.7.1 Constraint Derivation for VD Overflows. Algorithm 3 summarizes our constraint deviation scheme for resolving VD overflows.

Algorithm 3: 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  $r \leftarrow$ the clock region with the most VD overflow in $\gamma$;
2  $K' \leftarrow \varnothing$;
3  foreach $e \in \mathcal{E}$ that occupies VD resource in $r$ do
4      foreach blockage $B$ in Fig. 8 do
5          $\kappa' \leftarrow \kappa$;
6          $\kappa'_{e,b} \leftarrow 0, \forall b \in B$;
7          $K' \leftarrow K' \cup \kappa'$;
8      end
9  end
10 return $K'$;
```

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 8, 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.

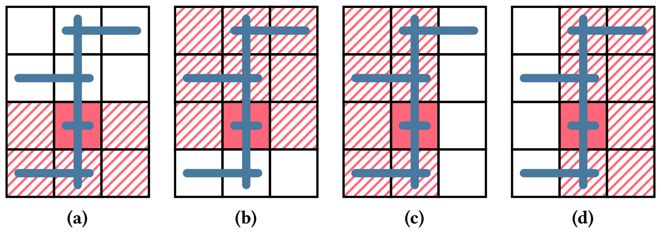

Figure 8: Four half-plane-based clock-assignment blockages (hatched/solid red regions) that are in the (a) south, (b) north, (c) west, and (d) east of a VD-overflowed clock region (solid red region).

3.7.2 Constraint Derivation for HD Overflows. Our constraint derivation for HD overflow is similar to that for VD, as described in Section 3.7.1 and Algorithm 3. However, given the fact that HD branches affect the tree topology much more locally than VD trunks, blockages of granularities finer than Fig. 8 might be able to achieve even better results. For example, the corner-based (Fig. 9) and the row-based (Fig. 10) blockages can potentially resolve the overflow with less cell movement compared with the blockages shown in Fig. 8. Surprisingly, as will be shown in Section 4.3, the blockage schemes in Fig. 9 and Fig. 10 cannot outperform that in Fig. 8 within a limited amount of time in our experiments. This might be because the blockages in Fig. 9 and Fig. 10 tend to cut the placeable region of each clock into non-convex and unconnected fragments, which significantly slow down the convergence of Algorithm 1. While using the blockages in Fig. 8, 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.

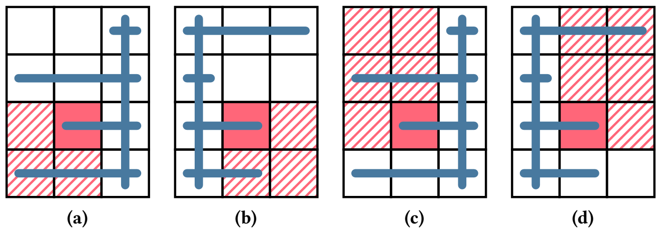

Figure 9: Corner-based clock-assignment blockages for an HD-overflowed clock region.

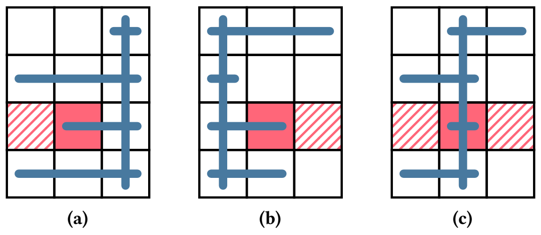

Figure 10: Row-based clock-assignment blockages for an HD-overflowed clock region.

3.8 Lower-Bound Cost Calculation

In this section, we introduce a method to calculate the lower-bound cost of Formulation (2) for a given clock-assignment constraint. This lower-bound cost is essential for: (1) the solution pruning in line 18 of Algorithm 1, and (2) the constraint sorting in line 19 of Algorithm 1. 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 (2)). Given these requirements, we propose the following lower-bound cost for a given clock-assignment constraint κ:

$$\mathtt{cost}_{\mathtt{LB}}(\kappa) = \sum_{v \in \mathcal{V}} \min_{\{r \in \mathcal{R} | \kappa_{e,r}=1, \forall e \in \mathcal{E}(v)\}} D_{v,r}, \tag{5}$$

where $\mathcal{E}(v)$ denotes the set of clock nets incident to cell v. Equation (5) can be proved as a lower-bound cost of Formulation (2), since it is the optimal solution of Formulation (2) without considering the logic resource constraint (2d). Moreover, its time complexity is only linear to the number of cells in the design, which is very computation-efficient.

4 EXPERIMENTAL RESULTS

We implemented the proposed techniques in C++ based on the placement framework in [15] 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 [28] released by Xilinx is used to demonstrate the effectiveness of the proposed approach. Routed wirelength reported by Xilinx Vivado v2016.4 [25] is used to evaluate the placement quality.

Table 2 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 2: 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	-

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

Table 3 compares our results with other state-of-the-art academic clock-aware placers, including UTPlaceF 2.0 [14], NTUfplace [9], RippleFPGA [21], and [15]. 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 [15] and our placer are executed with 16 and 10 threads, respectively. The runtime result of NTUfplace in Table 3 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 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.

4.2 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 [14] (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 [14]-Impl. Since [14]-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 4 compares the proposed approach with [14]-Impl. It can be seen that the proposed approach achieves similar results compared with [14]-Impl under relatively loose clock constraints (CC = 24/12). As the clock capacity reduces, the proposed approach starts to outperform [14]-Impl in both routed wirelength and runtime. On average, with CC = 8/7/6/5, [14]-Impl suffers 1.1%/2.3%/13.1%/30.4% wirelength degradation, while runs $\times$1.21/$\times$1.40/$\times$1.67/$\times$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 [14]-Impl. Therefore, our approach is especially effective for cases with high clock utilization.

4.3 Comparison of Different Clock-Assignment Blockage Schemes

Table 5 compares the three clock-assignment blockage schemes described in Section 3.7: (1) half-plane-based blockages (Fig. 8), (2) corner-based blockages (Fig. 9), and (3) row-based blockages (Fig. 10). 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. 8), but preserving the continuity and convexity of feasible regions should play a very important role here.

4.4 Branch-and-Bound Tree Exploration

Figure 11 visualizes the lower-bound costs (Eq. (5)) and the actual costs (Formulation (2)) 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. 11. 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.

5 CONCLUSION AND FUTURE WORK

In this paper, 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

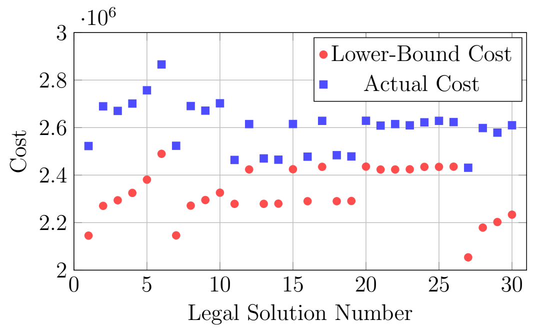

Figure 11: The lower-bound and actual costs of the first 30 feasible solutions found in a branch-and-bound procedure of *CLK-FPGA01* with CC = 6.

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 proposed approach outperforms other state-of-the-art approaches in routed wirelength with competitive runtime. In the future, we plan to further explore different branch-and-bound strategies and parallelize the tree exploration process.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported in part by Xilinx Inc.

REFERENCES

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

[2] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *FPL*. 213–222.

[3] Yu-Chen Chen, Sheng-Yen Chen, and Yao-Wen Chang. 2014. Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs. In *ICCAD*. 647–654.

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

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

[6] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[7] Xilinx Inc. 2018. http://www.xilinx.com.

[8] Myung-Chul Kim, Dong-Jin Lee, and Igor L. Markov. 2012. SimPL: An Effective Placement Algorithm. *IEEE TCAD* 31, 1 (2012), 50–60.

[9] Yun-Chih Kuo, Chau-Chin Huang, Shih-Chun Chen, Chun-Han Chiang, Yao-Wen Chang, and Sy-Yen Kuo. 2017. Clock-aware placement for large-scale heterogeneous FPGAs. In *ICCAD*. 519–526.

[10] Julien Lamoureux and Steven J. E. Wilton. 2008. On the Trade-off between power and flexibility of FPGA clock networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 1, 3 (2008), 13:1–13:33.

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

[12] Wuxi Li, Shounak Dhar, and David Z. Pan. 2018. UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing. *IEEE TCAD* 37, 4 (2018), 869–882.

[13] Wuxi Li, Meng Li, Jiajun Wang, and David Z. Pan. 2017. UTPlaceF 3.0: A parallelization framework for modern FPGA global placement. In *ICCAD*. 908–914.

[14] Wuxi Li, Yibo Lin, Meng Li, Shounak Dhar, and David Z Pan. 2018. UTPlaceF 2.0: A high-performance clock-aware FPGA placement engine. *ACM TODAES* 23, 4 (2018), 42.

[15] Wuxi Li and David Z. Pan. 2018. A new paradigm for FPGA placement without explicit packing. *IEEE TCAD* (2018).

[16] Tzu-Hen Lin, Pritha Banerjee, and Yao-Wen Chang. 2013. An efficient and effective analytical placer for FPGAs. In *DAC*. 10:1–10:6.

[17] Wen-Hao Liu, Yih-Lang Li, and Cheng-Kok Koh. 2012. A fast maze-free routing congestion estimator with hybrid unilateral monotonic routing. In *ICCAD*. 713–719.

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

Designs	UTPlaceF 2.0 [14]				NTUfplace [9]				RippleFPGA [21]				[15]				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

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

Table 4: Normalized Wirelength and Runtime Comparison with [14]-Impl Under Different Clock Capacities (CC)

Designs	CC = 24				CC = 12				CC = 8				CC = 7				CC = 6				CC = 5			
	[14]-Impl		Proposed		[14]-Impl		Proposed		[14]-Impl		Proposed		[14]-Impl		Proposed		[14]-Impl		Proposed		[14]-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.

Table 5: 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.

[18] Alexander S. Marquardt, Vaughn Betz, and Jonathan Rose. 1999. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *FPGA*. 37–46.

[19] Ryan Pattison, Ziad Abuowaimer, Shawki Areibi, Gary Gréwal, and Anthony Vannelli. 2016. GPlace: A congestion-aware placement tool for ultrascale FPGAs. In *ICCAD*. 68:1–68:7.

[20] Chak-Wa Pui, Gengjie Chen, Wing-Kai Chow, Ka-Chun Lam, Jian Kuang, Peishan Tu, Hang Zhang, Evangeline F.Y. Young, and Bei Yu. 2016. RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs. In *ICCAD*. 67:1–67:8.

[21] Chak-Wa Pui, Gengjie Chen, Yuzhe Ma, Evangeline F. Y. Young, and Bei Yu. 2017. Clock-aware ultraScale FPGA placement with machine learning routability prediction. In *ICCAD*. 915–922.

[22] Senthilkumar Thoravi Rajavel and Ali Akoglu. 2011. MO-Pack: Many-objective clustering for FPGA CAD. In *DAC*. 818–823.

[23] Amit Singh, Ganapathy Parthasarathy, and Malgorzata Marek-Sadowska. 2002. Efficient circuit clustering for area and power reduction in FPGAs. *ACM TODAES* 7, 4 (2002), 643–663.

[24] Love Singhal, Mahesh A. Iyer, and Saurabh Adya. 2017. LSC: A large-scale consensus-based clustering algorithm for high-performance FPGAs. In *DAC*. 30:1–30:6.

[25] Xilinx Vivado Design Suite. 2018. https://www.xilinx.com/products/design-tools/vivado.html.

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

[27] Stephen Yang, Aman Gayasen, Chandra Mulpuri, Sainath Reddy, and Rajat Aggarwal. 2016. Routability-driven FPGA placement contest. In *ISPD*. 139–143.

[28] Stephen Yang, Chandra Mulpuri, Sainath Reddy, Meghraj Kalase, Srinivasan Dasasathyan, Mehrdad E. Dehkordi, Marvin Tom, and Rajat Aggarwal. 2017. Clock-aware FPGA placement contest. In *ISPD*. 159–164.