MrDP: Multiple-row Detailed Placement of Heterogeneous-sized Cells for Advanced Nodes *

Yibo Lin[1], Bei Yu[2], Xiaoqing Xu[1], Jhih-Rong Gao[3], Natarajan Viswanathan[3], Wen-Hao Liu[3], Zhuo Li[3], Charles J. Alpert[3], and David Z. Pan[1]

[1]ECE Department, University of Texas at Austin, Austin, TX, USA
[2]CSE Department, The Chinese University of Hong Kong, NT, Hong Kong
[3]Cadence Design Systems Inc., Austin, TX, USA

ABSTRACT

As VLSI technology shrinks to fewer tracks per standard cell, e.g., from 10-track to 7.5-track libraries (and lesser for 7nm), there has been a rapid increase in the usage of multiple-row cells like two- and three-row flip-flops, buffers, etc., for design closure. Additionally, the usage of multi-bit flip-flops or flop trays to save power creates large cells that further complicate critical design tasks, such as placement. Detailed placement happens to be a key optimization transform, which is repeatedly invoked during the design closure flow to improve design parameters, such as, wirelength, timing, and local wiring congestion. Advanced node designs, with hundreds of thousands of multiple-row cells, require a paradigm change for this critical design closure transform. The traditional approach of fixing multiple-row cells during detailed placement and only optimizing the locations of single-row standard cells can no longer obtain appreciable quality of results. It is imperative to have new techniques that can simultaneously optimize both multiple- and single-row high cell locations during detailed placement. In this paper, we propose a new density-aware detailed placer for heterogeneous-sized netlists. Our approach consists of a chain move scheme that generalizes the movement of heterogeneous-sized cells as well as a nested dynamic programming based approach for wirelength and density optimization. Experimental results demonstrate the effectiveness of these techniques in wirelength minimization and density smoothing compared with the most recent detailed placer for designs with heterogeneous-sized cells.

1. INTRODUCTION

Using single-row height standard cells has been the dominant methodology for modern VLSI digital design. For a given technology node, the height and width of standard cells are carefully selected to optimize various characteristics, such as, timing, packing, and pin accessibility. The common nomenclature for cell libraries is "N"-track, with "N" being the height of the circuit row and standard cells in terms of the number of covered routing tracks. The last few years have seen a steady decrease in "N" with each new technology node, e.g., from 10 to 7.5 (and possibly lesser for 7nm). In this scenario, it is getting increasingly difficult to design complex circuit components (flip-flops, muxes, etc.) as single-row height cells, while satisfying required performance and routing characteristics. As a result, advanced node designs are increasingly adopting the design and usage of multiple-row height cells for such complex circuit components.

Additionally, to satisfy stringent power requirements, flip-flop merging and usage of multi-bit flip-flops (MBFFs) or flop trays is becoming increasingly prevalent [1–3] in modern designs. MBFF enables the sharing of clock buffers between flip-flops, which decreases both power and area. Statistics show that a 2-bit MBFF is able to achieve around 14% power reduction and 4% area reduction per bit, while a 4-bit MBFF can achieve around 22% power reduction and 29% area saving per bit [3]. But MBFFs happen to be large, multiple-row height cells. This significantly increases the complexity for steps like legalization and detailed placement.

In addition, to meet die-size requirements for area, power, and cost reduction, design densities are approaching the limit. It is common for designs with up to 90% density, which makes detailed placement critical to resolve local wiring congestion. In an extremely dense design, it is very difficult to insert or move large cells during legalization and detailed placement without significant disruption to the local neighborhood. Furthermore, the number of interconnect pins per standard cell varies for a given cell library and often lacks correlation to the cell area. Without careful planning, local congestion can be caused by accumulation of cells with high pin count. Therefore, it is critical to make proper usage of the limited die area for optimizing both wirelength and congestion.

Placement is usually divided into three steps, global placement, legalization and detailed placement [4]. Global placement determines the rough locations of cells while minimizing objectives, such as, wirelength, routability and timing. But the solution from global placement often contains overlap and thus is not design rule friendly. Legalization removes overlaps and aligns cells to placement sites. Finally, detailed placement tries to further improve the solution by moving cells locally. Sometimes legalization is integrated into detailed placement instead of a separate step.

Global placement techniques are fairly mature in handling the mixed-sized placement problem [5–10]. But there has been little research in detailed placement for heterogeneous-sized netlists, especially where the number of multiple-row height cells ranges in the hundreds of thousands, as seen in advanced node designs. Wu et al. [11] propose a straightforward technique to handle double-row height cells during detailed placement. In their method, they use cell grouping and cell inflation to convert all the single-row height cells in the design to double-row height cells. This results in a placement problem with only double-row height cells. Consequently, a conventional placement engine can be used to opti-

*This work is supported in part by CUHK Direct Grant for Research.

ICCAD '16, November 07-10, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4466-1/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2966986.2967055

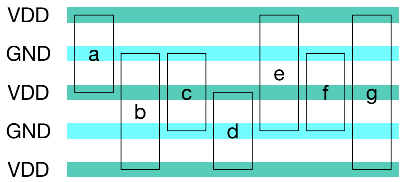

Figure 1: Example of multiple-row height cells in a layout.

mize the designs. However, this approach is unable to handle the power line alignment constraint from multiple-row height cells; e.g. cells with power rail on top and bottom have to be placed in rows with the same power line configuration. Another key drawback with this approach is its inability to handle larger cells that span three or more circuit rows. Chow et al. [12] propose the first legalization algorithm for multiple-row height standard cells with an objective of displacement minimization. They explore the insertion points in the layout and try to remove overlaps with minimum displacement.

To address the challenges in placement for advanced technology nodes, we propose a detailed placer for heterogeneous-sized netlists that addresses the traditional detailed placement objectives of wirelength, cell density and pin density [4, 5, 11, 13–16]. The major contributions are summarized as follows.

1. A chain move scheme that generalizes the movement of heterogeneous-sized cells to optimize wirelength, cell and pin density by searching for the maximum prefix sum of the improvements.
2. A nested dynamic programming based technique solving ordered double-row placement for wirelength optimization.
3. Outperform the most recent detailed placer for multiple-row height cells by 3.0% in scaled wirelength, 13.3% in cell density and 13.2% in pin density.

The rest of the paper is organized as follows. Section 2 illustrates the special constraints and problem formulation for the placement. Section 3 provides a detailed explanation of our proposed techniques. Section 4 verifies the effectiveness of our approach, followed by conclusion in Section 5.

2. PRELIMINARIES AND OVERALL FLOW

In this section, we will explain the constraints in placement for designs with heterogeneous-sized standard cells and give the problem formulation.

2.1 Power Line Alignment

Power line alignment is a special placement constraint from a multiple-row height cell. Fig. 1 illustrates an layout example of seven multiple-row height cells, where five cells take even number of rows (i.e. cells a, c, d, f and g). Cells a, d and g have power rails (VDD) on top and bottom of the cells, and ground rails (GND) in the middle. They must be placed in alternative rows with proper VDD/GND alignment, since we cannot fix the alignment through cell flipping or rotation. Similarly, cells c and f have VDD in the middle and GND on the top and bottom. The bottom of such cells must be aligned to rows with GND at the bottom. However, for cells with odd number of rows, such as cell b and e, there is no such constraint, since it has power rail on the top or bottom and ground rail on the other side. This configuration is the same as single-row height cells, so cell flipping and rotation can fix the alignment issue.

The constraint for power line alignment can be summarized as follows. An even-row height cell must align to placement rows with the same type of power line at the bottom as that in the cell, while any odd-row height cell, including single-row height cell, can align to any placement row with proper orientation.

2.2 Problem Formulation

In modern VLSI placement, the optimization usually includes multiple objectives, such as wirelength and density. Wirelength is still regarded as the major objective, while density metrics cannot be neglected, because pure wirelength-driven placement often produces congested solution that results in difficulty for post-placement stages, such as routing. Therefore, in this work we adopt the scaled wirelength metric from ICCAD 2013 placement contest [17] considering both wirelength and cell density. Half-perimeter wirelength (HPWL) is used as the wirelength metric, which is defined as follows:

$$\text{HPWL} = \sum_{n \in N} \max_{i \in n} x_i - \min_{i \in n} x_i + \max_{i \in n} y_i - \min_{i \in n} y_i, \tag{1}$$

where N denotes the set of interconnections in the circuit.

Average bin utilization (ABU) evaluates the density of a placement solution [8]. The average density of the top $\gamma\%$ bins of highest utilization is denoted by ABU_γ. The ABU penalty for density is computed from a weighted sum of overflow, which is defined in the following equations.

$$\text{overflow}_\gamma = \max\left(0, \frac{\text{ABU}_\gamma}{d_t} - 1\right), \tag{2a}$$

$$\text{ABU} = \frac{\sum_{\gamma \in \Gamma} w_\gamma \cdot \text{overflow}_\gamma}{\sum_{\gamma \in \Gamma} w_\gamma}, \Gamma \in \{2, 5, 10, 20\}, \tag{2b}$$

where d_t denotes the target utilization and w_2, w_5, w_{10}, w_{20} are set to 10, 4, 2, 1, respectively. With the definition of ABU penalty, ICCAD 2013 placement contest defines a scaled wirelength cost to generalize both wirelength and density costs, as shown in Eq. (3).

$$\text{sHPWL} = \text{HPWL} \cdot (1 + \text{ABU}). \tag{3}$$

In the ICCAD 2013 placement contest, only cell area utilization is included in the computation of ABU. In advanced technology nodes, area utilization is not enough to model the congestion, because some large cells may contain very few pins, while some small cells may in the contrast involve a lot of interconnections. So we propose average pin utilization (APU) that captures the pin distribution of the layout. The pin density in each bin is the ratio of number of pins to the number of placement sites in the bin. Once the pin density map is obtained, the computation of APU penalty is the same as that of ABU.

With all the metrics defined, the multiple-row detailed placement (MrDP) problem is defined as follows.

Problem 1 (MrDP). *Given an initial heterogeneous-sized standard cell placement plus a number of fixed macro blocks, either legal or not, we produce a legal placement solution with optimized wirelength and density, i.e. sHPWL and APU.*

2.3 Overall Flow

The overall flow of our detailed placement engine is shown in Alg. 1. Given the placement solution from global placement, we first check whether the placement is legal. If it is not legal, legalization is performed to remove overlaps and align power line of multiple-row cells. In this step, we first perform chain move algorithm (see Section 3.1) in overlap reduction mode. Because the initial placement solutions may contain many overlaps leading to large wirelength degradation in legalizer. Next the legalizer from [12] is called to ensure legality with minimum displacement. Then we perform wirelength optimization to improve both wirelength and density until less than 1% cells are moved or maximum iteration is reached. We allow at most 6 iterations in the experiment. The ordered double-row placement (see Section 3.2) is performed to further optimize wirelength before the final placement is produced.

Algorithm 1 Overall Placement Flow

```
Input: A set of placed cells $C$ in the layout.
Output: Legal placement with optimized wirelength and density.
1: if placement is not legal then
2:     Perform Chain Move in overlap reduction mode;
3:     Perform legalization if still not legal;
4: end if
5: repeat
6:     Perform Chain Move in wirelength mode;
7: until converged or maximum iteration reached
8: Perform ordered double-row placement;
```

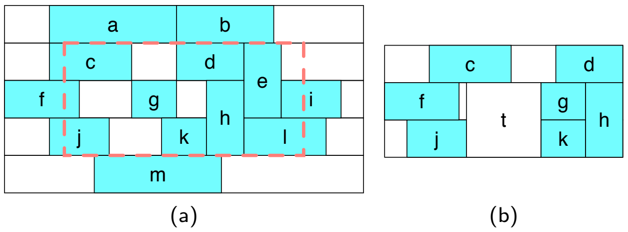

Figure 2: Example of (a) placement with multiple-row height cells (b) inserting another cell t by slightly shifting cell g and j.

3. DETAILED PLACEMENT FOR MULTIPLE-ROW CELLS

In this section, we will explain our placement algorithms such as Chain Move and Ordered Double-Row Placement in details.

3.1 Chain Move Algorithm

One of the typical detailed placement approaches is to improve wirelength in a cell-by-cell manner; i.e. pick a cell and move to better position or try to swap with another cell for better wirelength [4, 14, 15]. It is proved to be very effective in the detailed placement for single-row cells. However, the situation changes when it comes to multiple-row height cells. Since a multiple-row height cell occupies the space of contiguous rows, it is more likely to involve overlaps with multiple cells, which results in the failure of position search with previous approach. Fig. 2(a) gives an example of placement which is difficult to insert another multiple-row height cell t into the dashed region without perturbing at least two cells. With slightly shifting cells g and j, shown as Fig. 2(b), cell t can be placed in the dashed region without overlap. Similar situation may also occur to very large single-row height cells which are difficult to be fit into dense regions without perturbation of multiple cells.

If it is able to allow the movement of multiple cells at a time, there will be more candidate positions for better placement quality. Inspired by density preserving refinement from [9] and gain map from [18, 19], we develop an algorithm to allow other cells to move together when optimizing a target cell.

Definition 1 (Chain Move). *Each chain move contains a set of movements for one or several cells.*

A chain move involving multiple cells is usually triggered by the attempts of inserting a cell into a position resulting in overlaps with existing cells in that region, so the overlapped cells need to find new positions to resolve overlaps. If a cell is placed to a position without any overlap, there is only a single movement in the chain move.

Definition 2 (Cell Pool). *It is a queue structure used for temporary storage of cells within a chain move.*

In the example of Fig. 2, cell t overlaps with cells g and j when inserted to the dashed region, so cells g and j are added to the cell pool. In the following steps, cells in the cell pool are first popped out and placed until the cell pool goes empty, which indicates the end of a chain move.

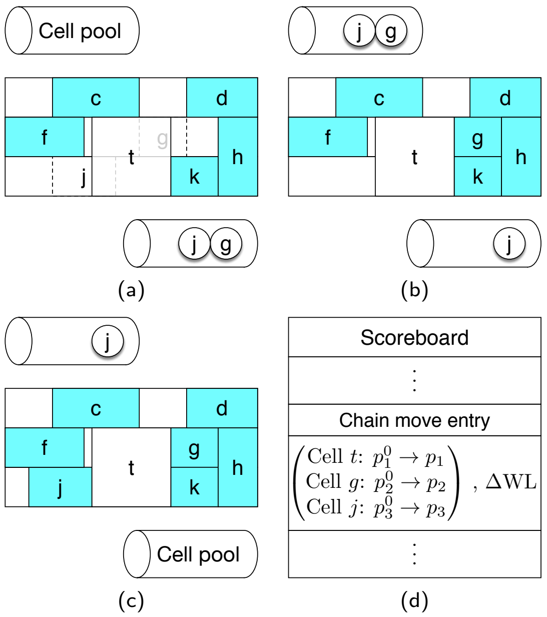

Figure 3: A chain move example of (a) 1st movement: place cell t to p_1 from p_1^0 and push overlapped cell g and j to cell pool (b) 2nd movement: pop cell g from cell pool and place to p_2 from p_2^0 (c) 3rd movement: pop cell j from cell pool and place to p_3 from p_3^0 (d) corresponding chain move entry in scoreboard.

Definition 3 (Scoreboard). *It consists of an array of chain move entries with corresponding changes in wirelength cost for each chain move.*

Since the positions of all cells are determined at the end of a chain move, we can compute accurate wirelength cost and record the differences with that at the beginning of the chain move. The scoreboard can help find a cumulatively good solution instead of that in a very greedy approach which usually requires improvements in each movement.

For the chain move example in Fig. 2, Fig. 3 gives the corresponding example of interaction between the cell pool and scoreboard. Here the horizontal cylinders on top of each Fig. 3(a) to 3(c) indicate the status of the cell pool before any movement, while the ones on the bottom indicate the status after the movements. At the beginning of the 1st movement, the cell pool is empty. Cell t is moved to position p_1 from p_1^0 but results in overlap with cells g and j during the 1st movement, so they are pushed into the cell pool. In the 2nd movement, cell g is popped from the cell pool and moved to position p_2 from p_2^0 to resolve overlap. Similarly, the 3rd movement places cell j to position p_3 from p_3^0. Fig. 3(d) shows the corresponding chain move entry in the scoreboard, which not only records each movement but also the change of wirelength cost before and after this chain move.

3.1.1 *Overview of Chain Move Algorithm*

The overview of the chain move algorithm is shown in Alg. 2 and the notations are defined in Table 1. In general each cell is only allowed to move once during one iteration. The function `ReorderCells` in line 1 of Alg. 2 shuffles the cell sequence in C. Then cell set C is copied to a first-in-first-out queue structure and the main loop of chain move algorithm begins.

Within the loop, we first try to fetch a cell from the cell pool. If the cell pool is empty, we then obtain the first cell c_i in C. Then region r_i for cell c_i is computed for search of candidate positions, which is completed by function `ComputeSearchRegion`.

For each candidate position a_j in A_i, the cost is computed by function `ComputeMoveCost` and the position with the best cost is applied to the cell from lines 15 to 28. When applying the best position, it is necessary to push all the overlapped cells in O_b to the cell pool and update the movement records in the scoreboard. If the cell pool goes to empty after a movement,

Table 1: Notations used in Chain Move Algorithm

Pool	The cell pool.
Board	The scoreboard.
p_i^0	Initial position of Cell c_i.
p_i	Candidate position of cell c_i.
O_i	The set of cells overlapping with cell c_i at p_i.
$cost_i$	The cost of cell c_i at p_i.
$p_b, O_b, cost_b$	Correspond to best $p_i, O_i, cost_i$, respectively.

Algorithm 2 Chain Move Algorithm

```
Input:  A set of placed cells $C$ in the layout.
Output: Move cells to minimize wirelength cost.
 1: ReorderCells($C$);
 2: Re-structure $C$ as a queue;
 3: while $C$ is not empty or $Pool$ is not empty do
 4:     if $Pool$ is not empty then
 5:         $c_i \leftarrow Pool$.pop();
 6:     else
 7:         $c_i \leftarrow C$.pop();
 8:         if $c_i$ has already been moved then
 9:             Continue;
10:         end if
11:     end if
12:     $r_i \leftarrow$ ComputeSearchRegion($c_i$);
13:     $A_i \leftarrow$ collect candidate positions in $r_i$;
14:     $cost_b \leftarrow \infty$;
15:     for each $a_j \in A_i$ do
16:         $(cost_i, p_i, O_i) \leftarrow$ ComputeMoveCost($c_i$, $a_j$);
17:         if $cost_i < cost_b$ then
18:             $cost_b \leftarrow cost_i$; $p_b \leftarrow p_i$; $O_b \leftarrow O_i$;
19:         end if
20:     end for
21:     Move $c_i$ to $p_b$;
22:     $Pool$.push($O_b$);
23:     $Board$.last.append($c_i, p_i^0 \rightarrow p_b$);
24:     if $Pool$ is empty then
25:         Compute $\Delta$WL for $Board$.last;
26:     end if
27: end while
28: BacktraceToBestEntry($C$, $Board$);
```

which means the end of the chain move, we can now compute the accurate wirelength change and update the scoreboard. At the end of each pass, function `BacktraceToBestEntry` scans the scoreboard to find the best cumulative wirelength.

3.1.2 *Max Prefix Sum of Wirelength Improvement*

Like that in the well-known KL and FM partitioning algorithm [18,19], we have a scoreboard that records the wirelength changes in each chain move, which helps find the maximum prefix sum of wirelength improvement by `BacktraceToBestEntry`. So the chain move scheme allows temporary degradation of wirelength as long as it eventually achieves better solutions, which can help find the best cumulative wirelength.

3.1.3 *Constraints to Chain Move*

There exist corner cases where a cell may fail to find any legal position in its search region. The corner case is likely to be triggered when all cells in a dense region have already been moved in this pass, because each cell is only allowed to move once in each pass. If such corner cases are triggered, we discard current chain and recover all the movements in this chain. Another corner case lies in the involvement of too many cells in a chain, which may result in the difficulty in searching for legal positions for the last cell. Therefore, we set an upper bound to the length of a chain to avoid long chains. Any chain exceeding the upper bound will trigger the discarding process. The maximum length of chain is set to 10000, but it is never triggered in the experiment.

Lemma 1. *If the placement is legal at the beginning of a chain move, the legality is maintained at the end of the chain move.*

Proof. If the chain is discarded, all movements are recovered, so there is no perturbation to the placement. Otherwise, the chain ends because the cell pool goes empty, which means the last movement does not cause any overlap. So the placement is still legal at the end of the chain move. The maintenance of legality is very meaningful to avoid wirelength degradation from extra legalization effort. □

3.1.4 *Visiting Order of Cells*

The visiting order of cells during each pass matters to the solution quality. If we keep a fixed order for each iteration, the wirelength saturates quickly and fails to descent further. So a suitable visiting order is essential to the solution quality under different objectives. Here we discuss the details about the function `ReorderCells` for different optimization objectives. In overlap reduction mode, multiple-row height cells and large cells have higher priority, because it is easier for small cells to find overlap-free positions and thus a legal placement can be found more efficiently. When it comes to wirelength minimization from a legal placement, those cells far away from their optimal regions are granted with high priority, because higher gain can be achieved by moving cells with longer distances.

3.1.5 *Search Region Computation*

We discuss the function `ComputeSearchRegion` here on search region computation. First we compute the optimal region as most previous global move algorithms do [14], but it is often congested. We extend the optimal region by mirroring the original position of the cell to the center of the optimal region and form a new box. Any bin intersecting with the search region will be considered for collection of candidate positions to the set A_i. We check bins from the ones close to the optimal region to farther ones. We observe that after several updates in line 18 to 20 for each cell, the final solution quality converges. To save runtime we exit early from the loop after trying several positions for each cell in the experiment.

3.1.6 *Move Cost Computation*

Now we explain the function `ComputeMoveCost`. The objective of the placement includes wirelength and density. In addition, each movement may lead to overlapping cells that will be collected to the cell pool. So the cost consists of three parts: wirelength cost, density cost, and overlap cost, shown as follows,

$$cost = \Delta\text{WL} \cdot (1 + \alpha \cdot c_d) + \beta \cdot c_{ov}, \tag{4}$$

where ΔWL denotes the wirelength cost, c_d denotes density cost and c_{ov} denotes the overlap cost. The weights α and β are set to 1.5 and 0.5 in the experiment.

Wirelength cost is in general defined as the HPWL change for the movement. However, if the cell is connected to some cells in the cell pool whose positions are not determined yet, such connections are ignored.

In the density cost, we consider both area density and pin density. In the placement that involves multiple-row height cells, the cells can be very large and result in the intersections with multiple bins. So the density increases in all bins are summed up for cost. Let c_{ad} denote the cost of area density and c_{pd} denote the cost of pin density. Let B be the set of bins intersected with the cell c_i at candidate position p_i and $d_a(b)$ and $d_p(b)$ denote the original area and pin density for bin b.

$$c_{ad} = \sum_{b \in B} \sum_{\gamma \in \Gamma} w_\gamma \cdot f(d_a, \Delta d_a, \mathrm{ABU}_\gamma), \tag{5a}$$

$$c_{pd} = \sum_{b \in B} \sum_{\gamma \in \Gamma} w_\gamma \cdot f(d_p, \Delta d_p, \mathrm{APU}_\gamma), \tag{5b}$$

$$c_d = 0.5 \times (\frac{c_{ad}}{d_t^a} + \frac{c_{pd}}{d_t^p}), \tag{5c}$$

$$f(d, \Delta d, \overline{d}) = \begin{cases} \frac{\Delta d}{\overline{d}}, & \text{if } d + \Delta d \geq \overline{d}, \\ 0, & \text{otherwise}, \end{cases} \tag{5d}$$

where Δd_a and Δd_p denote the area and pin density increase in each bin, d_t^a and d_t^p denote the target area and pin density for the layout, respectively. Function f computes the density cost and the cost only happens when the new density exceeds the average density of the top γ% bins. Although the weights for c_{ad} and c_{pd} can be adjusted for different targets, we set them equal in the experiment for simplicity.

The overlap cost c_{ov} is defined as the total area of overlapped cells times the total number of pins divided by row height. As the overlapped cells need to be inserted to the cell pool which results in the inaccuracy of wirelength cost computation, fewer pins are preferred for less contribution to the wirelength cost.

There are some hard constraints for a candidate position that lead to invalidate this candidate. Each overlapped cell must be no larger than current cell; otherwise, it is even more difficult to find legal positions for those overlapped cells. The overlapped cells must not be moved yet in current pass, because each cell can only move once within each pass of iteration.

3.1.7 *Various Optimization Modes*

The Chain Move algorithm can be configured to either overlap reduction or wirelength minimization. The main difference lies in the function `BacktraceToBestEntry` which will not be called in overlap reduction mode, because we observe that applying all the chain moves removes most of the overlaps regardless of potential wirelength degradation. Empirically we often still get some wirelength improvements. In this mode, there is an additional part of displacement cost added to Eq. (4). The purpose of the displacement cost is to reduce the perturbation to the global placement solution.

In wirelength minimization mode, we also perform local clustering of horizontally abutting cells in every odd iteration if we detect they become the bottlenecks of wirelength reduction. After chain move iterations, we fix multiple-row height cells and perform conventional global move to single-row height cells for further wirelength improvements. This incremental step usually converges at 1 or 2 iterations.

3.2 Ordered Double-Row Placement

The ordered single-row placement has been well explored in detailed placement for wirelength minimization and legalization [13, 16, 20–23]. There are also many single-row algorithms designed for manufacturability compliance, such as multiple patterning lithography, FinFET process and E-beam lithography [24–30]. The problem can be formulated into a dual min-cost flow problem that can be solved in $\mathcal{O}(n^2 \log m^2)$ time complexity for wirelength minimization [20], where n is the number of cells in a row and m is the number of nets involved. The runtime is reduced to $\mathcal{O}(m \log m)$ by the clumping algorithm from [21]. If each cell in a row has a maximum displacement M, the problem can be transferred to a shortest path problem and a dynamic programming (DP) based algorithm is able to solve the problem in $\mathcal{O}(M^2 n)$ [16, 26]. It can be further improved to $\mathcal{O}(Mn)$ by exploiting the monotonicity and pruning the solution space [30]. However, most of these algorithms only focus on single-row placement and are not able to deal with multiple-row height cells. Here

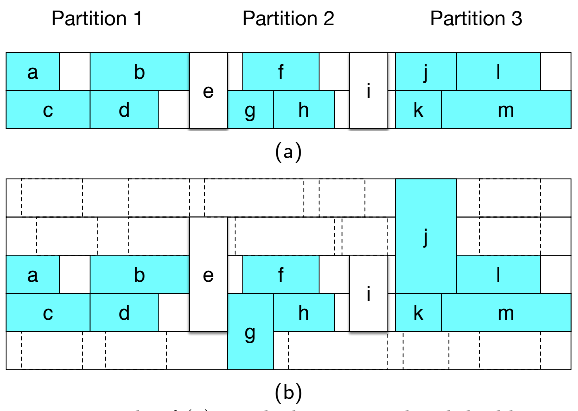

Figure 4: Example of (a) an ideal case in ordered double-row placement (b) a general case with large splitting cells and crossing cells such as cells e and j.

we define an ordered double-row placement problem as follows.

Problem 2 (Ordered Double-Row Placement). *Given two rows of cells that are ordered from left to right in each row, horizontally move the cells to optimize HPWL without ruining the order of cells in each row.*

Please note that the two sequences of cells may contain multiple-row height cells, shown as Fig. 4. Here are several definitions to the cells in the double-row placement problem.

Definition 4 (Double-Row Region R_{dr}). *The rectangular region defined by the target two rows to be solved.*

The target rows to be solved by double-row placement form a rectangular box. The region defined by the other rows will be referred to as a region outside the double-row region, denoted by $\overline{R}_{dr}$.

Definition 5 (Splitting Cell). *Any multiple-row height cell spans both rows in R_{dr}.*

In Fig. 4, cells e and i cover both lower and upper row in R_{dr}, so they are considered as splitting cells.

Definition 6 (Crossing cell). *Any multiple-row cell spans only one of the two rows in R_{dr}.*

Cells like g and j in Fig. 4 either take the lower or upper row in R_{dr}, and also intersect with $\overline{R}_{dr}$. They are considered as crossing cells.

There are several cases to this problem. The ideal case is that the double-row placement problem only consists of single-row height cells and double-row height splitting cells, which means all the cells will lie in R_{dr}, shown as Fig. 4(a). Two splitting cells e and i separate the each row into three parts, i.e. partition 1, 2, and 3. But this is not often true due to the existence of crossing cells and large splitting cells. Fig. 4(b) gives a general case for the double-row placement problem where some splitting cells and crossing cells span more than two rows. In this case where cells e, g, and j spread out of the rows, their movements must keep the order within the two rows and not cause any overlap in the other rows. We will first explain the algorithm with the ideal case in Fig. 4(a) and extend it to handle the general cases. For simplicity, we further assume in the ideal case, there is no inter-row connection between cells in the lower and upper row within each partition. The general double-row placement problem without ordering constraints is very difficult, since the general single-row placement problem is already known as $\mathcal{NP}$-hard [31].

3.2.1 *Nested Shortest Path Problem*

We first formulate the ordered double-row placement problem into a nested shortest path problem with outer and inner level.

Table 2: Notations in Ordered Double-Row Placement

M	Maximum displacement for a cell.
d_i	The displacement of cell c_i, $-M \leq d_i \leq M$.
z_i	A splitting cell in the splitting cell set SC.
y_i	A crossing cell in the crossing cell set CC.
v_i	A single-row height cell or crossing cell in the lower row of a partition.
u_i	A single-row height cell or crossing cell in the upper row of a partition.
PC_i	The set of cells in the partition between splitting cell z_{i-1} and z_i.

Then we solve it with a nested dynamic programming algorithm. Table 2 gives the notations used in the ordered double-row placement problem. We define the maximum displacement M such that each cell has $K = 2M + 1$ displacement values. Let z_{ij} denote the jth position for splitting cell z_i. Let r be the number of splitting cells in R_{dr}, b be the number of cells in the lower row of a partition, and t be the number of cells in the upper row of a partition.

The key observation to the ordered double-row placement problem is the independence of sub-problems within each partition providing the positions of splitting cells fixed. For instance, the sub-problem for cells in partition 1 of Fig. 4(a) becomes independent as long as the position of splitting cell e is determined. Similarly, the sub-problem in partition 2 only relies on the positions of the splitting cells e and i. Therefore, if we can determine the positions of the splitting cells, it is possible to solve the corresponding independent sub-problem. With such observation, we formulate a nested shortest path problem shown as Fig. 5, where we solve the positions of all the splitting cells with an outer-level shortest path problem whose edge weights are determined by a set of inner-level problems.

Fig. 5(a) gives the graph representation of the outer-level shortest path algorithm where each node denotes a candidate position of a splitting cell. We need to find the shortest path from s to t. However, the weights of edges in Fig. 5(a) are not determined yet because the minimum placement cost for cells within each partition is still unknown. With the previous independence property, we can compute the weight of any edge $z_{i-1,k} \rightarrow z_{ij}$ by solving the inner-level problem shown in Fig. 5(b). The inner-level problem consists of two shortest path problems for the lower and upper row in the partition. These two shortest path problems are independent due to the assumption in ideal case that there is no inter-row connection in a partition. Node $z_{i-1,k}$ and z_{ij} serve as the starting and terminating node in the inner-level problem.

3.2.2 *Nested Dynamic Programming*

In general any algorithm that solves shortest path can be applied to the nested shortest path problem defined above. For efficiency, we adapt the dynamic programming algorithm in [30] to solve the nested shortest path problem in the ordered double-row placement, which results in a nested dynamic programming scheme. Alg. 3 gives the skeleton of the nested dynamic programming algorithm. To highlight the nesting scheme, we omit the details that are the same as the ordered single-row placement and only keep the simplified key steps. The algorithm calls the function `SolveOuterLevel` to solve the outer-level shortest path problem. The kernel procedure of `SolveOuterLevel` lies in the three loops from lines 7 to 15. The cost of each candidate position is evaluated in lines 10 to 12 where function `ComputeDPCost` computes the cost for z_{i-1} and z_{ij} themselves and function `SolveInnerLevel` solves the inner-level problem for cost in the partition. Within a partition, `SolveInnerLevel` computes the cost of lower and upper row separately with the cost function `ComputeDPCost` and return the total cost. Since the dynamic programming for the inner-level problem is the same as single-row version in the ideal case, the details are omitted.

Algorithm 3 Ordered Double-Row Placement

```
Input: Two ordered sequences of cells.
Output: Shift cells to minimize wirelength.
 1: ... // prepare data $SC$
 2: SolveOuterLevel($SC$);
 3: return
 4:
 5: function SolveOuterLevel($SC$)
 6:     ...
 7:     for each $z_i \in SC$, $i \leftarrow 2$ to $r$ do
 8:         for each $d_i \in [-M, M]$ do
 9:             for each $d_{i-1} \in [-M, M]$ do
10:                 $cost_i(d_i) \leftarrow$ComputeDPCost($d_{i-1}, d_i$)
11:                         +SolveInnerLevel($d_{i-1}, d_i, PC_i$);
12:                 ... // process $cost_i(d_i)$ in DP
13:             end for
14:         end for
15:     end for
16:     ... // apply solution
17: end function
18:
19: function SolveInnerLevel($d_{i-1}, d_i, PC_i$)
20:     $cost_1 \leftarrow$ solve DP for lower row in $PC_i$;
21:     $cost_2 \leftarrow$ solve DP for upper row in $PC_i$;
22:     return $cost_1 + cost_2$;
23: end function
```

The wirelength cost computed in `ComputeDPCost` adopts the cost function defined in [14] for single-row placement. If a cell c_i connects to another cell c_j in the same row and c_j is on the left of c_i, we assume the position of c_j is on the left boundary of the row for wirelength cost computation; if c_j is on the right of c_i in the same row, the position of c_j is assumed to be the right boundary of the row. For any c_j in a different row to c_i, its actual position is used. This wirelength cost turns out to be equivalent to HPWL in single-row placement and the equivalence holds in the ideal case of double-row placement as well.

Lemma 2. *Alg. 3 gives optimal solution for the wirelength cost to the ordered double-row placement under the ideal case.*

The proof is omitted here due to page limit.

The runtime for Alg. 3 turns out to be $\mathcal{O}(M^2 n)$ where n is the total number of cells in R_{dr}. Considering the $r + 1$ partitions defined by r splitting cells, within each partition PC_i, the lower row contains b_i cells and the upper row contains t_i cells. Assume `ComputeDPCost` takes constant time and $n \gg r$. The dynamic programming scheme takes $\mathcal{O}(Mn)$ to solve single-row placement [30]. So solving partition PC_i for one time takes $\mathcal{O}(Mb_i)+\mathcal{O}(Mt_i)$ in `SolveInnerLevel`. The runtime complexity for Alg. 3 can be computed as follows,

$$\begin{aligned} \text{complexity} &\approx \sum_{i=1}^{r+1} M \cdot (\mathcal{O}(Mb_i) + \mathcal{O}(Mt_i)) \\ &= \mathcal{O}(M^2(n-r)) \approx \mathcal{O}(M^2 n). \end{aligned} \tag{6}$$

3.2.3 *Extension To General Cases*

The potential overlaps to $\overline{R}_{dr}$ must be considered due to the existence of large splitting cells and crossing cells in a general case. During the ordered double-row placement, any position of a cell overlapping with any placement site already taken by other cells in $\overline{R}_{dr}$ should be avoided; i.e. assign a very large cost to such positions. We can add a large penalty to a position in `ComputeDPCost` without losing the optimality since such penalty only depends on the position of the cell itself.

However, under a general case, the wirelength cost computed by `ComputeDPCost` in the inner-level problem is no longer always equivalent to HPWL. because a cell in the lower row of a partition may have connection with another cell in the upper row. Such inaccuracy from the wirelength cost usually comes from short inter-row connections, so the overhead is small. Be-

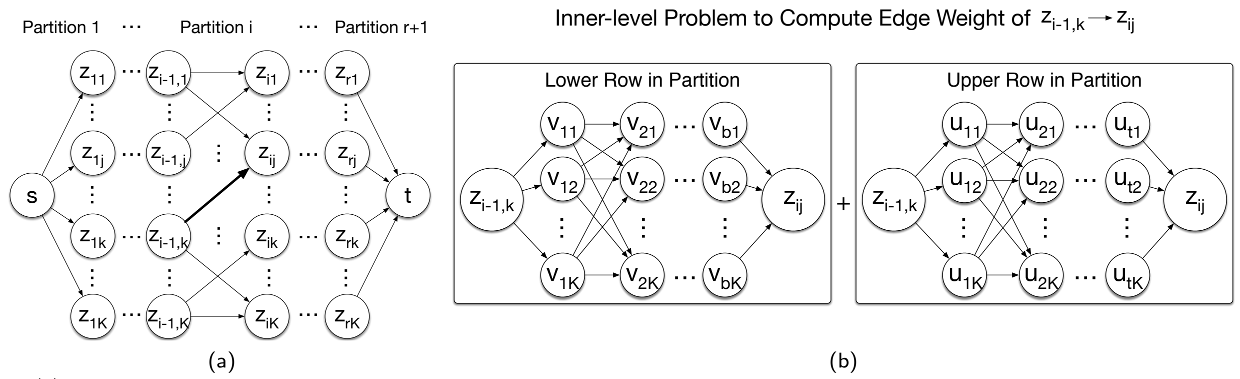

Figure 5: (a) Outer-level shortest path problem that solves the positions of splitting cell $z_1, z_2, \ldots, z_r$. The weights of edges in each partition need to be computed by solving the inner-level problems and (b) an inner-level problem computes the edge weight of $z_{i-1,k} \rightarrow z_{ij}$ by solving the shortest path problem of lower and upper row in the partition with given positions of the splitting cells $z_{i-1,k}$ and z_{ij}.

sides wirelength, the nested dynamic programming scheme can also be adapted to support other objectives, such as displacement and local congestion.

Although ordered double-row placement can minimize wirelength, it may squeeze the whitespaces in dense regions and result in congestion. To mitigate such side effects, we fix the cells in congested regions and only move cells in low-density regions. In general the algorithm can also be applied to resolve overlaps for legalization, but the computation effort becomes an issue for layouts with large amount of overlaps due to its quadratic relation with maximum displacement. Therefore, we adopt it as an incremental optimization technique for legal designs.

4. EXPERIMENTAL RESULTS

Our algorithm was implemented in C++ and tested on an eight-core 3.40 GHz Linux server with 32 GB RAM. Single thread is used in the experiment. We validate our algorithm on two sets of benchmarks. The first set of benchmarks are generated from ISPD05 placement benchmark suite by [11] with only single-row and double-row height cells. Double-row height cells are randomly generated from about 30% single-row height cells. The state-of-the-art wirelength-driven global placer `POLAR` [9] is used for global placement. We obtain the binary from [11] and all the results are collected from our machine. The second set of benchmarks are modified from ICCAD14 placement benchmark suite [32] in which we resize cells such as flip-flops to double-row height and some large cells such as NAND4_X4 and INV_X32 to three-row and four-row height cells. We adopt the evaluation script from ICCAD13 placement contest to verify the legality, wirelength and density of our placement solution. The bin sizes are set to 9×9 row heights according to the evaluation script. The target pin density for APU evaluation is set to the average pin density of top 60% densest bins.

Table 3 shows the information of benchmarks and comparison between our algorithm and [11]. The sizes of the designs vary from 200K to 2M with utilizations from 67.70% to 91.10%. The ratio of multiple-row height cells are shown as "DH". The wirelength for the input global placement solution is shown as "GP", which is not legalized yet. The results of our algorithm is shown as "`MrDP`". Runtime is shown as "CPU" in seconds.

Since [11] only considers wirelength, we first compare wirelength in which `MrDP` achieves smaller HPWL in all benchmarks on an average of 1.2%. We can also see from the table that `MrDP` can achieve even more significant improvement in sHPWL, 3.0% on average, which indicates better cell density in the placement solution. The ABU penalty from `MrDP` is 13.3% smaller than that from [11] and APU penalty shows 13.2% improvement. Although `MrDP` is slightly slower than [11], even the largest benchmark with 2 million cells can be finished within 10 minutes, which is still affordable in placement.

Table 4 gives experimental results on modified ICCAD14 benchmarks. To the best of our knowledge, no published detailed placers are reported to explicitly handle such benchmarks with various multiple-row height cells yet. The ratio of multiple-row height cells varies from 17.17% to 41.09% for different benchmarks, shown as "MH". We keep the same target utilizations as the contest setting. The data under "Initial" denotes the evaluation of initial solutions that still contain overlaps. We can see that `MrDP` achieves 3.0% improvement in HPWL and 3.7% improvement in sHPWL. The cell and pin density penalty also decreases by 22.5% and 15.3% respectively.

We also study the trade-off between performance and runtime for different maximum displacement M in ordered double-row placement in Fig. 6. With the increase of M, wirelength drops while the runtime rises quadratically. The wirelength starts to saturate after M goes larger than 8. To trade-off runtime and performance, we set M to 8 placement sites in the experiment. In addition, although we call [12] in the legalization step, it actually does nothing because the chain move in overlap reduction mode has already removed all overlaps in the experiment.

5. CONCLUSION

In this paper, we have addressed the placement challenges in advanced technology nodes and proposed a detailed placer for heterogeneous-sized cells to help resolve these challenges. Two major techniques have been introduced to generalize the optimization of both single-row height cells and multiple-row height cells, including a chain move scheme to find maximum prefix sum of wirelength improvement and a nested dynamic programming algorithm for double-row placement. Experimental results demonstrate our algorithm outperforms the most recent detailed placer for multiple-row height cells in both wirelength and density.

6. REFERENCES

[1] Mark Po-Hung Lin, Chih-Cheng Hsu, and Yao-Tsung Chang. Recent research in clock power saving with multi-bit flip-flops. In *Proc. MWSCAS*, pages 1–4, 2011.

[2] Chang-Cheng Tsai, Yiyu Shi, Guojie Luo, and Iris Hui-Ru Jiang. FF-Bond: multi-bit flip-flop bonding at placement. In *Proc. ISPD*, pages 147–153, 2013.

[3] Chih-Cheng Hsu, Yu-Chuan Chen, and Mark Po-Hung Lin. In-placement clock-tree aware multi-bit flip-flop generation for power optimization. In *Proc. ICCAD*, pages 592–598, 2013.

[4] Wing-Kai Chow, Jian Kuang, Xu He, Wenzan Cai, and Evangeline F. Y. Young. Cell density-driven detailed placement with displacement constraint. In *Proc. ISPD*, pages 3–10, 2014.

[5] Tung-Chieh Chen, Tien-Chang Hsu, Zhe-Wei Jiang, and Yao-Wen Chang. NTUplace: a ratio partitioning based placement algorithm for large-scale mixed-size designs. In *Proc. ISPD*, pages 236–238, 2005.

Table 3: Comparison of our algorithm with Wu et al. [11]

Design	Size	DH	Util	Target	HPWL			sHPWL			ABU penalty		APU penalty		CPU	
		%	%	Util%	GP	[11]	MrDP	GP	[11]	MrDP	[11]	MrDP	[11]	MrDP	[11]	MrDP
adaptec1_dr	211K	30.18	90.84	91	95.57	91.35	91.03	134.45	96.22	96.88	0.0533	0.0642	0.8943	0.7668	38.3	44.2
adaptec2_dr	255K	30.16	89.12	90	105.75	105.66	104.07	121.10	107.40	105.68	0.0165	0.0154	2.2661	2.0553	42.5	48.6
adaptec3_dr	452K	30.11	78.44	80	241.83	242.13	237.69	305.53	273.94	267.20	0.1314	0.1242	2.6111	2.2966	82.8	84.7
adaptec4_dr	496K	30.19	67.70	75	206.81	208.92	204.94	279.16	253.97	240.62	0.2156	0.1741	2.4462	2.0664	83.6	88.6
bigblue1_dr	278K	30.14	73.44	80	116.95	113.09	112.48	134.39	133.34	127.03	0.1791	0.1293	0.5442	0.3683	36.9	52.6
bigblue2_dr	558K	32.90	68.99	75	159.59	160.86	158.11	230.82	197.11	190.54	0.2253	0.2051	1.2189	1.0881	78.3	101.1
bigblue3_dr	1097K	30.31	91.10	91	413.75	418.69	412.01	499.20	431.31	428.86	0.0301	0.0409	1.9053	1.7502	224.9	264.9
bigblue4_dr	2177K	30.26	73.88	75	881.32	882.51	876.84	1166.86	1099.14	1049.69	0.2455	0.1971	0.9599	0.7562	322.3	438.1
avg.	-	-	-	-	277.69	277.90	274.65	358.94	324.05	313.31	0.1371	0.1188	1.6057	1.3935	113.7	140.4
ratio	-	-	-	-	1.000	1.001	0.989	1.000	0.903	0.873	1.000	0.867	1.000	0.868	1.000	1.235

Table 4: Experimental results on modified ICCAD 2014 benchmarks

Design	Size	MH	Util	Target	HPWL		sHPWL		ABU		APU		CPU
		%	%	Util%	Initial	MrDP	Initial	MrDP	Initial	MrDP	Initial	MrDP	
vga_lcd	165K	21.99	54.98	70	4.19	4.02	4.39	4.20	0.0471	0.0444	0.1310	0.1184	27.3
b19	219K	27.93	52.56	76	3.32	3.17	3.47	3.27	0.0440	0.0295	0.2750	0.2247	31.9
leon3mp	650K	35.61	51.78	70	15.19	14.34	15.76	14.52	0.0377	0.0121	0.1623	0.1107	261.4
leon2	795K	41.09	59.82	70	31.97	31.32	33.88	32.94	0.0595	0.0517	0.1087	0.0779	405.8
dist	133K	26.34	65.23	75	5.06	4.81	5.06	4.81	0.0000	0.0000	0.1704	0.1203	17.0
mult	160K	14.81	60.14	65	2.95	2.76	3.08	2.84	0.0427	0.0300	0.1224	0.1547	20.7
netcard	961K	17.17	47.43	72	41.13	40.23	43.18	42.24	0.0498	0.0499	0.1441	0.1364	296.1
avg.	-	-	-	-	14.83	14.38	15.54	14.97	0.0401	0.0311	0.1591	0.1347	151.4
ratio	-	-	-	-	1.000	0.970	1.000	0.963	1.000	0.775	1.000	0.847	-

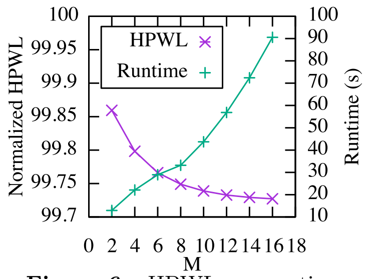

Figure 6: HPWL v.s. runtime

[6] Natarajan Viswanathan, Min Pan, and Chris Chu. FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. In *Proc. ASPDAC*, pages 135–140, 2007.

[7] Myung-Chul Kim, Dong-Jin Lee, and Igor L. Markov. SimPL: An effective placement algorithm. *IEEE TCAD*, 31(1):50–60, 2012.

[8] Myung-Chul Kim, Natarajan Viswanathan, Charles J. Alpert, Igor L. Markov, and Shyam Ramji. MAPLE: multilevel adaptive placement for mixed-size designs. In *Proc. ISPD*, pages 193–200, 2012.

[9] Tao Lin, Chris Chu, Joseph R Shinnerl, Ismail Bustany, and Ivailo Nedelchev. POLAR: placement based on novel rough legalization and refinement. In *Proc. ICCAD*, pages 357–362, 2013.

[10] Jingwei Lu, Hao Zhuang, Pengwen Chen, Hongliang Chang, Chin-Chih Chang, Yiu-Chung Wong, Lu Sha, Dennis Huang, Yufeng Luo, Chin-Chi Teng, et al. ePlace-MS: Electrostatics-based placement for mixed-size circuits. *IEEE TCAD*, 34(5):685–698, 2015.

[11] Gang Wu and Chris Chu. Detailed placement algorithm for VLSI design with double-row height standard cells. *IEEE TCAD*, 2015.

[12] Wing-Kai Chow, Chak-Wa Pui, and Evangeline F. Y. Young. Legalization algorithm for multiple-row height standard cell design. In *Proc. DAC*, pages 83:1–83:6, 2016.

[13] Ulrich Brenner and Jens Vygen. Faster optimal single-row placement with fixed ordering. In *Proc. DATE*, pages 117–121, 2000.

[14] Min Pan, Natarajan Viswanathan, and Chris Chu. An efficient and effective detailed placement algorithm. In *Proc. ICCAD*, pages 48–55, 2005.

[15] Sergiy Popovych, Hung-Hao Lai, Chieh-Min Wang, Yih-Lang Li, Wen-Hao Liu, and Ting-Chi Wang. Density-aware detailed placement with instant legalization. In *Proc. DAC*, pages 122:1–122:6, 2014.

[16] Taraneh Taghavi, Charles Alpert, Andrew Huber, Zhuo Li, Gi-Joon Nam, and Shyam Ramji. New placement prediction and mitigation techniques for local routing congestion. In *Proc. ICCAD*, pages 621–624, 2010.

[17] Myung-Chul Kim, Natarajan Viswanathan, Zhuo Li, and Charles Alpert. ICCAD-2013 CAD contest in placement finishing and benchmark suite. In *Proc. ICCAD*, pages 268–270, 2013.

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

[19] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. DAC*, pages 175–181, 1982.

[20] Jens Vygen. Algorithms for detailed placement of standard cells. In *Proc. DATE*, pages 321–324, 1998.

[21] Andrew B Kahng, Paul Tucker, and Alex Zelikovsky. Optimization of linear placements for wirelength minimization with free sites. In *Proc. ASPDAC*, pages 241–244, 1999.

[22] Andrew B. Kahng, Igor L. Markov, and Sherief Reda. On legalization of row-based placements. In *Proc. GLSVLSI*, pages 214–219, 2004.

[23] Peter Spindler, Ulf Schlichtmann, and Frank M Johannes. Abacus: fast legalization of standard cell circuits with minimal movement. In *Proc. ISPD*, pages 47–53, 2008.

[24] Haitong Tian, Yuelin Du, Hongbo Zhang, Zigang Xiao, and Martin D. F. Wong. Triple patterning aware detailed placement with constrained pattern assignment. In *Proc. ICCAD*, pages 116–123, 2014.

[25] Jian Kuang, Wing-Kai Chow, and Evangeline F. Y. Young. Triple patterning lithography aware optimization for standard cell based design. In *Proc. ICCAD*, pages 108–115, 2014.

[26] Bei Yu, Xiaoqing Xu, Jhih-Rong Gao, Yibo Lin, Zhuo Li, Charles Alpert, and David Z. Pan. Methodology for standard cell compliance and detailed placement for triple patterning lithography. *IEEE TCAD*, 34(5):726–739, May 2015.

[27] Hsi-An Chien, Ye-Hong Chen, Szu-Yaun Han, Hsiu-Yu Lai, and Ting-Chi Wang. On refining row-based detailed placement for triple patterning lithography. *IEEE TCAD*, 34(5):778–793, 2015.

[28] Yibo Lin, Bei Yu, Biying Xu, and David Z. Pan. Triple patterning aware detailed placement toward zero cross-row middle-of-line conflict. In *Proc. ICCAD*, pages 396–403, 2015.

[29] Yuelin Du and Martin D. F. Wong. Optimization of standard cell based detailed placement for 16 nm FinFET process. In *Proc. DATE*, pages 357:1–357:6, 2014.

[30] Yibo Lin, Bei Yu, Yi Zou, Zhuo Li, Charles J. Alpert, and David Z. Pan. Stitch aware detailed placement for multiple e-beam lithography. In *Proc. ASPDAC*, pages 186–191, 2016.

[31] Salim Chowdhury. Analytical approaches to the combinatorial optimization in linear placement problems. *IEEE TCAD*, 8(6):630–639, 1989.

[32] Myung-Chul Kim, Jin Hu, and Natarajan Viswanathan. ICCAD-2014 CAD contest in incremental timing-driven placement and benchmark suite. In *Proc. ICCAD*, pages 361–366, 2014.