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

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

\textsuperscript{1}ECE Department, University of Texas at Austin, Austin, TX, USA
\textsuperscript{2}CSE Department, The Chinese University of Hong Kong, NT, Hong Kong
\textsuperscript{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 \cite{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 \cite{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 \cite{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 \cite{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. \cite{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-
2.1 Power Line Alignment

Power line alignment is a special placement constraint from a multiple-row height cell. Fig. 1 illustrates a 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 f have VDD power rail (VDD) on top and bottom of the cells, and ground rail (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:

\[ \text{HPWL} = \sum_{n \in N} \max_{i \in \Gamma} x_i - \min_{i \in \Gamma} x_i + \max_{i \in \Gamma} y_i - \min_{i \in \Gamma} y_i, \]

where \( N \) denotes the set of interconnections in the circuit.

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

\[ \text{overflow}_\gamma = \max(0, \frac{\text{ABU}_\gamma}{d_t} - 1), \]

\[ \text{ABU} = \frac{\sum_{r \in \Gamma} r \cdot \text{overflow}_\gamma}{\sum_{r \in \Gamma} r}, \Gamma \in \{2, 5, 10, 20\}, \]

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 APU penalty, ICCAD 2013 placement contest defined a scaled wirelength cost to generalize both wirelength and density costs, as shown in Eq. (3).

\[ \text{sHPWL} = \text{HPWL} \cdot (1 + \text{ABU}). \]


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 cell $t$ when it comes to multiple-row height cells. Since a multiple-row height cell is likely to involve overlaps with multiple cells, which results in the failure of position search with previous approach.

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.

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 movement.

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 RecorderCells 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 $O_k$ to the cell pool and update the movement records in the scoreboard. If the cell pool goes empty after a movement,
Table 1: Notations used in Chain Move Algorithm

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pool</td>
<td>The cell pool.</td>
</tr>
<tr>
<td>Board</td>
<td>The scoreboard.</td>
</tr>
<tr>
<td>$p_i^0$</td>
<td>Initial position of Cell $c_i$.</td>
</tr>
<tr>
<td>$p_i$</td>
<td>Candidate position of cell $c_i$.</td>
</tr>
<tr>
<td>$O_i$</td>
<td>The set of cells overlapping with cell $c_i$ at $p_i$.</td>
</tr>
<tr>
<td>$cost_i$</td>
<td>The cost of cell $c_i$ at $p_i$.</td>
</tr>
<tr>
<td>$p_i, O_i, cost_i$</td>
<td>Correspond to best $p_i, O_i, cost_i$, respectively.</td>
</tr>
</tbody>
</table>

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. $cost_i \leftarrow \infty$;
   15. for each $a_j \in A_i$ do
      16. $(cost_i, p_i, O_i) \leftarrow$ ComputeMoveCost($c_i, a_j$);
      17. if $cost_i < cost$ then
         18. $cost_i \leftarrow cost_i$; $p_i \leftarrow p_i$; $O_i \leftarrow O_i$;
      19. end if
   20. end for;
   21. Move $c_i$ to $p_i$;
   22. $Pool$.push($O_i$);
   23. $Board$.last.append($c_i, p_i^0 \to p_i$);
   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,

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

where $\Delta WL$ denotes the wirelength cost, $c_{4d}$ denotes density cost and $c_{ov}$ denotes the overlap cost. The weights $\alpha$ and $\beta$ 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 $c_{4rd}$ denote the cost of area density and $c_{p4d}$ 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 $d_{4d}(b)$ and $d_{p4d}(b)$ denote...
the original area and pin density for bin $b$.

\begin{align}
    c_{ad} &= \sum_{b \in B} \sum_{\gamma \in \Gamma} w_\gamma \cdot f(d_a, \Delta d_a, \text{ABU}_\gamma), \\
    c_{pd} &= \sum_{b \in B} \sum_{\gamma \in \Gamma} w_\gamma \cdot f(d_p, \Delta d_p, \text{APU}_\gamma), \\
    c_d &= 0.5 \times \left( \frac{c_{ad}}{d_i^2} + \frac{c_{pd}}{d_f^2} \right), \\
    f(d, \Delta d, \bar{d}) &= \begin{cases} 
        \frac{\Delta d}{\bar{d}}, & \text{if } d + \Delta d \geq \bar{d}, \\
        0, & \text{otherwise,}
    \end{cases}
\end{align}

where $\Delta d_a$ and $\Delta d_p$ denote the area and pin density increase in each bin, $d_i^2$ and $d_f^2$ 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 $\gamma\%$ 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.

### 3.2.1 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

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

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 $j$th 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 $c_i$ is determined. Similarly, the sub-problem in partition 2 only relies on the positions of splitting cells $c_i$ and $c_j$. 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 $z_{i-1,k} \to z_{ij}$ 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: $\vdots$ // prepare data $SC$
2: SolveOuterLevel($SC$);
3: return
4: function SolveOuterLevel($SC$)
5: $\vdots$
6: for each $z_i \in SC$, $i \leftarrow 2$ to $r$
7: for each $d_i \in [-M, M]$
8: for each $d_{i-1} \in [-M, M]$
9: $\cos_t(d_i) \leftarrow$ ComputeDPCost($d_{i-1}, d_i$)
10: + SolveInnerLevel($d_{i-1}, d_i, PC_i$);
11: $\vdots$ // process $\cos_t(d_i)$ in DP
12: end for
13: end for
14: $\vdots$
15: end for
16: $\vdots$ // apply solution
17: end function
18: function SolveInnerLevel($d_{i-1}, d_i, PC_i$)
19: $\cos_t \leftarrow$ solve DP for lower row in $PC_i$;
20: $\cos_t \leftarrow$ solve DP for upper row in $PC_i$;
21: return $\cos_t + \cos_t$;
22: 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 $c_i$ is on the left of $c_j$, we assume the position of $c_i$ is on the left boundary of the row for wirelength cost computation; if $c_j$ is on the right of $c_i$, the position of $c_i$ is assumed to be the right boundary of the row. For any $c_1$ 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 $O(M^2n)$ 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 $O(Mn)$ to solve single-row placement [30]. So solving partition $PC_i$ for one time takes $O(Mb_i) + O(Mt_i)$ in SolveInnerLevel. The runtime complexity for Alg. 3 can be computed as follows,

$$
\text{complexity} \approx \sum_{i=1}^{r+1} M \cdot (O(Mb_i) + O(Mt_i)) = O(M^2(n - r)) \approx O(M^2n). \quad (6)
$$

3.2.3 Extension To General Cases

The potential overlaps to $R_{dr}$ 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 $R_{dr}$ 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-
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 relationship 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 threads are 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 NAND4X4 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 $9 \times 9$ 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

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

<table>
<thead>
<tr>
<th>Design</th>
<th>Size</th>
<th>DH</th>
<th>Util %</th>
<th>Target %</th>
<th>HPWL</th>
<th>hHPWL</th>
<th>ABU penalty</th>
<th>APU penalty</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptec1dr</td>
<td>211K</td>
<td>30.18</td>
<td>30.84</td>
<td>91</td>
<td>95.57</td>
<td>91.35</td>
<td>91.83</td>
<td>114.45</td>
<td>96.88</td>
</tr>
<tr>
<td>adaptec2dr</td>
<td>255K</td>
<td>30.16</td>
<td>89.12</td>
<td>90</td>
<td>105.75</td>
<td>105.66</td>
<td>104.07</td>
<td>121.10</td>
<td>105.68</td>
</tr>
<tr>
<td>adaptec3dr</td>
<td>452K</td>
<td>30.11</td>
<td>78.44</td>
<td>80</td>
<td>241.83</td>
<td>243.12</td>
<td>237.69</td>
<td>305.53</td>
<td>273.94</td>
</tr>
<tr>
<td>adaptec4dr</td>
<td>490K</td>
<td>30.19</td>
<td>67.70</td>
<td>75</td>
<td>206.81</td>
<td>208.92</td>
<td>204.91</td>
<td>279.16</td>
<td>253.97</td>
</tr>
<tr>
<td>bigblue1dr</td>
<td>278K</td>
<td>30.14</td>
<td>73.44</td>
<td>80</td>
<td>116.95</td>
<td>113.09</td>
<td>112.48</td>
<td>134.39</td>
<td>133.34</td>
</tr>
<tr>
<td>bigblue2dr</td>
<td>559K</td>
<td>32.09</td>
<td>68.99</td>
<td>75</td>
<td>159.59</td>
<td>160.86</td>
<td>158.11</td>
<td>230.82</td>
<td>197.11</td>
</tr>
<tr>
<td>bigblue4dr</td>
<td>1097K</td>
<td>30.31</td>
<td>91.10</td>
<td>90</td>
<td>413.75</td>
<td>418.69</td>
<td>412.01</td>
<td>499.20</td>
<td>431.31</td>
</tr>
<tr>
<td>bigblue4dr</td>
<td>2177K</td>
<td>30.26</td>
<td>78.78</td>
<td>75</td>
<td>861.32</td>
<td>882.51</td>
<td>876.84</td>
<td>1166.86</td>
<td>1099.14</td>
</tr>
<tr>
<td>avg.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>277.69</td>
<td>277.90</td>
<td>274.65</td>
<td>358.94</td>
<td>324.05</td>
</tr>
<tr>
<td>ratio</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.000</td>
<td>1.001</td>
<td>0.989</td>
<td>1.000</td>
<td>0.903</td>
</tr>
</tbody>
</table>

**Table 4:** Experimental results on modified ICCAD 2014 benchmarks

<table>
<thead>
<tr>
<th>Design</th>
<th>Size</th>
<th>MH</th>
<th>Util %</th>
<th>Target %</th>
<th>HPWL</th>
<th>hHPWL</th>
<th>ABU penalty</th>
<th>APU penalty</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgajpl</td>
<td>165K</td>
<td>21.99</td>
<td>54.98</td>
<td>70</td>
<td>4.19</td>
<td>4.02</td>
<td>3.70</td>
<td>0.0141</td>
<td>0.0144</td>
</tr>
<tr>
<td>b19</td>
<td>219K</td>
<td>27.93</td>
<td>52.92</td>
<td>76</td>
<td>3.32</td>
<td>3.17</td>
<td>3.47</td>
<td>0.0540</td>
<td>0.0295</td>
</tr>
<tr>
<td>leon3mp</td>
<td>650K</td>
<td>35.61</td>
<td>51.78</td>
<td>70</td>
<td>15.19</td>
<td>14.34</td>
<td>14.56</td>
<td>0.0377</td>
<td>0.0312</td>
</tr>
<tr>
<td>leon2</td>
<td>795K</td>
<td>41.69</td>
<td>59.80</td>
<td>70</td>
<td>31.32</td>
<td>31.32</td>
<td>32.88</td>
<td>0.0505</td>
<td>0.0550</td>
</tr>
<tr>
<td>dist</td>
<td>133K</td>
<td>26.34</td>
<td>65.25</td>
<td>75</td>
<td>5.06</td>
<td>4.81</td>
<td>5.06</td>
<td>0.0990</td>
<td>0.0980</td>
</tr>
<tr>
<td>multi</td>
<td>160K</td>
<td>14.81</td>
<td>60.14</td>
<td>65</td>
<td>2.95</td>
<td>2.76</td>
<td>2.88</td>
<td>0.0427</td>
<td>0.0400</td>
</tr>
<tr>
<td>netcard</td>
<td>961K</td>
<td>17.17</td>
<td>47.43</td>
<td>72</td>
<td>41.13</td>
<td>40.93</td>
<td>41.83</td>
<td>0.0498</td>
<td>0.0499</td>
</tr>
<tr>
<td>avg.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>14.83</td>
<td>14.38</td>
<td>15.54</td>
<td>0.0401</td>
<td>0.0331</td>
</tr>
<tr>
<td>ratio</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.000</td>
<td>1.001</td>
<td>0.989</td>
<td>1.000</td>
<td>0.903</td>
</tr>
</tbody>
</table>

**Figure 6:** HPWL v.s. runtime