Optimal Multi-Row Detailed Placement for Yield and Model-Hardware Correlation Improvements in Sub-10nm VLSI

Changho Han+, Kwangsoo Han‡, Andrew B. Kahng†‡, Hyein Lee‡, Lutong Wang‡ and Bangqi Xu‡
†CSE and ‡ECE Departments, UC San Diego, La Jolla, CA, USA
+Samsung Electronics Co., Ltd., Hwaseong-si, Gyeonggi-do, South Korea
{kwhan, abk, hyeinlee, luw002, bax002}@ucsd.edu, changho1.han@samsung.com

***Abstract*—In sub-10nm nodes, a change or *step* in diffusion height between adjacent standard cells causes yield loss as well as a form of model-hardware miscorrelation called *neighbor diffusion effect* (NDE). Cell libraries must inevitably have multiple diffusion heights (numbers of fins in PFETs and NFETs) in order to enable flexible exploration of the power-performance envelope for design. However, this brings *step*-induced risks of NDE, for which guardbanding is costly, as well as yield loss. Special filler cells can protect against harmful NDE effects, but are costly in terms of area. In this work, we develop dynamic programming-based single-row and double-row detailed placement optimizations that optimally minimize the impacts of NDE. Our algorithms support a richer set of cell movements than in previous works – i.e., flipping, relocating and reordering within the original row; we also consider cell displacement and flipping costs. Importantly, to our knowledge, our dynamic programming-based optimal detailed placement algorithm is the first to handle multiple rows with multiple-height cells that can be reordered. We further develop a timing-aware approach, which is capable of recovering (or, improving) the worst negative slack (WNS) by creating additional diffusion *steps* around timing-critical cells.**

I. Introduction

In advanced technology nodes where the sizes of geometries are near limits of underlying patterning technology, detailed cell placement has become more restricted by various front-end-of-line (FEOL) rules [6]. Even after such FEOL rules are introduced to guarantee reliable patterning for a given lithographic patterning technology, there exist "weak" patterns that cause model-hardware miscorrelation. An important type of weak pattern is a diffusion *step*, which corresponds to a difference in the heights of the diffusion areas between two neighboring transistors. Figure 1(a) illustrates three diffusion *steps* between four transistors. If the heights of neighboring diffusion regions are different, there is a diffusion *step*, e.g., transistor *T2* has a diffusion *step* to each of *T1* and *T3*. The diffusion *step* causes (i) yield loss due to the corner rounding effect in patterning [24]; and (ii) neighbor diffusion effect (NDE) [2].

Yield loss. In older technology nodes, resolution of conventional 193i lithography is sufficient for patterning of diffusion geometries. Thus, the diffusion region of each transistor is patterned separately. However, in advanced nodes, due to the small geometries and limited resolution of the patterning technologies, it is impractical to pattern each diffusion shape separately. Therefore, the diffusion shapes of transistors are merged and patterned as a single polygon; the transistors are then separated by using diffusion breaks (which are achieved by applying diffusion cuts) [22], as shown in Figure 1(a). Figure 1(b) illustrates the desired pattern of a single polygon to generate the diffusion regions of four transistors. The actual pattern of the polygon is shown in Figure 1(c). Figure 1(d) illustrates the final printed diffusion layout with diffusion cuts. At the boundaries of diffusion where diffusion *steps* exist, fin shapes and diffusion shapes are distorted due to the corner rounding phenomena in lithography. A distorted and/or sharp-angled end of a fin may cause an increase in electrical field, resulting in gate oxide breakdown [18]. Further, such distorted diffusion shapes cause dramatic shifts in threshold voltage (Vt), or even device failure in sub-10nm nodes.[1] This Vt shift or device failure has negative impact on design performance and quality. For example, Vt variation can cause setup time and/or hold time violations in a design. As a result, the maximum frequency that the design can achieve is reduced, or the design can even fail. Our preliminary study (VGA, 85% utilization in an N7 (7nm) design enablement) shows that approximately 60% of the adjacent pairs of cell instances have inter-cell *steps*, with an estimated impact on yield of -3.6%.[2] In light of this, minimizing diffusion *steps* helps to recover the yield of designs by reducing Vt (and thus speed) variation of transistors.

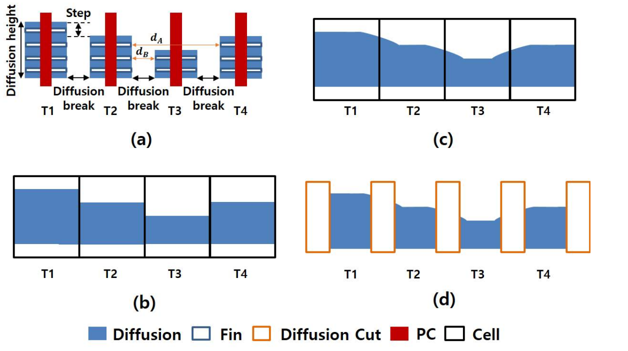

Fig. 1: (a) Diffusion *step* and fin spacing, (b) desired pattern, (c) actual diffusion region showing corner rounding, and (d) diffusion breaks (after diffusion cuts applied).

Neighbor diffusion effect. NDE refers to the impact of the horizontal spacing between diffusion regions on the performance of transistors. More specifically, the drive strength (i.e., I_{on}) and the leakage power (i.e., I_{off}) of a transistor fin is a function of the horizontal spacing to the adjacent diffusion regions of the transistor fin. Since NDE changes the electrical characteristics of transistors, it affects the power, performance and area of designs [2]. For example, Figure 1(a) shows the transistor fins A and B with the spacings to their neighboring diffusion area, i.e., d_A and d_B, respectively. As d_A and d_B are different, I_{on} and I_{off} of the two transistors are different (e.g., $I_{off}(A) = f(d_A) \neq I_{off}(B) = f(d_B)$). Notably, I_{off} (leakage) is changed significantly by NDE. In the 10nm node, leakage variation can be 2× per transistor (i.e., varying across $[0.5\times I_{off,nom}, 2\times I_{off,nom}]$ or an even wider range) depending on the STI process and device types [24]. For a single inverter with a diffusion *step* next to the PFET and a diffusion *step* next to the NFET, the two devices

[1]According to our collaborator [24], there can be >150mV Vt shift in the 10LPE node, which is > 3× the allowed variation range.

[2]Based on guidance from our collaborator [24], after scaling to account for our small testcase sizes, we assume a failure rate of 2ppm for each *step*, and 1ppm after we remove the *step*. See Table IV in Section V below.

in combination result in higher leakage, but their respective impacts on I_{on} (timing) will balance out [24].

In this work, we use a bimodal assumption to simplify the NDE problem: either of two leakage values is assumed for a given transistor, depending on whether there exists a diffusion region on the nearest neighboring site of the transistor. In a conventional place-and-route flow, intra-cell NDE (i.e., NDE effect within a standard cell) is captured by library characterization since the diffusion shapes within a cell are pre-determined. However, it is difficult to capture inter-cell NDE since neighboring diffusion shapes are determined by detailed placement. Thus, in general, library characterization always assumes existence of a full-height neighboring diffusion region on standard cell boundaries, which causes miscorrelation between the model (i.e., library) and the hardware (i.e., diffusion shapes at standard cell boundaries and their device performance impacts) in a design. Therefore, minimizing diffusion *steps* in detailed placement is a key idea toward reduction of model-hardware miscorrelation.

Current limitations and our approach. In order to reduce diffusion *steps*, special non-functional filler cells are developed for instantiation between functional cells [15]. However, the solution space is limited given a fixed layout, and this approach (effectively similar to cell padding) is expensive in terms of area. Other works [5] [14][19][23] propose graph-algorithmic or dynamic programming methods to resolve complex design rules in advanced nodes. However, the solution spaces considered are typically limited due to the assumption of (ordered)-single-row placement.[3] Recent works [13][21] on multi-row detailed placement involve a heuristic approach, and no advanced node rules are considered.

In this paper, we present an optimal multi-row detailed placement optimization, with support of a richer set of cell movements than previous works, to reduce inter-cell *steps*. To our knowledge, ours is the first optimal detailed placement framework to incorporate all types of cell movements, with support of multi-height cells. Our main contributions are summarized as follows.

- We propose an optimal single-row dynamic programming-based approach to minimize a cost function that includes diffusion *steps*. Our proposed algorithm is capable of all types of cell movements – i.e., cell variants, relocating, and reordering (i.e., *P-reordering* with $P > 2$).
- We extend our approach to achieve an *optimal double-row* dynamic programming-based approach with support of movable, and partially reorderable, double-height cells.
- We extend our formulation to a potential timing-aware optimization that leads to $6\times$ increase in *intentional* steps around timing-critical cells to improve the timing performance.
- We achieve up to 90% inter-cell diffusion *step* reduction compared to the current tool flow.

The remainder of this paper is organized as follows. Section II reviews related works. Section III describes the problem formulation and dynamic programming-based single-row detailed placement methodology. Section IV describes the double-row detailed placement flow. In Section V, we describe our experimental setup and results. Section VI gives conclusions and directions for ongoing work.

II. Previous Work

We classify relevant previous works on detailed placement into three categories: (i) detailed placement for advanced nodes, (ii) mixed cell-height placement, and (iii) NDE-aware detailed placement.

[3]Lin et al. [14] propose a *P-reordering* problem. However, only *2-reordering* (i.e., neighbor cell switching) is presented. We describe our methodology to handle the *P-reordering* problem in Section III.

Detailed placement for advanced nodes. To support complex design rules introduced in advanced nodes, the objectives of detailed placement have changed from classical objectives (e.g., wirelength reduction [7][8][9][11][12][17]) in recent years. The works of [14][19][23] resolve triple-patterning issues. Yu et al. [23] propose shortest path and dynamic programming algorithms to solve the ordered single row (OSR) placement. Tian et al. [19] develop a weighted partial MAX SAT approach to solve the OSR problem. Lin et al. [14] propose a local reordered single row refinement (LRSR) and implement a 2-reordering (i.e., neighboring cell switching) approach using a unified graph model. Du and Wong [5] apply a shortest-path algorithm supporting flipping and 2-reordering to address the drain-drain abutment problem in FinFET-based cell placement. The works of [3][6] propose mixed integer linear programming (MILP)-based methods to comply with drain-drain abutment, minimum implant area and minimum oxide jog length rules, and to increase vertical M1 connections.

Mixed cell-height placement. Wu et al. [21] propose a pairing technique to handle double-height cells for detailed placement. Their method simply groups or inflates cells so that all cells become double-height cells, after which a conventional detailed placer can be used. Recently, Lin et al. [13] have proposed a *chain move* scheme along with a nested dynamic programming-based approach to support multiple cell-height placement. They first perform chain moves to save wirelength cost. On top of this, dynamic programming is applied to solve the nested shortest path problem. Other techniques [4] are developed to support non-integer-ratio (e.g., mixture of 8T and 12T cells) mixed cell-height placement.

NDE-aware placement. Ou et al. [16] perform NDE-aware analog placement by modifying and integrating a compact model for NDE into an existing analog placement algorithm. Oh et al. [15] develop special filler cells to mitigate NDE.

In summary, many works such as [5][14][19][23] propose graph or dynamic programming models to resolve complex design rules in advanced nodes. However, their solution spaces are limited by the assumption of (ordered)-single- row placement. Two recent works [13][21] on multi-row detailed placement give heuristic approaches, but no advanced node rules are considered. Our work is distinguished from all previous approaches in that (i) we formulate an optimal single-row and double-row dynamic programming-based approach to minimize a cost function that includes diffusion *steps*; (ii) we support a richer set of cell movements than in previous works – i.e., flipping, relocating and reordering – via a a systematic methodology to handle *P-reordering* with $P > 2$; and (iii) our formulation supports multi-height cells with movable, and partially reorderable, double-height cells (that is, single-height cells can be reordered with double-height cells, but double-height cells are ordered).

III. Single-Row Optimization

In this section, we describe the problem statement and our dynamic programming formulation for single-row detailed placement.

Single-Row Optimization Problem. *Given an initial legalized single-row placement, perturb the placement to minimize inter-cell diffusion* steps.

Inputs: A legalized single-row placement, available cell variants, and cost function of a diffusion *step*.

Output: Optimized single-row detailed placement with minimized overall cost (including inter-cell diffusion *steps*).

Constraints: Maximum displacement range, maximum reordering range, availability of cell flipping.

A. Filler Cell and Step Costs

TABLE I: Cost for one diffusion *step*.

Spacing (sites)	0	1	2	3	4+
Cost	1	$+\infty$	1	1	0

Table I describes inter-cell diffusion *step* cost. For each pair of adjacent cells, if there are zero, two or three empty sites in between, the cost is equal to the number of inter-cell diffusion *steps*; if there are at least four empty sites in between, the cost is always zero. That is, with four or more empty sites we can always assume proper filler cell insertions resulting in no inter-cell diffusion *steps*. Figure 2 shows an example of filler cell insertion between two functional cells that have different diffusion heights at edges that face each other. If the two functional cells have fewer than four empty sites in between, filler cells can only match one of the diffusion heights. As a result, there always exists at least one diffusion *step* that affects one of the two functional cells. However, with a spacing of four or more sites, a legal diffusion height transition can always be achieved by one or more contiguous filler cell(s). Thus, the filler cell(s) can match both the diffusion heights of the two functional cells. In a relevant advanced technology, the minimum filler cell width is two placement sites. Therefore, adjacent functional cells must abut, or have at least two empty sites between them, in order to insert a filler cell. In our implementation, we avoid single-site spacings by assigning infinite cost to such scenarios, as indicated in Table I.

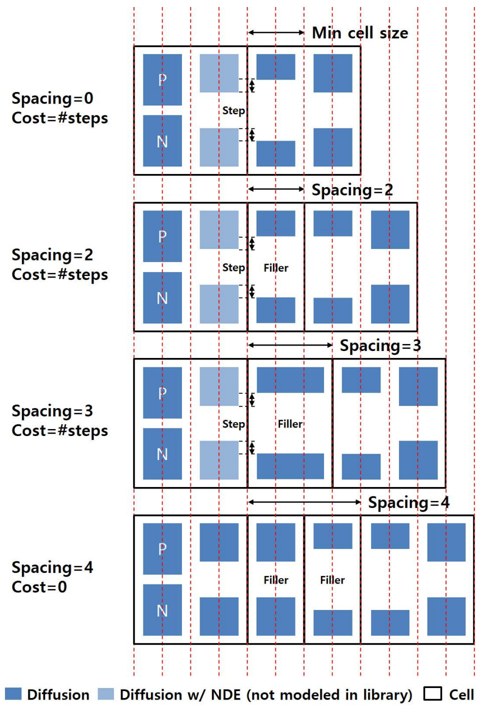

Fig. 2: Filler cell insertion given different spacings.

B. Notations

Table II shows notations used in our formulation. For each cell c_k, **cell index** k is its (left-to-right) sequentially ordered position in the initial placement. Given a set of cells (C) in a row of an initial placement, the leftmost cell is c_1, and the rightmost cell is $c_{|C|}$.

TABLE II: Notations.

Notation	Meaning
C	set of cells in a row of the initial placement
c_k	k^{th} cell in the left-to-right ordered initial placement, where k is *the cell index*
v	a cell variant
$w_{k,v}$	width of c_k with a variant v
$[-x_\triangle, x_\triangle]$	displacement range
x_k	absolute x-coordinate of c_k in the initial placement, in units of placement sites
l	displacement of a cell from the initial placement, in units of placement sites
$[-r, r]$	reordering range
i	number of placed cells
j	position shift of a cell from the initial placement
s	placement status
$d[i][j][v][l][s]$	minimum cost when i cells are placed with case (j, v, l, s)

For each c_k, we define **cell variants** (v) which correspond to different cell orientations and cell layouts with the same functionality. To minimize #diffusion *steps*, we can use several variants of a cell with the same functionality, for which layouts have different diffusion heights. In our experiments below, $v = 0$ indicates the cell orientation in the initial placement, and $v = 1$ indicates the flipped (i.e., mirrored about the y-axis) cell orientation. $w_{k,v}$ is the width of cell c_k with variant v, in units of placement sites. Flipping a cell does not change the set of sites that the cell occupies.

We define the **displacement range** $[-x_\triangle, x_\triangle]$ as the constraint that a cell cannot move more than $x_\triangle$ sites from its initial placement. We use x_k to denote the initial left x-coordinate of c_k, in units of placement sites. Thus, c_k can be placed with its left x-coordinate in the interval $[x_k - x_\triangle, x_k + x_\triangle]$. We use l to denote the **displacement** (in sites) from the initial cell placement (i.e., $l \in [-x_\triangle, x_\triangle]$).

We support cell reordering with a **reordering range** $[-r, r]$, i.e., given r, in the placement solution c_k can have a new sequentially ordered position within the range $k - r, k - r + 1, \ldots, k + r$.

In our dynamic programming, we place one cell at a time from left to right, and the index i is used to indicate that i cells have been placed. Given a cell reordering range $[-r, r]$, cells c_k with $k < i - r$ are placed, $i - r \leq k \leq i + r$ may or may not be placed, and $k > i + r$ are not placed. For the $2r + 1$ cells such that $i - r \leq k \leq i + r$, we use a binary array s to denote the placement status of each cell. Here, s is a binary array of size $(2r + 1)$, i.e., $s \in \{0, 1\}^{2r+1}$. Each bit in the array indicates whether the corresponding cell is placed or not. For example, if we have five cells c_1 to c_5, $i = 3$ and $r = 1$, then s captures the placement status of cells c_2, c_3 and c_4. $s = [0, 1, 1]$ means that c_2 is not placed, while c_3 and c_4 are placed. Figure 3 illustrates six placement solutions with three legal states when $i = 3$. In this example, c_1 must be placed and c_5 must not be placed. We note that the indices of s correspond to k (position in the initial placement), but not the final position. For example, $s[0]$ always represents the status for c_2, and $s[2]$ always represents the status for c_4, regardless of the actual sequence of positions, as shown in Figure 3(b). Also, when we have placed i cells, since cells with index $k < i - r$ must be placed, we must have placed $i - (i - r - 1) = r + 1$ cells that have cell index $i - r \leq k \leq i + r$. Thus, at all times, a legal status array s has exactly $r + 1$ elements equal to 1.

Given i, to identify the last placed cell c_k (that is, the i^{th} cell to have been placed), we define the **position shift** as j, where $k = i + j$. For example, in Figure 3(c), given $i = 3$, the position shift $j = -1$ tells that the last placed cell is c_2, since $2 = 3 + (-1)$.

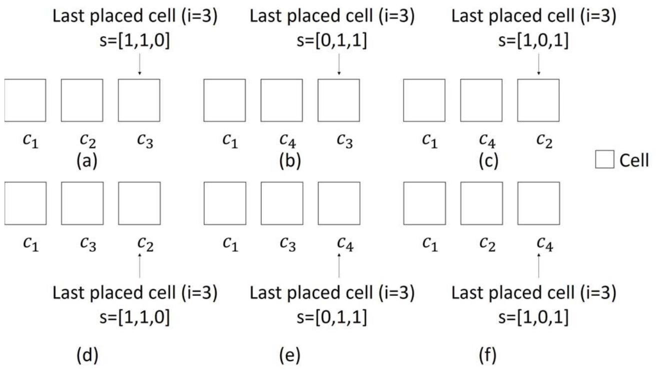

Fig. 3: Illustration of six placement solutions with three legal states given $i = 3$ and $r = 1$.

At the heart of our dynamic programming recurrence, we use $d[i][j][v][l][s]$ to represent the minimum cost when i cells have been placed. From this array, we can obtain the last placed cell c_k, where $k = i+j$. We can also tell the variant v in use, the displacement l, and the status s. We define the above as *case* (j, v, l, s), with i implicitly given, for simplicity. Therefore, we complete the row placement once we reach $i = |C|$, and we obtain the optimal solution by finding the minimum cost among all cases of $i = |C|$. We note that the optimized placement can be traced back from $d[|C|][j][v][l][s]$ all the way to $d[0][j][v][l][s]$.

C. Dynamic Programming Formulation

Algorithm 1 describes our dynamic programming (DP) procedure for single-row placement in detail. Line 2 initializes the DP solution array. Lines 3–13 describe the main algorithm. Starting with placing the first cell, the algorithm incrementally adds (places) cells to the current partial placement solution. Procedure $getNext()$ returns a list of legal next cells and the respective status of each of these cells. Along with legal (j', s') from Line 5, Line 6 checks all possible cases (v', l') considering placement legality and displacement constraints, as shown in Equation (1). Lines 7–9 update the minimum cost for the case (j', v', l', s') when we place the $i' = (i+1)^{st}$ cell. In Lines 14–17, we obtain the minimum cost among all legal cases when $i = |C|$, and Line 18 returns the minimum cost for the current row.

$$x_{i+j} + l + w_{i+j,v} \leq x_{i'+j'} + l' \tag{1}$$

The function $cost(^{i',j',v',l'}_{i,j,v,l})$ calculates the cost as a weighted sum of (i) diffusion *step* cost, (ii) displacement cost, and (iii) cell variant cost, as shown in Equation (2). The diffusion *step* cost is calculated as total #inter-cell diffusion *steps* between the i^{th} and $(i')^{th}$ placed cells. The displacement cost is equal to the absolute value of l'. As noted above, in this work we assume two cell variants: original orientation and flipped orientation. We set the variant cost to one if a cell is flipped ($v' = 1$), and zero otherwise. Two weighting factors α and β are used to balance the three cost terms. We describe experiments regarding the impact of weighting factors in Section V.

$$cost(^{i',j',v',l'}_{i,j,v,l}) = cost_{step} + \alpha \cdot cost_{disp} + \alpha \cdot \beta \cdot cost_{var} \tag{2}$$

Algorithm 2 details our methodology to obtain next status. That is, given a binary status array for i, we would like to obtain the status array for $i' = i + 1$. Line 2 initializes the list of next available $(cellIndex, status)$ combinations. In Line 3, we first shift the status array one bit to the left to obtain the cell placement status for $i' = i+1$. Then, Lines 4–8 check whether cell $c_{i'-r}$ must be placed as the $(i')^{th}$ cell. If we do not place $c_{i'-r}$ as the $(i')^{th}$ cell, then cell $c_{i'-r}$ will be placed out of its reordering range. Thus, we set $s[-r] = 1$ and return so that we make sure to choose $c_{i'-r}$ as the $(i')^{th}$ cell. Lines 10–15 check whether any binary indicator $s[m]$ is equal to zero. If so, $c_{i'+m}$ could be the next legally placed cell. In such a case, we add $(m, nextStatus)$ to the list.

Algorithm 1 Dynamic programming (single-row)

```
1: Initialize for all legal cases $(j, v, l, s)$
2:   $d[0][j][v][l][s] \leftarrow 0$, $d[i][j][v][l][s] \leftarrow +\infty, (0 < i \leq |C|)$
3: for all $i = 0$ to $|C| - 1$ do
4:   for all $d[i][j][v][l][s] \neq +\infty$ do
5:     for all $(j', s') \in$ getNext($s$) do
6:       for all $(v', l')$ do
7:         $i' = i + 1$
8:         $t \leftarrow d[i][j][v][l][s] + cost(^{i',j',v',l'}_{i,j,v,l})$
9:         $d[i'][j'][v'][l'][s'] \leftarrow \min(d[i'][j'][v'][l'][s'], t)$
10:      end for
11:    end for
12:  end for
13: end for
14: $finalCost \leftarrow \infty$
15: for all $(j, v, l, s)$, $i = |C|$ do
16:   $finalCost \leftarrow \min(d[|C|][j][v][l][s], finalCost)$
17: end for
18: Return $finalCost$
```

Algorithm 2 Procedure $getNext$ (single-row)

```
1: Inputs: $s$
2: Initialize $nextList \leftarrow \emptyset$
3: $s \leftarrow$ shiftLeft1Bit($s$)
4: if $s[-r] = 0$ then
5:   $s[-r] \leftarrow 1$
6:   $nextStatus \leftarrow s$
7:   $nextList \leftarrow nextList \cup (-r, nextStatus)$
8:   Return $nextList$
9: end if
10: for all $m \in [-r, r]$ do
11:   if $s[m] = 0$ then
12:     $nextStatus \leftarrow s$
13:     $nextStatus[m] \leftarrow 1$
14:     $nextList \leftarrow nextList \cup (m, nextStatus)$
15:   end if
16: end for
17: Return $nextList$
```

IV. Double-Row Optimization

In this section, we describe the problem statement and the dynamic programming approach for *double-row detailed placement considering double-height cells as well as reordering, flipping and available cell variants.*

Double-Row Optimization Problem. *Given an initial legalized double-row placement with double-height cells, perturb the placement within each row to minimize inter-cell diffusion* steps.

Inputs: Legalized double-row placement, available cell variants, and cost function of a diffusion *step*.

Output: Optimized double-row detailed placement with minimized overall cost (including inter-cell diffusion *steps*).

Constraints: Maximum displacement range, maximum reordering range, availability of cell flipping.

We make the following assumptions with respect to this problem statement.

Assumption 1. *Cell rows can be fully separated from each other every two consecutive rows.*

In the case of placement rows that contain only single-height cells, the assumption is correct by definition. However, for any cell row, a double-height cell that occupies sites in the row must span to either the upper neighboring row or the lower neighboring row, but not both. Figure 4(a) shows such separable pairs of cell rows, where rows 1 and 2 with double-height cells do not interfere with rows 3 and 4. By contrast, in Figure 4(b), row 2 has double-height cells E and F which interfere with both row 1 and row 3, violating our assumption. Given the interleaving of VDD/VSS power rails in modern libraries, our assumption is normally satisfied. In other words, all double-height cells in the current technology node tend to have the same power rail configuration. (In Figure 4(b), cell F has a different type of power rail design (VDD-VSS-VDD) than the other double-height cells (VSS-VDD-VSS).) We do not have such double-height library cells in the current technology node.[4]

Assumption 2. *The relative positions among double-height cells are fixed.*

For two double-height cells A and B, if A is initially to the left of B ($x_A < x_B$), then we require that in our final placement, c_A remains to the left of c_B. We note that double-height cells usually are complex functional cells (e.g., flip-flops) and that all double-height cells span more than 12 placement sites in width in our N7 FinFET technology library. Given the maximum displacement range $x_\Delta = 7$ that we apply to experiments for all design blocks, Assumption 2 practically does not sacrifice solution quality. We note that we still allow reordering between a single-height cell and a double-height cell (thus, the double-height cells are *partially reorderable*) so as to maximize the *steps* reduction.

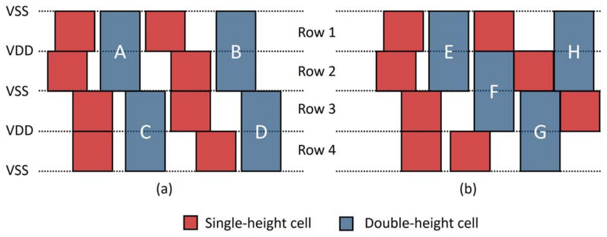

Fig. 4: Illustrations of double-height cells in placement rows. (a) Separable pairs of cell rows, reflecting power rail design of double-height cells in current N10 libraries. (b) Non-separable pairs of cell rows.

Given the above assumptions, our approach can provide optimal placement solutions for two consecutive rows sharing common double-height cells. Overall, double-row optimization uses single-row optimization as a basic building block. From each double-height cell, we invoke separate single-row optimization for each of the two rows, and merge the solutions once we encounter the next double-height cell. The merging is designed to preserve all optimal candidates, while enabling movable and partially reorderable double-height cells. Our development is similar to that of Algorithm 1, where we saw that given the minimum costs of all *cases* (j, v, l, s) for i, we could derive the minimum costs of all *cases* (j', v', l', s') for $i' = i + 1$. Now, let us extend the definition of *case* to support double-height cells. We define $CASE$ $(v, l, j_0, s_0, j_1, s_1)$ given I, where I is the number of placed *double-height* cells. Subscripts 0 and 1 refer to row 0 and row 1, respectively. In Algorithm 1, we obtain the last placed cell c_k from i and j. Here, in double-row optimization, we know exactly the last placed double-height cell, and we would like to obtain i_0 and

[4]Our collaborator [24] at a major advanced foundry indicates that all double-height cells have only one power rail configuration in the 10LPE node. Cells with height of four or more rows account for less than 1% of all instances, and thus our formulation can be easily adopted if we just assume that these very large (height $\geq$ four rows) cells are fixed.

Algorithm 3 Dynamic programming (double-row)

```
1: **Initialize** $DHCellList \leftarrow getOrigDHOrdering()$
2: **Initialize for all legal CASES** $(v, l, j_0, s_0, j_1, s_1)$
3:     $D[0][v][l][j_0][s_0][j_1][s_1] \leftarrow 0$
       $D[I][v][l][j_0][s_0][j_1][s_1] \leftarrow +\infty, (0 < I \leq |DHCellList| + 1)$
4: **for all** $I = 0$ to $|DHCellList|$ **do**
5:     **for all** $D[I][v][l][j_0][s_0][j_1][s_1] \neq +\infty$ **do**
6:         **for all** legal $(v', l', j'_0, s'_0, j'_1, s'_1)$ **do**
7:             $I' = I + 1$
8:             $t \leftarrow D[I][v][l][j_0][s_0][j_1][s_1] + Cost\left(\substack{I', v', l', j'_0, s'_0, j'_1, s'_1 \\ I, v, l, j_0, s_0, j_1, s_1}\right)$
9:             $D[I'][v'][l'][j'_0][s'_0][j'_1][s'_1] \leftarrow$
                 $\min(D[I'][v'][l'][j'_0][s'_0][j'_1][s'_1], t)$
10:        **end for**
11:    **end for**
12: **end for**
13: **for all** $(v, l, j_0, s_0, j_1, s_1)$, $I = |DHCellList|$ **do**
14:     $sol \leftarrow \min(D[I][v][l][j_0][s_0][j_1][s_1], sol)$
15: **end for**
16: **Return** $sol$
```

Algorithm 4 *Cost* (double-row)

```
1: **Inputs:** $I, v, l, j_0, s_0, j_0, s_0, I', v', l', j'_0, s'_0, j'_1, s'_1$
2: $k_0 \leftarrow getK(I, 0)$, $k_1 \leftarrow getK(I, 1)$
3: $k'_0 \leftarrow getK(I', 0)$, $k'_1 \leftarrow getK(I', 1)$
4: $i_0 \leftarrow k_0 + j_0$, $i_1 \leftarrow k_1 + j_1$
5: $i'_0 \leftarrow k'_0 + j'_0$, $i'_1 \leftarrow k'_1 + j'_1$
6: $d_0 \leftarrow optSR_0\left(\substack{i'_0, j'_0, v', l', s'_0 \\ i_0, j_0, v, l, s_0}\right)$
7: $d_1 \leftarrow optSR_1\left(\substack{i'_1, j'_1, v', l', s'_1 \\ i_1, j_1, v, l, s_1}\right)$
8: $totCost \leftarrow d_0 + d_1$
9: **Return** $totCost$
```

i_1 (number of cells placed in row 0 and row 1, respectively). These can be obtained from j_0 and j_1. Given the double-height cell's initial position k_0 in row 0 and k_1 in row 1, $i_0 = k_0 - j_0$ and $i_1 = k_1 - j_1$. The values of v, l, s_0 and s_1 can be obtained directly from *CASE*.

We give a precise description of our double-row dynamic programming in Algorithm 3. Line 1 obtains the double-height cell sequence from the initial (i.e., input) two-row placement. We note that two virtual double-height cells are added to "pad" the input at the start and at the end of the placement rows, respectively. Lines 2-3 initialize the DP solution array for double-height cells. Lines 4-12 describe the main algorithm. Starting with the (left) virtual double-height cell, the algorithm incrementally places double-height cells and updates minimum costs for all *CASES*. In Lines 13–15, we obtain the minimum cost among all legal *CASES* when we reach the ending (right) virtual cell ($I = |DHCellList|$), and Line 16 returns the minimum cost for two rows.

Algorithm 4 describes the cost function in our double-row DP. Line 2 retrieves the double-height cell position in the initial placement for each of the rows. Line 3 gets the next double-height cell similarly. Line 4 obtains the numbers of cells (i_0 and i_1) that have been placed for the two rows. And, Line 5 obtains the numbers of cells (i'_0 and i'_1) that we must place by the time we reach the next double-height cell. For example, for row 0, we need to place cells starting from the *case* (j_0, v, l, s_0) with i_0, until we reach the *case* (j'_0, v', l', s'_0) with i'_0. The above can be achieved by $optSR$ – a modified version of the single-row dynamic programming. In $optSR$, we make sure that we do not place any double-height cells other than $c_{i'_0}$. Thus, Assumption 2 is maintained. In Lines 8 and 9, we return the two-row sum of costs.

We highlight the fact that in our implementation, given the starting *case* (j, v, l, s) with i, $optSR$ calculates all minimum costs of *case* (j', v', l', s') with i', where $k' = i' + j'$, within one functional call to our single-row DP. With this, #single-row DPs is proportional only to #*cases*, rather than to #*CASES*.

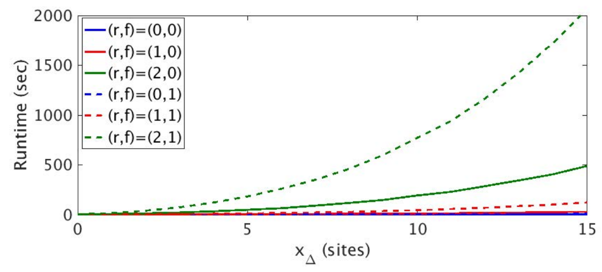

Fig. 5: Sensitivity of runtime to (x_Δ, r, f).

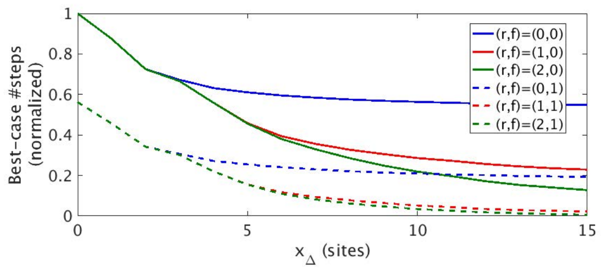

Fig. 6: Sensitivity of #*steps* to (x_Δ, r, f).

V. Experiments

We implement our dynamic programming in C++ with OpenAccess 2.2.43 [27] to support LEF/DEF [26], and with OpenMP [29] to enable thread-level parallelism. We perform experiments in an N7 FinFET technology with multi-height triple-Vt libraries from a leading technology consortium. We apply our detailed placement optimization to ARM Cortex M0 and four design blocks (AES, JPEG, VGA and MPEG) from OpenCores [28]. Design information is summarized in Table III. We synthesize designs using *Synopsys Design Compiler L-2016.03-SP4* [30], and then place and route using *Cadence Innovus Implementation System v15.2* [25]. All experiments are performed with 8 threads on a 2.6GHz Intel Xeon server.

TABLE III: Design information.

design	#inst	clock period
AES	∼12K	500ps
M0	∼10K	500ps
JPEG	∼54K	500ps
VGA	∼69K	500ps
MPEG	∼14K	500ps

A. Scalability/Sensitivity Study

To assess the scalability of our approach, we sweep (x_Δ, r), i.e., maximum allowed cell displacement x_Δ (in placement sites) and maximum allowed one-sided reordering r, and study the impact on runtime. In this experiment, we sweep x_Δ from 0 to 15, and r from 0 to 2. A cell can freely move across 31 placement sites, and can have up to 5 different positions in a placement row, if we set $x_\Delta = 15$ and $r = 2$. We also experiment with enabling ($f = 1$) or disabling ($f = 0$) of cell flipping. We set $(\alpha, \beta) = (0, 0)$ as these parameters do not have any impact on the complexity of our formulation. We use design block AES for this study.

Our study results are shown in Figure 5. For cell flipping, we can see that changing from $(x_\Delta, r, f) = (15, 2, 0)$ to $(15, 2, 1)$ incurs a runtime increase of 4×. We also find that the runtime generally grows quadratically with the number of available placement sites per each cell. However, for cell reordering, there is a dramatic increase in runtime as r goes up, e.g., we observe 12× runtime increase going from $r = 1$ to $r = 2$.

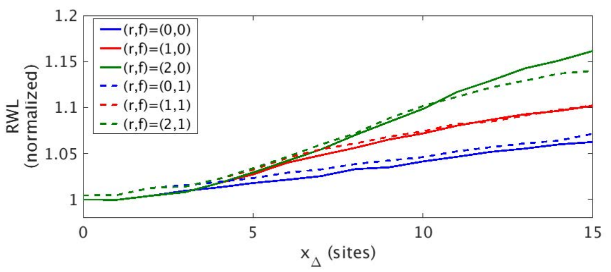

Fig. 7: Sensitivity of routed wirelength to (x_Δ, r, f).

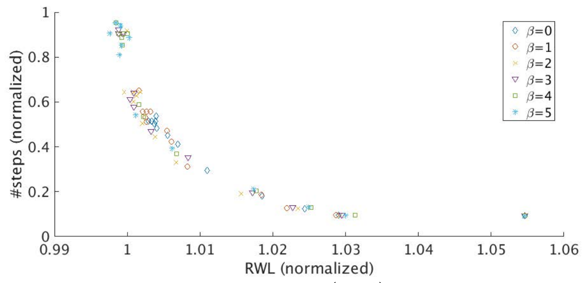

Fig. 8: Impacts of weighting factors (α, β) on the tradeoff between RWL and #*steps*.

Figures 6 and 7 show #diffusion *steps* and routed wirelength (RWL) as we sweep (x_Δ, r, f). Since our algorithm only optimizes #diffusion *steps* when $(\alpha, \beta) = (0, 0)$, here we see RWL that corresponds to a best-case (minimized) #*steps*. We see from Figure 6 that enabling flipping can reduce #*steps* by half even without cell movement. As x_Δ increases, flipping shows consistent advantage over non-flipping cases in terms of #*steps*, at a minor cost of RWL. Moreover, for $f = 1$, there is only 1% benefit of using $r = 2$ over $r = 1$, at the cost of 12× the runtime; this suggests that $r \geq 2$ may not offer significant benefit in reducing #*steps*. In Figure 7, RWL increases linearly as x_Δ goes up. Based on these studies, to balance solution quality and runtime we apply $(x_\Delta, r, f) = (7, 1, 1)$ in all of the following experiments.

B. Study of Weighting Factors

We also investigate impacts on RWL and #*steps* of the weighting factors (α, β) for cell displacement and flipping. We sweep α from 0 to 1 with step size of 0.1 (with additional points at $\alpha = 0.01$ and 0.05), and β from 0 to 5 with step size of 1. We perform this experiment using design block AES, with results shown in Figure 8. Given fixed β (i.e., $\beta = 1$ in red dots), as α changes, there is a clear tradeoff between RWL and #*steps*. Notably, compared to a displacement-oblivious ($\alpha = 0$) optimization, using $\alpha = 0.01$ can directly reduce the RWL overhead from 6% to 3% compared to the routing of the original placement solution, without sacrificing #*steps*. Therefore, we apply $\alpha = 0.01$ in all following experiments. Similarly, we choose $\beta = 1$ as our parameter setting for all following experiments.

C. Main Results

We apply our double-row dynamic programming-based optimization to all our design blocks using the aforementioned parameter settings, i.e., $(x_\Delta, r, f) = (7, 1, 1)$ and $(\alpha, \beta) = (0.01, 1)$. Table IV shows the *step* reduction, runtime and estimated yield improvement for all five design blocks. We also report the impact on other metrics, i.e., routed wirelength (RWL), worst negative slack (WNS) and leakage power as reported by the place-and-route tool [25]. The results are shown in Table IV. For all designs, we achieve up to 90% reduction in #*steps* at the cost of around 3% RWL increase. The results also show that our optimization has negligible impact on WNS and that we can slightly improve the leakage. In addition, we perform a preliminary yield estimation assuming 2ppm failure rate for each *step*, and 1ppm failure rate after we remove the *step* (recall Footnote 2). Based on this assumption, we can see a yield improvement of up to 3.59% for a design block of 69K instances. We note that the yield improvement is expected to grow markedly with the die size. A larger design of millions of instances may see more benefits. Figure 9 shows the layouts of placements before and after our optimization.

We also investigate the improvement achieved by our double-row optimization over single-row optimization and previous works. We compare double-row (DR) optimization to (i) single-row (SR) optimization [5][14], and (ii) ordered double-row (ODR) optimization [13]. For (i), we use the proposed methodology in Section III and fix the locations of all multi-height cells. We note that our SR implementation is equivalent to [5][14], supporting neighboring cell swapping and cell flipping with the adaptation of NDE. In SR, we use the same displacement range and reordering range as in DR. For (ii), we simply run our DR optimization with zero reordering range to achieve an ODR equivalent to [13]. The comparison is shown in Table V. For design blocks with fewer double-height cells, SR performance is competitive with that of ODR. However, for design blocks with more double-height cells, ODR is significantly better (up to 21% more *step* reduction) than SR due to movable double-height cells. The results show that DR effectively reduces the diffusion *steps* by around half compared to SR, and by around 40% compared to ODR. On average, DR has 11.6% more *step* reduction than ODR, and 17.7% more than SR, compared to the initial number of diffusion *steps*. This suggests the importance of supporting movable and reorderable double-height cells, as there will be substantial benefits.

D. Performance Improvement Using Intentional Steps

Finally, similar in spirit to [10], we explore the possibility of improving design performance with *intentional steps* – i.e., using filler cells that create an *intentional step* to the neighboring timing-critical functional cell so as to improve the timing of that functional cell.[5] In the cost function, we use a third weighting factor δ to represent the benefit of an *intentional step* to a timing-critical cell. We sweep δ from 0 to -2 with a step size of -0.2. We select 5% of all cells as timing-critical cells and perform optimization using all design blocks. The results are shown in Figure 10. We use $orig.opt$ to represent the results with $\delta = 0$, and $time.opt$ to represent the results with $\delta = -2$. We achieve up to 6× increase in #*filler-induced steps* incident to timing-critical cells, at the cost of slightly increased #*non-filler-induced steps* to non-timing-critical cells. This translates to 2.47 *steps* per timing-critical cell in $time.opt$, compared to 0.42 *steps* in $orig.opt$. Overall, we can still reduce 50% of total *steps*, showing the effectiveness of our algorithm. We note that as we are making more *intentional steps* to timing-critical cells, we have a smaller

[5]An *intentional inter-cell step* may increase/decrease the drive strength of the function cell. E.g., a *step* adjacent to a PFET may decrease the drive strength while a *step* adjacent to an NFET may increase the drive strength. Here, instead of using a filler cell to match diffusion heights for both the NFET and the PFET of the function cell (to reduce #*steps*), we create a filler-induced *intentional step* by matching the diffusion height for only the PFET, thus increasing the drive strength for the NFET. We note that exact timing and power impacts and tradeoffs will vary with STI processes.

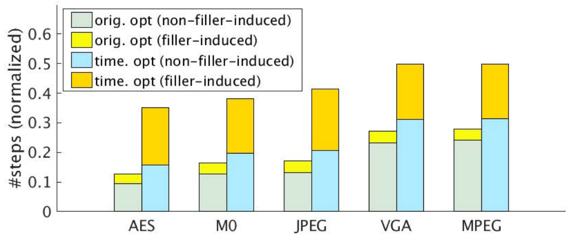

Fig. 10: Comparison of #filler-induced *steps* and total #*steps* for all design blocks before ($orig.opt$, $\delta = 0$) and after ($time.opt$, $\delta = -2$) using *intentional steps*.

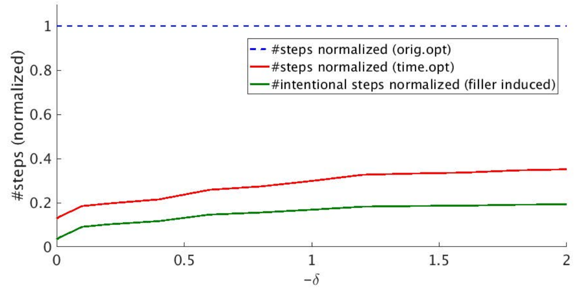

Fig. 11: Sensitivity of filler-induced *steps* to δ. (AES)

solution space for non-timing-critical cells. Thus, we also generate more *steps* to non-timing-critical cells. We furthermore observe that as δ decreases, there is an upper bound of #*intentional steps* that we can achieve, as shown in Figure 11. This may help set expectations for benefits that might be derived from a more comprehensive, timing-aware flow (which we leave for future work).

VI. Conclusions

In this work, we present an *optimal* dynamic programming-based single-/double-row detailed placement methodology to minimize diffusion *steps* in sub-10nm VLSI, for improved yield and mitigation of NDE. Our work achieves several improvements as compared to previous works: (i) optimal dynamic programming with support of a richer set of cell movements, i.e., flipping, relocating and enhanced reordering; (ii) optimal double-row dynamic programming *with support of movable and reorderable double-height cells*; and (iii) a novel performance improvement technique using *intentional steps*. The proposed techniques achieve up to 90% reduction of inter-cell diffusion *steps*, with scalable runtime and high die utilization in an N7 node enablement. Open directions for future research include (i) optimal multi-row multi-height detailed placement; and (ii) a more comprehensive timing-aware optimization flow.

References

[1] S.-H. Baek, H.-Y. Kim, Y.-K. Lee, D.-Y. Jin, S.-C. Park and J.-D. Cho, "Ultra High Density Standard Cell Library Using Multi-Height Cell Structure", *Proc. SPIE*, 2008, pp. 72680C-72680C.
[2] D. C. Chen, G. S. Lin, T. H. Lee, R. Lee, Y. C. Liu, M. F. Wang, Y. C. Cheng and D. Y. Wu, "Compact Modeling Solution of Layout Dependent Effect for FinFET Technology", *Proc. ICMTS*, 2015, pp. 110-115.
[3] P. Debacker, K. Han, A. B. Kahng, H. Lee, P. Raghavan and L. Wang, "Vertical M1 Routing-Aware Detailed Placement for Congestion and Wirelength Reduction in Sub-10nm Nodes", *Proc. DAC*, 2017, pp. 51:1-51:6.
[4] S. Dobre, A. B. Kahng and J. Li, "Mixed Cell-Height Implementation for Improved Design Quality in Advanced Nodes", *Proc. ICCAD*, 2015, pp. 854-860.
[5] Y. Du and M. D. F. Wong, "Optimization of Standard Cell Based Detailed Placement for 16nm FinFET Process", *Proc. DATE*, 2014, pp. 1-6.

TABLE IV: Experimental results for all design blocks.

Design	Util	#steps		RWL (μm)		WNS (ns)		Leakage (mW)		Runtime	Est. Yield
	(%)	Init	Final (Δ%)	Init	Final (Δ%)	Init	Final	Init	Final (Δ%)	(sec)	Impr. %
AES	85%	7973	750 (-90.6%)	31873	32898 (+3.2%)	-0.013	-0.016	16.1	15.8 (-1.9%)	37	+0.71
M0	85%	6588	842 (-87.2%)	27670	28470 (+2.9%)	-0.043	-0.087	18.9	18.6 (-1.7%)	38	+0.57
JPEG	85%	34760	4555 (-86.9%)	101000	105550 (+4.5%)	-0.019	-0.004	96.3	94.5 (-1.8%)	156	+2.86
VGA	85%	50766	11816 (-76.7%)	208155	214169 (+2.9%)	-0.137	-0.118	208.3	205.5 (-1.3%)	195	+3.59
MPEG	85%	9994	2402 (-76.0%)	38896	39950 (+2.7%)	-0.005	-0.033	33.2	33.0 (-1.8%)	25	+0.75

TABLE V: Comparison of diffusion *steps* with SR (to match [5][14]), ODR (to match [13]) and DR. DH%:= % of double-height cells.

design	DH%	Init	SR	ODR	DR
AES	4.3%	7973	1278 (-84.0%)	1869 (-76.6%)	750 (-90.6%)
M0	8.4%	6588	1612 (-75.5%)	1742 (-73.6%)	842 (-87.2%)
JPEG	8.3%	34760	9275 (-73.3%)	8341 (-76.0%)	4555 (-86.9%)
VGA	24.8%	50766	27054 (-46.7%)	16405 (-67.7%)	11816 (-76.7%)
MPEG	23.0%	9994	5071 (-49.3%)	3444 (-65.5%)	2402 (-76.0%)
Avg.	–	1.00×	0.34×(-65.8%)	0.28×(-71.9%)	0.16×(-83.5%)

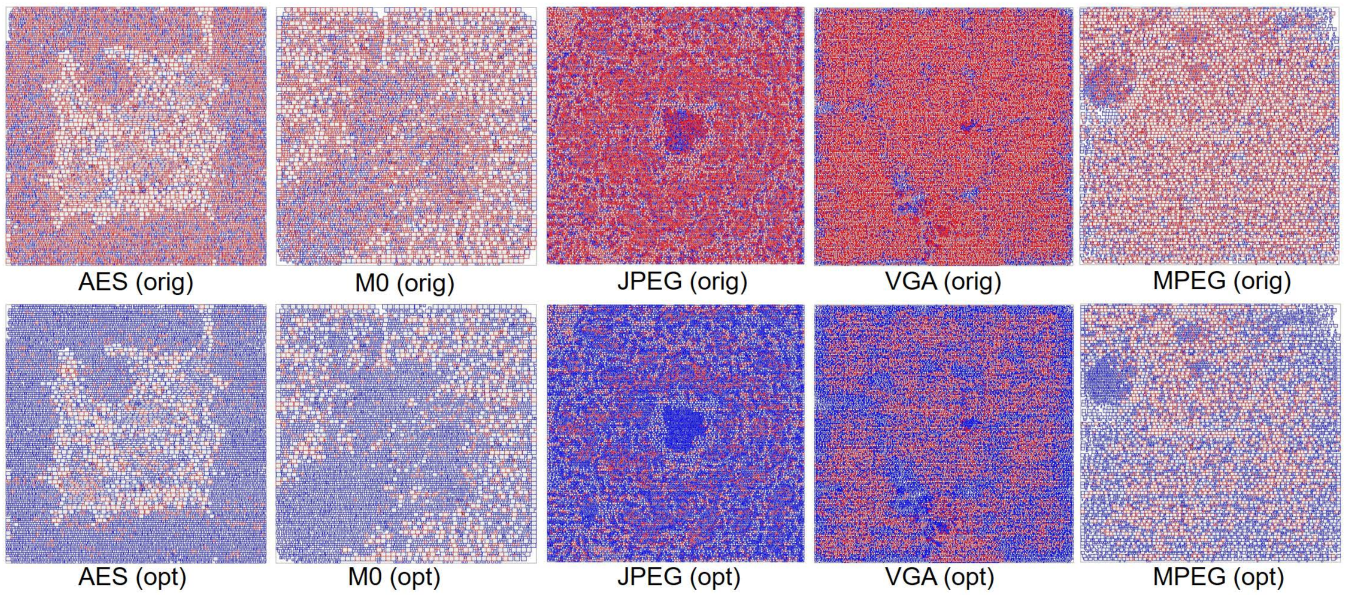

Fig. 9: Layouts of placement before (orig) and after (opt) our optimization. Red color indicates cell instances with diffusion *steps* and blue color indicates cell instances without diffusion *steps*.

[6] K. Han, A. B. Kahng and H. Lee, "Scalable Detailed Placement Legalization for Complex Sub-14nm Constraints", *Proc. ICCAD*, 2015, pp. 867-873.

[7] D. Hill, "Method and System for High Speed Detailed Placement of Cells Within an Integrated Circuit Design", *US Patent 6370673*, 2002.

[8] S.-W. Hur and J. Lillis, "Mongrel: Hybrid Techniques for Standard Cell Placement", *Proc. ICCAD*, 2000, pp. 165-170.

[9] A. B. Kahng, I. L. Markov and S. Reda, "On Legalization of Row-Based Placements", *Proc. GLSVLSI*, 2004, pp. 214-219.

[10] A. B. Kahng, P. Sharma and R. O. Topaloglu, "Exploiting STI Stress for Performance", *Proc. ICCAD*, 2007, pp. 83-90.

[11] A. B. Kahng, P. Tucker and A. Zelikovsky, "Optimization of Linear Placements for Wirelength Minimization with Free Sites", *Proc. ASP-DAC*, 1999, pp. 241-244.

[12] S. Li and C.-K. Koh, "Mixed Integer Programming Models for Detailed Placement", *Proc. ISPD*, 2012, pp. 87-94

[13] Y. Lin, B. Yu, X. Xu, J.-R. Gao, N. Viswanathan, W.-H. Liu, Z. Li, C. J. Alpert and D. Z. Pan, "MrDP: Multiple-row Detailed Placement of Heterogeneous-sized Cells for Advanced Nodes", *Proc. ICCAD*, 2016, pp. 7:1-7:8.

[14] Y. Lin, B. Yu, B. Xu and D. Z. Pan, "Triple Patterning Aware Detailed Placement Toward Zero Cross-Row Middle-of-Line Conflict", *Proc. ICCAD*, 2015, pp. 396-403.

[15] S.-K. Oh, "Standard Cell Library, Method of Using the Same, and Method of Designing Semiconductor Integrated Circuit", *US Patent Application,*, US20160055283, February 2016.

[16] H.-C. Ou, K.-H. Tseng, J.-Y. Liu, I.-P. Wu and Y.-W. Chang, "Layout-Dependent-Effects-Aware Analytical Analog Placement", *IEEE Trans. on CAD* 35(8) (2016), pp. 1243-1254.

[17] M. Pan, N. Viswanathan and C. Chu, "An Efficient and Effective Detailed Placement Algorithm", *Proc. ICCAD*, 2005, pp. 48-55.

[18] M. Tarabbia, A. Mittal and N. Hindawy, "Forming FinFET Cell with Fin Tip and Resulting Device", *US Patent App*, US20150137203.

[19] H. Tian, Y. Du, H. Zhang, Z. Xiao and M. D. F. Wong, "Triple Patterning Aware Detailed Placement with Constrained Pattern Assignment", *Proc. ICCAD*, 2014, pp 116-123.

[20] C.-H. Wang, Y.-Y. Wu, J. Chen, Y.-W. Chang, S.-Y. Kuo, W. Zhu and G. Fan, "An Effective Legalization Algorithm for Mixed-Cell-Height Standard Cells", *Proc. ASP-DAC*, 2017, pp. 450-455.

[21] G. Wu and C. Chu, "Detailed Placement Algorithm for VLSI Design with Double-Row Height Standard Cells", *IEEE Trans. on CAD* 35(9) (2016), pp. 1569-1573.

[22] R. Xie, K.-Y. Lim, M. G. Sung and R. R.-H. Kim, "Methods of Forming Single and Double Diffusion Breaks on Integrated Circuit Products Comprised of FinFET Devices and The Resulting Products", *US Patent, US9412616*, August 2016.

[23] B. Yu, X. Xu, J.-R. Gao, Y. Lin, Z. Lee, C. J. Alpert and D. Z. Pan, "Methodology for Standard Cell Compliance and Detailed Placement for Triple Patterning Lithography", *IEEE Trans. on CAD* 34(5) (2015), pp. 726-739.

[24] Model-Hardware Correlation Team, *Samsung Electronics Co., Ltd.*, Nov. 2016.

[25] Cadence Innovus User Guide, http://www.cadence.com

[26] LEF/DEF reference 5.7. http://www.si2.org/openeda.si2.org/projects/lefdef

[27] Si2 OpenAccess. http://www.si2.org/?page=69

[28] OpenCores: Open Source IP-Cores, http://www.opencores.org

[29] OpenMP Architecture Review Board, "OpenMP Application Program Interface, Version 4.0".

[30] Synopsys Design Compiler User Guide, http://www.synopsys.com