This paper appeared in: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 39 (2020), 2872-2885

BonnCell: Automatic Cell Layout in the 7nm Era

Pascal Cremer, Stefan Hougardy, Jannik Silvanus, and Tobias Werner

***Abstract*—Multi patterning technology used in 7nm technology and beyond imposes more and more complex design rules on the layout of cells. The often non local nature of these new design rules is a great challenge not only for human designers but also for existing algorithms. We present a new flow for automatic cell layout generation that is able to deal with these challenges by globally optimizing several design objectives simultaneously. Our transistor placement algorithm not only minimizes the total cell area but at the same time guarantees the routability of the cell and finds a best arrangement and folding of the transistors. Our routing engine computes a detailed routing of all nets simultaneously. It computes a netlength optimal routing using a mixed integer programming formulation. Additional DFM constraints are added to this model to improve yield and reduce chip manufacturing costs.**

We present experimental results on current 7nm designs. Our approach allows to compute optimized layouts within a few minutes, even for large complex cells. The algorithms are used for the design of logic cells compatible with a published 7nm technology from a leading chip manufacturer where they meet manufacturability requirements and significantly reduced design turn around times.

***Index Terms*—automated cell generation, cell design, multiple patterning lithography, design for manufacturability**

I. Introduction

In a hierarchical design of a complex chip the *cells* are the functional units at the lowest level of the hierarchy. A cell realizes simple logical functionality as for example an AND-function, a buffer or a latch. These cells are used many times on a chip and therefore much effort is spent to find area optimized transistor level layouts. This reduces the overall chip area and thereby improves chip costs. A *standard cell library* is a collection of these cells that contains many implementations of the same logic function, differing in the number of inputs, power level, and timing behavior. The total number of cells contained in a standard cell library is in the range between 100 and 2,000 cells. Some applications require *custom cells*, i.e. cells not contained in the standard cell library. An example is the design of high-speed SRAM, where dynamic logic is used which cannot be mapped to standard logic gates.

Lacking high-quality automatic cell layout generators, these custom cells have so far been built manually by experienced designers. The design rules and DFM (design for manufacturability) requirements become increasingly complex with each new technology and the number of different cells used in modern designs is growing steadily. In addition, time-to-market pressure is increasing. The manual layout of a complex cell can take several days making this process a severe bottleneck in turnaround time, particularly due to increased automation in other design steps. This drives the need for high-quality automatic cell layout generators.

P. Cremer, S. Hougardy, and J. Silvanus are with the Research Institute for Discrete Mathematics, University of Bonn, Germany (e-mail: hougardy@or.uni-bonn.de).

T. Werner is with IBM Systems, Böblingen 71032, Germany (e-mail: tobias.werner@de.ibm.com).

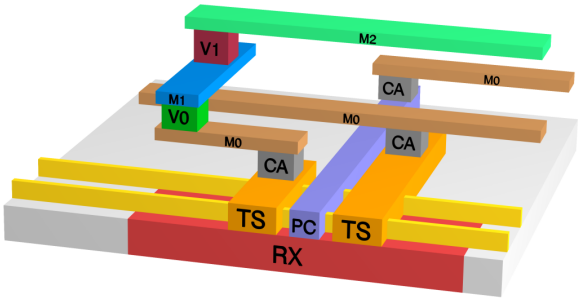
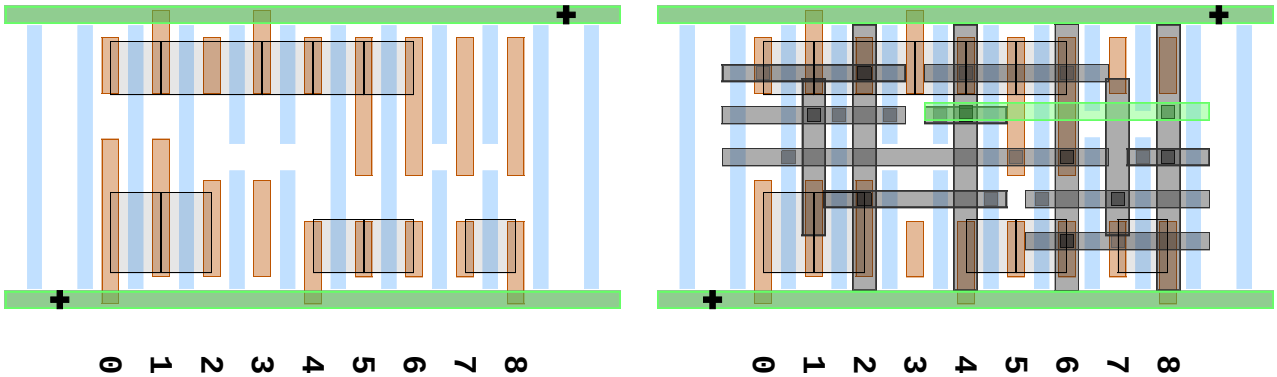


Fig. 1. Schematic view of a 7nm layout showing a single finFET and some wiring. Fins (yellow), M0, and M2 are horizontal while PC, TS, and M1 are vertical layers. The diffusion area is denoted by RX. The via layers are CA, V0, and V1.

In this paper, we present a new flow for the automatic generation of cell layouts, both for placement and routing. Our approach provides solutions that are provably optimal in terms of area consumption and guarantees routability. All cells produced by our algorithm are guaranteed to be LVS (Layout Vs Schematic) and DRC (Design Rule Check) clean. We have implemented cell level design rules consistent with [14] which describes Globalfoundries' proposed 7nm technology and consistent with the SADP trim shape process as described in [2], [12]. We consider all these design rules already during the placement and the routing phase. This is a crucial requirement for the current 7nm technology and beyond as design rule cleanness can no longer be achieved by local post processing operations alone. We also consider many DFM rules to optimize the yield.

In Globalfoundries' 7nm node all cell internal layers are uni-directional (see Fig. 1). The layers PC and TS are used to contact gates and source/drain contacts of the finFETs. A finFET can have more than one gate. We refer to the number of gates as the number of *fingers* of the transistor. A transistor is called *folded* if it has more than one finger. Three additional wiring layers M0, M1, and M2 are used that alternately have horizontal and vertical orientation. As M2 is mainly used for inter cell connections, it should be used for internal cell wiring only if necessary. The wiring layers are connected by via layers CA, V0, and V1, where CA connects both PC and TS with M0, V0 connects M0 with M1, and V1 connects M1 with M2. Different multiple patterning techniques are applied for these layers. SAQP (self aligned quadruple patterning) is used for the fins, SADP (self aligned double patterning) for the metal layers, and up to four times litho-edge for the via layers [26].

The *cell layout problem* can be described as follows. As



(a) Placement containing power bus (green), PC (blue), TS (brown), and FET boundaries (black). FETs are arranged in two horizontal stacks containing five FETs each.

(b) DRC clean routing with optimal manufacturability and wire length.

Fig. 2. Example results after (a) placement, (b) routing and DFM post processing.

input an image of the cell is given, i.e. an area with predefined horizontal power tracks at the top and bottom of the cell, equidistant vertical tracks for PC, TS, and M1, fin positions, and (not necessarily equidistant) horizontal tracks for M0 and M2. The finFETs, partitioned into p-FETs and n-FETs, have to be placed in two horizontal stacks in between the power tracks (see Fig. 2a). The electrical connectivity of the FETs and their sizes is described in a netlist. The task is to decide how many fingers a FET should use and to assign a location to each FET. Both choices are subject to the design rules and DFM constraints. Here, the width of the cell is the most important optimization criterion as this determines the area of the cell on the chip. Given a placement of FETs, the goal in routing is to find an embedding of rectilinear Steiner trees which realizes the given netlist. This has to be done meeting the design rules and DFM constraints, as well. As the overall goal in routing, we minimize weighted net length, with the topmost available layer M2 being more expensive than other layers. Other objectives (e.g. pin accessibility [25], number of vias, or electromigration reliability [11], [17], [26]) can be included as well.

A crucial point for the placement algorithm is to guarantee routability without making pessimistic assumptions. We achieve this by fully integrating the routing algorithm into our placement algorithm. This way we are able to find area minimum placements which are guaranteed to be routable. Among all such placements, our algorithm finds one that minimizes the estimated weighted netlength, or if more running time is allowed, it even guarantees to find an area optimum solution that minimizes the weighted netlength.

Simple sequential rip-up and reroute approaches turned out to fail for most of our placements. Moreover, these approaches do not allow to guarantee the routability of a given placement. Instead, we use an approach that allows us to route all nets *simultaneously* and consider all design rules already while building up the nets. The latter is required because only few design rule violations can be fixed after routing due to the limited space of our compact placements. Our router respects DFM constraints in the post processing phase. We also have successfully extended our approach to *multi-row* cells (see Section II-K).

A. Related Work

Most previous work on cell layout only focuses on subproblems or restricted versions of the general cell layout problem and is not directly applicable to 7nm layouts. Moreover, design rules and DFM requirements are more restrictive in 7nm than in previous 14nm/15nm [13] and 10nm finFET technology nodes.

Many different approaches have been suggested for the transistor placement problem. In [1] and [10] combinatorial algorithms are presented that optimize the cell area but assume a given transistor folding. The authors of [6] use an integer programming approach that optimizes cell area and includes transistor folding but they assume the possibility to pair n-FETs and p-FETs. In [19] a branch and bound approach is used for transistor placement that allows to optimize additional objectives, but transistor folding is not considered.

The placement algorithm has to consider the complicated dependencies between positions of n-FETs and p-FETs that arise due to the design rules of the SADP trim mask for the FEOL (front end of line) layers. Several approaches to handle SADP in automated design have been suggested, e.g. [24], [26]. Our approach differs in that we allow variable gate widths during trim mask generation and guarantee to find valid solutions if existent. Moreover, our placement algorithm guarantees routability by applying the routing algorithm while constructing a placement. To the best of our knowledge, our placement algorithm is the first to guarantee a routable placement without wasting cell area.

For intra cell routing many different approaches are known. Traditional channel routing [22] and simple rip-up and reroute strategies [15] fail in recent technologies. More successful are SAT-based [18] and the closely related integer-programming based [25] approaches. In these approaches, a set of candidate solutions is generated for each net. A SAT-formulation or an integer linear program is used to select one realization for each net so that all design rules are met.

Our routing approach is also based on an integer programming formulation. However, we do not need to pre-compute candidate solutions for each net but instead implicitly generate all possible routings for all nets simultaneously while packing

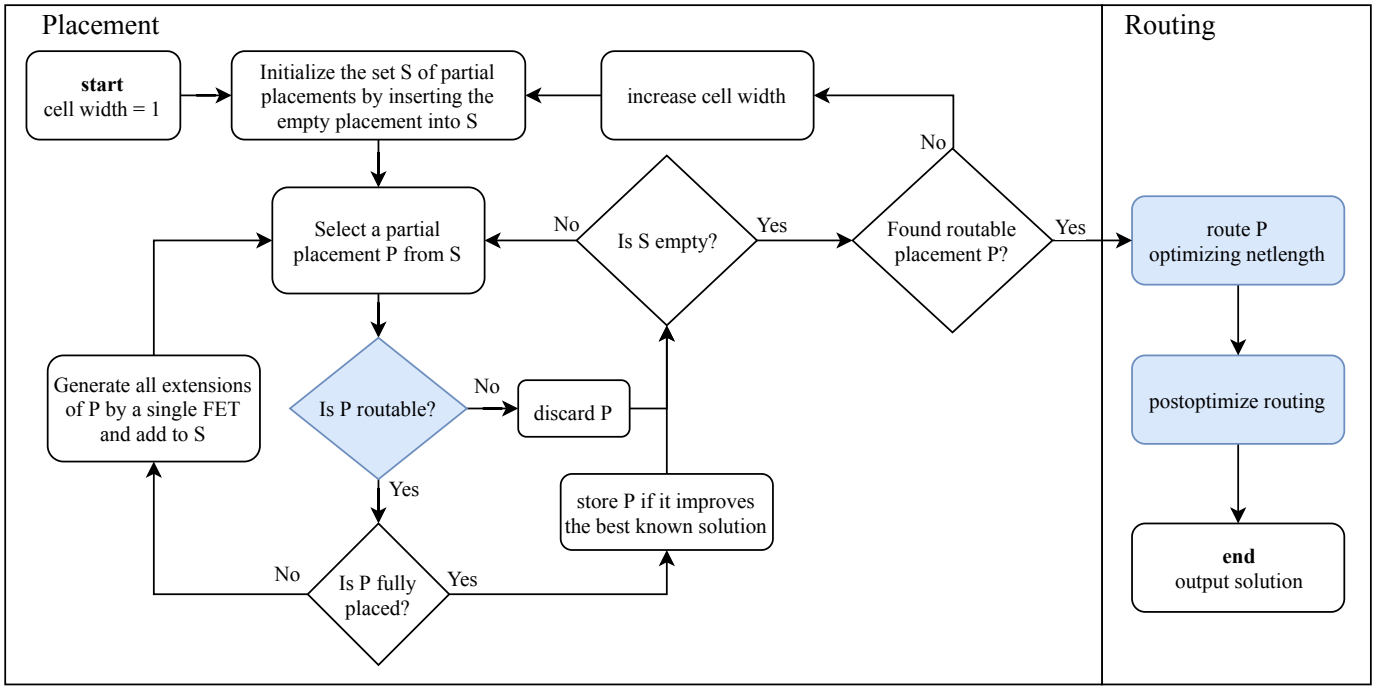


Fig. 3. Chart of our proposed new flow. Routing steps are in blue. Details of the branch and bound placement algorithm are given in Sections II-D to II-J. Note that the placement algorithm involves queries to a routability oracle which is a key feature of our new flow.

them. Using this approach we do not have to restrict the candidate solutions in advance (e.g. by restricting them to lie within some bounding box around the terminals) but are able to consider all possible routings. While this makes the search space much larger, our well chosen integer programming formulation turns out to be quickly solvable by standard MIP solvers. In [9] a conversion from an integer programming formulation to a SAT-formulation is used to speed up solution times.

Only few approaches exist so far that simultaneously solve the placement and routing problem for a cell. One such example is [8]. However, all these approaches have to simplify the placement and/or routing problem to reduce the search space complexity. Our approach does not require any simplifications and therefore is able to guarantee optimality of solutions without requiring too much running time. Typically, our approach is able to place and route cells with up to 15 FETs within a few minutes.

Previous work on BonnCell appeared in [3]. Compared to this earlier version, many improvements have been made. Most importantly, our placement algorithm now, for the first time, allows computing routable layouts minimizing total cell area.

Our new flow is depicted in Fig. 3, the details will be described in the following sections: In Section II, we discuss the placement algorithm, followed by the routing algorithm in Section III. Section IV reports the results of our implementation on cells at the 7nm technology node.

II. Placement

A. Problem Definition

The input of the cell placement problem is a set $\mathcal{F}$ of FETs, a set $\mathcal{N}$ of nets and a large number of technology-specific constraints. A *FET* is characterized by a tuple $(S_{\min}, S_{\max}, N_g, N_s, N_d, t, v)$, where

- $[S_{\min}, S_{\max}] \subseteq \mathbb{N}^2$ is the legal size interval of the FET measured in the number of fins intersected by the gates,
- N_g, N_s, and N_d are the nets connected to gate, source, and drain, respectively,
- $t \in \{\text{n-FET}, \text{p-FET}\}$ is the FET's type, and
- $v \in \mathbb{N}$ is its VT level.

A legal realization of a FET must intersect $S \in [S_{\min}, S_{\max}]$ fins. By using an interval for possible FET sizes, we allow optimizations over the size of the FET, if the design allows the flexibility. Otherwise, an interval containing a single FET size can be specified. We allow each FET to be folded, i.e. allow realizations with different numbers of *fingers*. Therefore, solving the placement problem does not only include the assignment of locations to each transistor but also deciding how large the FET should be exactly and how many fingers should be used. The total size S of a FET can be distributed to several fingers. Using only one finger, the FET is realized with one gate, intersecting S fins. Using a larger number of fingers, the FET is realized with several gates, located next to each other, which in total intersect S fins. If, for example, the size of a FET is 6 (measured in fins) it can be realized with 1, 2, 3, and 6 fingers, each covering 6, 3, 2, and 1 fin respectively. The number of fins intersected by a single finger is called the *height* of a FET. Depending on the used cell image, some FET heights can be forbidden, e.g. most images do not allow FET heights of 1 and there is also an upper bound on the allowed height. A FET realized with f fingers has f gates and $f + 1$ source and drain contacts. A FET with several fingers connects source and drain nets alternately. The placement algorithm is also allowed to swap FETs. In this case, the source and drain

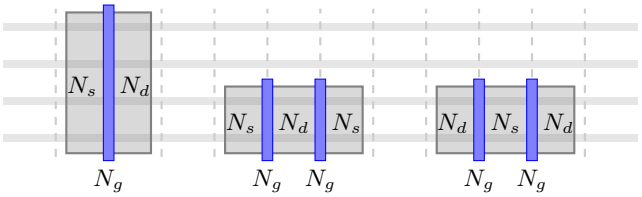


Fig. 4. A FET of size 4 realized with 1 finger, 2 fingers, and 2 fingers swapped. The heights are 4, 2, and 2 fins respectively. Gates are shown in blue, source and drain contacts in gray.

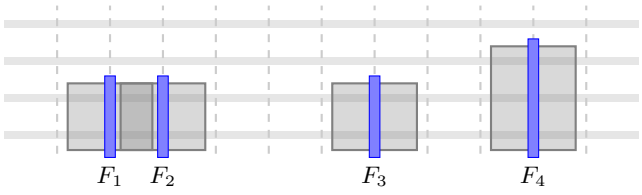


Fig. 5. Illustration of placement legality rule. FETs F_1 and F_2 have the same VT level, same height and same contact net facing each other. Thus they are allowed to share their diffusion regions. FETs F_3 and F_4 have different height and must therefore be separated by two empty PC tracks.

contacts of the FET exchange their places. Fig. 4 shows the same FET realized in three different ways. The transistors are arranged in two horizontal *stacks*, one next to each power rail. One stack consists of the cell's n-FETs and is placed directly next to the lower power rail, whereas the other stack contains the p-FETs and is placed directly next to the upper power rail.

Definition 1. *The configuration c of a FET is defined as the tuple (x, f, h, s) with*

- *location $x \in \mathbb{N}$ (measured in PC tracks),*
- *finger number $f \in \mathbb{N}_{>0}$,*
- *height $h \in \mathbb{N}_{>0}$ (measured in fin intersections per finger), and*
- *swap status $s \in \{true, false\}$.*

For $s = false$ the leftmost contact belongs to the source net, for $s = true$ it belongs to the drain net. Given a configuration c the terms $x(c)$, $f(c)$, $h(c)$, and $s(c)$ give the location, finger number, height, and swap status of c respectively.

Definition 2. *Given n FETs $F_1, \ldots, F_n$, a placement C is defined as a tuple $(c_1, \ldots, c_n)$, where c_i is a configuration for FET F_i.*

The output of the placement algorithm is a placement C and the routability guarantee (see Section II-F) of the placement. This information is then passed to the routing algorithm (see Section III).

B. Design Rules

There are many design rules which have to be obeyed. We split them into two sets, legality and routability.

Legality. The cell image guarantees that FETs do not overlap vertically. In horizontal direction FETs need to stay within the cell image which is given by the cell width W_{cell}. They also need to obey some minimum horizontal distance to each other depending on their configurations. This rule only applies to neighboring FETs. Two FETs of a placement are neighbors if there is no other FET in between.

Two neighboring FETs are allowed to share contacts if they have the same VT level, same height, and the overlapping source and drain contacts belong to the same net. More formally, this is captured by the following definition.

Definition 3. *Two neighboring FETs F_1, F_2 with configurations $c_1 = (x_1, f_1, h_1, s_1)$, $c_2 = (x_2, f_2, h_2, s_2)$ and $x_2 > x_1$ are allowed to share if*

- $h_1 = h_2$,
- $N_R(F_1, c_1) = N_L(F_2, c_2)$, *and*
- $VT(F_1) = VT(F_2)$,

where $N_L(F, c)$ denotes the leftmost net of FET F in configuration c. Similarly $N_R(F, c)$ denotes the rightmost net of FET F in configuration c. $VT(F)$ denotes the VT level of F. In this case the configuration is legal if $x_2 \geq x_1 + f_1$. Otherwise it is legal if $x_2 \geq x_1 + f_1 + 2$.

For sharing FETs the diffusion regions overlap and the contact is used simultaneously by both FETs. If sharing is not allowed, the FETs must be separated by at least two empty PC tracks. Fig. 5 gives an illustration of this rule. Placements that obey this rule are called *legal*.

Routability. A legal placement might still not be manufacturable. There are many more complicated rules. For example, all gates are manufactured with self aligned double patterning (SADP). In the first step, a regular pattern of unidirectional poly shapes is generated. In the second step, these shapes are cut off by a trim mask, leaving the desired gates. Not all legal placements admit a legal layer decomposition. Furthermore, the placement is useless if it is not routable on the metal layers. Therefore a full routability check needs to decide whether a placement is usable or not.

The placement legality rules will be obeyed by our placement algorithm by construction. Routability rules are checked by using the routing algorithm as described in Section III as a black box oracle. Only routable placements are returned by our algorithm.

C. Objective Function

We only consider placements which are routable under consideration of all LVS and DRC rules. We simply call these placements *routable*. There may be many routable placements for a given instance, some of which are more preferable than others. For a routable placement, we define its objective value as the tuple consisting of

1) cell width W_{cell},
2) weighted netlength estimation,

which is minimized lexicographically. More precisely, the algorithm returns a placement which respects all design rules and additionally globally minimizes the cell width W_{cell}. If there are several such placements the algorithm chooses a placement among them with the minimum weighted netlength estimate.

D. Placement Algorithm

Our algorithm is capable to realize multi-row cells. These cells occupy multiple circuit rows and have several pairs of stacks placed upon each other. For the moment, we focus on single-row instances with two stacks, more details on our multi-row implementation will be given in Section II-K.

The placement algorithm is based on a branch and bound approach. It consists of two parts, the first (Algorithm PLACECELL) iterates over the cell width in increasing order and calls Algorithm PLACEFIXEDCELLWIDTH for each fixed width. Algorithm PLACEFIXEDCELLWIDTH solves the placement problem and returns an optimum placement or that no routable placement exists with the given cell width. It proceeds by placing the FETs iteratively from left to right for both stacks simultaneously. After some FETs have already been placed, the next FET is chosen from the remaining FETs and placed to the right of the already placed FETs on this stack. For this FET all possible configurations (position, number of fingers, height, swap status) are tried. The resulting search tree is illustrated in Fig. 6. For some FET configurations, the resulting partial placement is illegal and discarded directly. The remaining placements are legal but might not be routable. Therefore, routability is checked for each of these placements by a call of the routing algorithm (Section III). Since we iterate over the cell width in increasing order, we know that the first found routable placement has minimum cell width. Netlength, the secondary objective, is optimized by comparing all routable placements with minimum width.

The algorithm presented above is very simple and can quickly be implemented. However, the resulting search tree becomes infeasibly large, even for small cells. In order to achieve practical running time, we develop several speed up techniques, presented in Sections II-E to II-J.

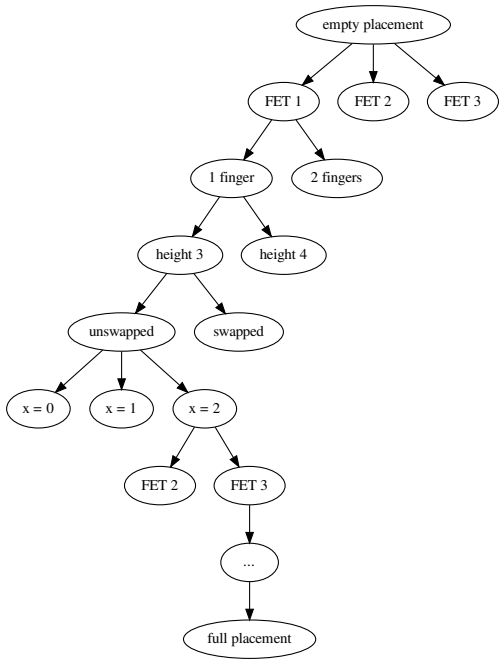


Fig. 6. Placement search tree. FETs are placed from left to right in all possible permutations and all configurations.

Algorithm 1 PLACECELL

Input:
$\mathcal{F}$ ▷ FETs to be placed
Output:
Routable placement with minimum width
1: **for** $W_{\text{cell}} := 1, 2, \ldots$ **do**
2: SETCELLWIDTH(W_{cell})
3: $P :=$ PLACEFIXEDCELLWIDTH($\mathcal{F}, \emptyset$)
4: **if** $P \neq$ null **then**
5: **return** P
6: **end if**
7: **end for**

E. Cell Width Pruning

The idea of pruning in any branch and bound algorithm is to remove infeasible or non-optimal nodes of the search tree without visiting them. For a given node, we want to detect situations in which all of its ancestors are either not routable or not optimal. In cell width pruning this means that given some partial placement with configurations for FETs $F_1, \ldots, F_k$, we prove that for any configuration of the remaining FETs the resulting placement is illegal. Note that at this point we are

Algorithm 2 PLACEFIXEDCELLWIDTH

Input:
$\mathcal{F}$ ▷ all FETs
$\mathcal{F}_p$ ▷ placed FETs with applied configuration
Output:
Routable placement with given width
or null if no such placement exists.
1: **if** $\mathcal{F}_p = \mathcal{F}$ **then** ▷ all FETs placed
2: $P :=$ CURRENTPLACEMENT($\mathcal{F}_p$)
3: **if** ISROUTABLE(P) **then**
4: **return** P
5: **else**
6: **return** null
7: **end if**
8: **end if**
9: $P_{\text{best}} :=$ null ▷ null indicates no placement
10: **for** $F \in \mathcal{F} \setminus \mathcal{F}_p$ **do**
11: **for** $c \in$ LegalConfigurations(F) **do**
12: ApplyConfiguration(F, c)
13: $P :=$ PLACEFIXEDCELLWIDTH($\mathcal{F}, \mathcal{F}_p \cup \{F\}$)
14: $P_{\text{best}} \leftarrow$ BEST(P_{best}, P)
▷ minimum w.r.t. objective value (see Section II-C)
15: **end for**
16: **end for**
17: **return** P_{best} ▷ might be null

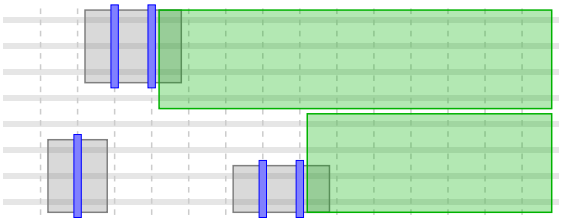


Fig. 7. Partial placement with three placed FETs. The remaining FETs can only be placed in the green area to the right of the placed FETs.

inside the cell width loop. Therefore, the cell width is already fixed, forcing all FETs to some limited area. Furthermore, since FETs are placed from left to right the only available space is to the right of all already placed FETs, see Fig. 7. We want to decide whether the remaining space suffices to place all remaining FETs. We run this step independently on both stacks since the FET sharing rules do not impose any constraints for FETs from different stacks. Since the FET sharing rules only apply to FETs which are direct neighbors, only the rightmost placed FET and the width of the remaining area are relevant. Note that our goal here is merely to prune subtrees which would violate the FET sharing rules. This by itself does not guarantee that the remaining placements will be routable which will be checked at a different point of the algorithm.

Our cell width pruning is based on Euler chains which have already been used in previous work on automated cell design [20]. As we will see, the Euler chains method requires restrictions on the search space of all possible configurations of yet to place FETs. We iterate over all possible restrictions and apply the method for each one of them. If there is no legal placement for any possible restriction, we know that there is no legal placement at all. If, on the other hand, we find a legal placement obeying some restrictions we keep the node in our search tree. These restrictions are defined as follows.

Definition 4. *A restriction r for a FET F is defined as a tuple (f, h, s), where*

- *f denotes the number of fingers,*
- *h the height, and*
- *s the swap status.*

A FET configuration $c = (x, f, h, s)$ obeys the restriction $r = (f', h', s')$, if

- *$f = f'$,*
- *$h = h'$, and*
- *f is even $\Rightarrow s = s'$.*

Definition 5. *A placement $C = (c_1, \ldots, c_n)$ obeys the restriction $R = (r_1, \ldots, r_n)$ if c_i obeys r_i for all i. The set of all legal placements obeying R is defined as $\mathcal{C}(R) := \{C : C \text{ legal, obeying } R\}$.*

Note that for an odd number of fingers, the swap status is not controlled by a restriction.

Algorithm CELLWIDTHPRUNING is presented below. The function RESTRICTIONS$(\mathcal{F})$ returns the set of all possible restrictions R, s.t. a placement C obeying R exists. The function MINWIDTH$(\mathcal{F}, C_\mathrm{L}, R)$ determines the minimum width of a placement of $\mathcal{F}$ obeying the restriction R with leftmost placed FET C_L. MINWIDTH uses a graph model and Euler walks. We will show how to implement MINWIDTH$(\mathcal{F}, C_\mathrm{L}, R)$ for $C_\mathrm{L} = \text{null}$. $C_\mathrm{L} = \text{null}$ means that no FET has been placed on this stack yet. This algorithm can then easily be extended for $C_\mathrm{L} \neq \text{null}$.

Algorithm 3 CELLWIDTHPRUNING

Input:
$\mathcal{F}$ ▷ FETs to be placed
C_L ▷ null or fixed leftmost FET
W_max ▷ Maximum allowed placement width
Output:
Does a placement with width at most W_max exist?

```
for R ∈ RESTRICTIONS(F) do
    if MINWIDTH(F, C_L, R) ≤ W_max then
        return true
    end if
end for
return false
```

In the following, we will develop the MINWIDTH algorithm and prove its correctness. We start with a formal definition of the width of a placement.

Definition 6. *The width of a placement $C = (c_1, \ldots, c_n)$ is defined as*

$$W(C) := \max_i (x(c_i) + f(c_i)) - \min_i x(c_i). \tag{1}$$

This allows us to formulate our central lemma.

Lemma 1. *Let R be a placement restriction. Then,*

$$\min_{C \in \mathcal{C}(R)} W(C) = \min_{C \in \mathcal{C}(R)} \left(\sum_{i=1}^{n} f(c_i) + 2N_{\textit{no-share}}(C) \right), \tag{2}$$

where $N_{\textit{no-share}}(C)$ is the number of neighboring FETs in C which are not allowed to share.

Proof. "$\geq$": For a given legal configuration C, assume w.l.o.g. that the indices are sorted s.t. $x(c_i) < x(c_j)$ for $i < j$. Let

$$G(c_i, c_{i+1}) := x(c_{i+1}) - x(c_i) - f(c_i) \tag{3}$$

be the gap between configurations c_i and c_{i+1}. Then, we have

$$\begin{aligned} W(C) &= x(c_n) + f(c_n) - x(c_1) && (4)\\ &= \sum_{i=1}^{n-1} (x(c_{i+1}) - x(c_i)) + f(c_n) && (5)\\ &= \sum_{i=1}^{n-1} (G(c_i, c_{i+1}) + f(c_i)) + f(c_n) && (6)\\ &= \sum_{i=1}^{n} f(c_i) + \sum_{i=1}^{n-1} G(c_i, c_{i+1}). && (7) \end{aligned}$$

If F_i and F_{i+1} are allowed to share we have $G(c_i, c_{i+1}) \geq 0$, otherwise $G(c_i, c_{i+1}) \geq 2$. This implies

$$W(C) \geq \sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C). \tag{8}$$

From Eq. (8) we immediately get

$$\min_{C\in\mathcal{C}(R)} W(C) \geq \min_{C\in\mathcal{C}(R)} \left(\sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C) \right). \quad (9)$$

"$\leq$": Equality in Eq. (8) is obtained if all neighboring FETs F_i, F_{i+1} have $G(c_i, c_{i+1}) = 0$ if they can share and $G(c_i, c_{i+1}) = 2$ otherwise. Such a placement can always be obtained from an existing placement C with $W(C) > \sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C)$ by moving FETs closer to each other.

Let C be a placement which minimizes

$$\sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C). \quad (10)$$

Then we can construct a placement C' from C for which $W(C') = \sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C)$. This yields

$$\min_{C\in\mathcal{C}(R)} \left(\sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C) \right) = W(C') \quad (11)$$

$$\geq \min_{C\in\mathcal{C}(R)} W(C), \quad (12)$$

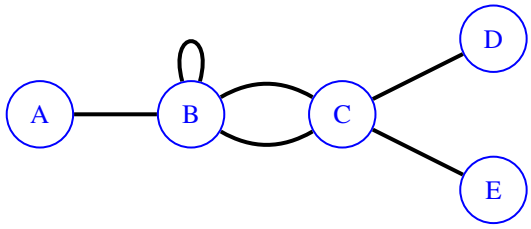
which concludes the proof. □

The left hand side of Eq. (2) is $\textsc{MinWidth}(\mathcal{F}, C_{\text{L}}, R)$ for $C_{\text{L}} = \text{null}$. Moreover, in Eq. (2), the sum $\sum_{i=1}^{n} f(c_i)$ does not depend on C but is equal for all C obeying R. Therefore, in order to determine $\min_{C\in\mathcal{C}(R)} W(C)$ we only need to determine $\min_{C\in\mathcal{C}(R)} 2N_{\text{no-share}}(C)$. This is where we use Euler chains.

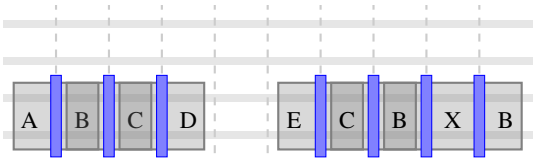
We sort the FETs which are to be placed into different groups. FETs within the same group can potentially share if placed next to each other. FETs from different groups are never allowed to share. This reduces the problem to independent groups.

FETs are only allowed to share if they have the same VT level and height. Therefore, we sort the FETs into groups with equal VT level and height. There needs to be a gap between two FETs of different groups. The number of gaps is minimized if all FETs within one group are placed directly next to each other. This leaves $N_{\text{group}} - 1$ gaps between the groups, where N_{group} is the number of groups.

Within each group, two FETs can share if the overlapping region belongs to the same net, see Fig. 5. For FETs with an even number of fingers the outward facing nets are equal on both sides. If the FET is unswapped it is the source net on both sides, and the drain net if it is swapped. For fixed swap status for all FETs with an even number of fingers, it is known for all FETs which nets belong to the outward contacts. Since we only minimize over placements obeying restriction R, we know which pairs of FETs are allowed to overlap. The remaining part of the placement configuration space which is not restricted is the order of FETs and the swap status of FETs with an odd number of fingers. In the following, we present a linear time algorithm based on Eulerian walk partitioning to find optimum FET orderings and swap states. A similar approach has already been used in [1] with additional analog-specific performance constraints. They assume that all devices have exactly one finger but the same algorithm can be used when the FETs obey restriction R.



(a) FET graph with 5 nodes corresponding to nets (A, B, C, D, E) and 5 edges, corresponding to FETs with an odd number of fingers and 1 loop corresponding to a FET with an even number of fingers.



(b) Placement with minimum number of gaps. This corresponds to the walks A, B, C, D, and E, C, B, B. Note that the net X does not appear as a node in the graph since it is the inner net of a FET with an even number of fingers.

Fig. 8. FET graph and placement corresponding to a minimum partition into walks.

We use a graph model to determine the minimum number of gaps that need to be left within one group. For each group, we construct a graph $G = (V, E)$, where V is the set of nets. For each FET with an odd number of fingers, we add an edge $e = \{v, w\}$ connecting the nodes corresponding to the source and drain net of the FET. For each FET with an even number of fingers, we add a loop $e = \{v, v\}$, where v is the net which belongs to the leftmost and rightmost contact of the FET. See the illustration in Fig. 8. The idea of this graph is that two FETs can share if their corresponding edges in the graph have a common vertex. Furthermore, a set of FETs can be placed next to each other without any gap if and only if there exists a walk in G consisting of the edges corresponding to the FETs. We call such a placement without gaps a *chain*.

Theorem 1. *Let F be a set of FETs with equal VT level, equal height, a fixed number of fingers, and fixed swap status for FETs with an even number of fingers. Let k be the number of walks of a minimum partition of the edges of $G(F)$ into walks. For any placement C, let $C_{|F}$ be the sub placement of C consisting of the FETs F. Then*

$$\min_{C\in\mathcal{C}(R)} N_{\textit{no-share}}(C_{|F}) = k - 1. \quad (13)$$

Proof. "$\leq$": Let $P_1, \ldots, P_k$ be a partition of the edges of G into walks. Then for each walk, we can place the FETs next to each other in a chain. Since there are k walks, this gives k chains with $k - 1$ gaps in between.

"$\geq$": Let C be a placement minimizing $N_{\text{no-share}}(C_{|F})$. By construction of the graph G, any chain corresponds to a walk in G. Therefore, this placements yields a partition of G into $N_{\text{no-share}}(C_{|F}) + 1$ walks. □

The size k of a minimum partition of a connected graph G into walks can be determined by the degrees of the vertices. Euler's well-known result states that for a connected graph a single walk suffices if no more than 2 vertices have odd degree. The following extension is also well-known:

Theorem 2. *Let G be a connected graph, N_{odd} the number of vertices with odd degree, and*

$$k := \max\left(\frac{N_{odd}}{2}, 1\right). \tag{14}$$

Then k is the size of a minimum partition of the edges of G into walks.

To obtain the size of a minimum partition into walks for a potentially not connected graph, one has to sum the number of walks for each connected component.

Using Theorem 1 and Theorem 2 we are able to compute the minimum number of gaps in a placement restricted by R in linear time. Since we want to know if the placement width is below some given threshold $W_{\max}$, we only have to deal with placements C for which $\sum_i f(c_i) \leq W_{\max}$. In practice, this means that we start with the minimum number of fingers of each FET and incrementally distribute additional fingers to the FETs. Instead of iterating over all swap states of the FETs with an even number of fingers, one can solve a certain VERTEX COVER problem, where the set of chosen vertices corresponds to the set of swapped FETs. While this does not improve the worst case running time, simple heuristics for VERTEX COVER reduce the running time in practice. Both techniques speed up calculating the pruning step significantly and lead to very fast running times in practice which are negligible to other parts of the algorithm, e.g. routability checks.

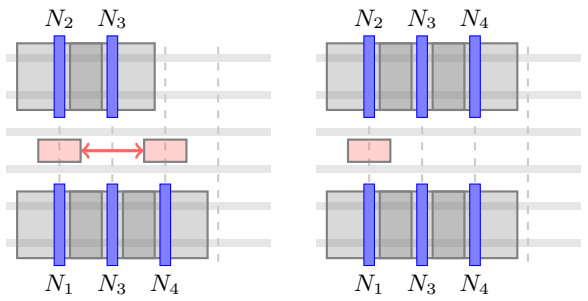
F. Routability Check

Legal placements are useless if they are not manufacturable or routable. Therefore, we directly check routability already during placement. This approach guarantees that the placement algorithm will return a routable solution. This is different from common approaches that only approximately model routability, e.g. by using a netlength-based objective function. Furthermore, our approach allows the user to define custom constraints, like blocking certain tracks or fixing the position of routing pins. These constraints ensure a seamless embedding of the cell into the hierarchical context. We let our routing engine decide about the routability of a placement. Since the routing engine is able to deal with custom constraints, the placement will automatically adapt to them as well.

In its simplest version, the routability check is run at the leaves of the search tree and discards all nodes which fail. Again, the simple approach is too slow in practice and several speedup techniques are used.

G. Partial Placement Routability Check

Most legal placements are not manufacturable or not routable. It is therefore essential to detect partial placements which cannot be completed to a routable placement as early as possible. The key difficulty here is the non-monotonicity of partial placements w.r.t. to routability. Counter-intuitively, a partial placement can be unroutable but the same partial placement with an additional placed FET is routable. The same is true for the FET distance rules. Fig. 9 shows an example where the addition of a FET to an illegal partial placement yields a legal placement.



(a) Illegal placement with 5 FETs.

(b) Legal placement with 6 FETs, which is an extension of the illegal placement on the left.

Fig. 9. Adding a FET to an illegal partial placement can make it legal. Gate nets N_1 and N_2 are different and need to be separated by a trim shape (red). (a) Gate net N_4 has no FET placed opposite and therefore also needs to be separated from the other stack by a trim shape. These trim shapes are too close to each other and make the placement illegal. (b) Adding a FET with gate net N_4 removes the need for the second trim shape and legalizes the situation.

Therefore it is not possible to use a partial placement as input to our routing engine as if it was a full placement to determine routability of its ancestors. The problem is solved by assigning one of three states to every position in the partial placement. The states are

- FET placed with configuration c
- empty (no FET will be placed here in the future)
- unknown (potentially empty, but some FET might be placed here in the future)

The routing formulation only adds constraints for cases where the presence or absence of a FET is known. If the presence of a FET is unknown, no constraints are added. For the example in Fig. 9-(a), no constraints are added which would enforce the existence of the right trim shape.

H. FEOL, FET Access, and Full Routing

Due to the large complexity of the routing problem, routing oracle calls are very expensive. In many cases (partial) placements are illegal even when only a restricted set of rules is considered. We distinguish between three sets of rules, called *phases*:

1) *FEOL* (front end of line). Contains only rules below M0. This includes floating gates, RX coloring, fin trim shapes, and PC trim shapes among others.
2) *FET-access*. Contains rules up to M0. Some connections of FETs can be routed below M0 mainly by sharing with other FETs. For all other connections, we know that all their terminals must be connected to M0. In this phase, we enforce that all terminals of these nets are connected to M0. Placements which are illegal w.r.t. FET-access constraints usually have a highly congested region with

many terminals of different nets. All M0 trim shape rules and full via coloring are contained in this group. Also contains all FEOL rules.

3) *Full*. The entire routing. Also honoring net connections to user-specified pin tracks or respecting forbidden tracks. Full routability can only be checked for full placements, i.e. leaves of the search tree.

If a placement is illegal w.r.t. FEOL rules, we know that it is also illegal w.r.t. to FET-access or full, since the FEOL rules are a subset of the rules of the other phases. Similarly a placement illegal w.r.t. FET-access is also illegal w.r.t. full routing. The phases are run one after the other. We stop as soon as one phase proves that the placement is illegal or all phases are legal. Due to the large difference in complexity of the phases, their average running time also differs by orders of magnitude. An FEOL oracle call lasts about 0.1 seconds, FET-access running time is in the order of seconds and full routing can take minutes to hours on large instances. Since many (partial) placements are already illegal w.r.t. FEOL rules, the phase-based approach is much faster compared to directly checking full routability.

The second major advantage is that FEOL and FET-access routability can also be checked for partial placements which is not possible for full routability. This is essential to prune partial placement nodes of the search tree which violate some DRC constraint without extending them to all possible full placements.

I. FEOL Routing Oracle Cache

In contrast to the previously presented techniques, routing oracle caching does not affect the search tree. The running time of the routing oracle dominates the total placement running time. Many partial placements look similar from a routability perspective and calling the oracle twice for similar instances can be avoided. This is especially the case for FEOL routing of partial placements. In FEOL routing only the lower layers are considered which means that some details of the partial placement do not matter for FEOL routability. Two instances are equivalent routing instances for example if they can be transformed into each other by a permutation of nets. They still originate from different placements, but their routing problems are either both routable or both unroutable. In these cases, a single routing oracle call suffices. To detect these situations, we transform partial placements into a data structure called *routability data* which contains all information needed by the FEOL routing oracle but hides other information contained in the placement, for example net names. If two partial placements are transformed into the same routability data we know, by construction, that the FEOL routability oracle will give the same answer for both. This is exploited by storing all queries to the routing oracle in a cache.

In practice, we observe that on average 9 out of 10 queries to the FEOL routing oracle, can be answered by a cache lookup. This results in a factor 10 speedup.

J. Netlength Pruning

Once a routable placement has been found, the algorithm continues to search for placements with better objective value. Since the cell width is fixed at this point, the determining criterion is netlength. We use the bounding box netlength model which can also be applied to partial placements. Additionally, we calculate lower bounds for the netlength of partial placements, also considering the FETs which still have to be placed. In this way, we are able to prune parts of the search tree which cannot contain routable solutions which are better than the one we have already found.

K. Multi-Row Cells

Very large cells are typically not implemented on one but several neighboring circuit rows. Such *multi-row* cells can also be placed with our algorithm. To place a cell on several circuit rows, we first compute an assignment of FETs to the rows using a mixed integer programming approach. Assignments are evaluated by their number of row-crossing connections and an estimation for the total cell width. Using Algorithm PLACE-CELL (Section II-D), the rows are placed one after the other with each new placement respecting constraints enforced by already placed rows.

As the assignment problem of FETs to rows does not model the resulting total cell width exactly, we cannot guarantee global optimality for the placement of multi-row cells. However, each individual row has minimum possible width, subject to the routing constraints implied by the previously placed rows.

III. ROUTING

As described in Section II, the routing engine is not only used to compute a routing after the cell has been placed, but instead also called many times during the placement algorithm to ensure that the computed placement is routable. In the following, we explain the details of the full router that runs after the placement algorithm. Required modifications for usage as routing oracle during the placement phase are given in Section III-G.

In order to solve the routing problem on a placed cell, we use a mixed integer programming (MIP) approach. Next to the input already given for the placement problem (Section II), the cell routing problem expects the location of each FET from the placement.

Our goal is to find a valid routing that minimizes the wire length and number of vias in order to optimize the power, timing, and yield properties of the cell. Note that this does not impose an algorithmic limitation, as our routing engine allows us to optimize arbitrary linear objective functions.

In the remainder of this section, we first describe the grid graph in which we search a disjoint minimum-cost Steiner tree packing. Then, we give the description of our core MIP model for the Steiner tree packing problem, and explain how design rules, in particular trim shape rules and via coloring rules, are incorporated into that model. Finally, we discuss required modifications and a post processing routine that optimizes the routing with respect to DFM (design for manufacturability).

A. Grid Graph Construction

Since each wiring layer only allows either vertical or horizontal wires, we represent the cell routing space by a three-dimensional grid graph $G = (V, E)$ with edge costs. For each layer, we are given a set of routing tracks specifying feasible positions for wires which are not necessarily equidistant.

By intersecting routing tracks on adjacent layers, we obtain the vertex set V. The edge set E consists both of line segments connecting adjacent intersections on the same layer as well as vias between stacked vertices on adjacent layers. Each edge $e \in E$ represents the center line of a possible wire shape. Trim shapes allow using edges partially, cf. Section III-E.

Edge costs are obtained by multiplying their geometric length by a layer-, track-, and net-dependent value. This allows to trade off wire length against the number of vias and to leave more space for inter-cell routing by increasing certain edge costs. For example, on M1, only every second track is usable for inter-cell routing due to the power via pattern, and M2 is widely used for inter-cell routing.

B. Mixed Integer Programming Formulation

First, we describe the core MIP we use to model the Steiner tree packing problem in graphs. Then, in Sections III-D to III-F, we explain how design rules are incorporated into the model.

For each net $k \in \mathcal{N}$ and each edge $e \in E$, we add a binary variable x_e^k specifying whether edge e is used by net k. Furthermore, for each edge $e \in E$, we introduce a binary variable x_e that determines whether edge e is used by some net, and add the constraint $x_e = \sum_{k \in \mathcal{N}} x_e^k$. Since x_e is upper bounded by 1, this constraint already guarantees edge disjointness of integral solutions.

In the following, for some vertex set $X \subset V$, we refer by $\delta(X)$ to the set of edges between X and $V \setminus X$, and, in the directed case, by $\delta^+(X)$ to the set of edges leaving X and by $\delta^-(X)$ to the set of edges entering X.

We ensure connectivity by adding for each net $k \in \mathcal{N}$ a formulation of the Steiner tree problem in graphs to the model. Note that using a formulation with a strong relaxation is essential for small running times. In [5], the undirected cut relaxation is used for that purpose: For each net $k \in \mathcal{N}$, we denote by $T_k \subseteq V$ the set of its terminals. We say that a cut $\delta(X)$ *separates* T_k if both $T_k \cap X$ and $T_k \setminus X$ are nonempty. Then, for each cut separating the terminal set, the undirected cut relaxation requires at least one edge of the cut to be contained in the Steiner tree, i.e. $\sum_{e \in \delta(X)} x_e^k \geq 1$. However, the undirected cut relaxation has an integrality gap of 2 [4], which is already asymptotically attained in the special case that G is a circuit, even if all vertices are terminals, as the fractional solution $x^k \equiv \frac{1}{2}$ demonstrates.

One can strengthen this relaxation by using a bidirected auxiliary graph (V, A) with $A = \{(i, j) : \{i, j\} \in E\}$ which contains two opposing edges (i, j) and (j, i) for each original edge $\{i, j\} \in E$. Choose an arbitrary root terminal $r_k \in T_k$ and add usage variables $\vec{x}_{ij}^k$ for all directed edges $(i, j) \in A$. Then, for each cut $\delta^+(X) \subset A$ with $r_k \in X$ and $T_k \setminus X$ nonempty, require that at least one edge leaving X is used, i.e. $\sum_{(i,j) \in \delta^+(X)} \vec{x}_{ij}^k \geq 1$. Finally, lower bound the usage of each original edge $\{i, j\} \in E$ by the *sum* of the usages of both directed edges (i, j) and (j, i). This relaxation is called bidirected cut relaxation. The integrality gap of the bidirected cut relaxation is unknown, the largest known lower bound is $6/5$ [21], and no upper bound stronger than 2, which is implied by the integrality gap of the undirected cut relaxation, is known.

By introducing additional flow variables, one can eliminate the exponential number of cut constraints, resulting in the multicommodity flow relaxation, first introduced in [23]. This relaxation is equivalent to the bidirected cut relaxation [16] and was already used in [7] to solve Steiner tree packing problems. We will also use the multicommodity flow relaxation:

For each net k, we denote the set of sink terminals $T_k \setminus \{r_k\}$ by S_k. Then, the multicommodity flow relaxation introduces a commodity for each sink $s \in S_k$ and requires a flow of one unit of the commodity from r_k to s to be supported by $\vec{x}^k$. More precisely, for each net $k \in \mathcal{N}$, sink $s \in S_k$ and directed edge $(i, j) \in A$, a flow variable f_{ij}^{ks} that is upper bounded by $\vec{x}_{ij}^k$ is introduced, representing the flow of the commodity s of net k along the directed edge (i, j). Then, we add flow conservation constraints at vertices in $V \setminus \{s, r_k\}$ and enforce that r_k sends one unit of flow and that s receives one unit of flow of commodity s.

Finally, to ensure vertex disjointness, for each net $k \in \mathcal{N}$ and vertex $v \in V$, we add a binary vertex usage variable x_v^k, which upper bounds usage variables of incident edges, and add the constraint that each vertex may be used by at most one net.

The complete model is given in Fig. 10, where we denote by $b^{ks}(v) := \sum_{(i,j) \in \delta^+(v)} f_{ij}^{ks} - \sum_{(i,j) \in \delta^-(v)} f_{ij}^{ks}$ the flow balance of commodity s of net k at a vertex $v \in V$.

$$
\begin{aligned}
& & \min \quad & \sum_{e \in E} c_e x_e & & \\
\text{s.t.} \quad & x_e & = \quad & \sum_{k \in \mathcal{N}} x_e^k & & \forall\, e \in E \\
& x_e & \in \quad & \{0, 1\} & & \forall\, e \in E \\
& x_e^k & \in \quad & \{0, 1\} & & \forall\, e \in E,\, k \in \mathcal{N} \\
& b^{ks}(v) & = \quad & \begin{cases} 1 & \text{if } v = r_k \\ -1 & \text{if } v = s \\ 0 & \text{otherwise} \end{cases} & & \\
& & & & & \forall\, v \in V,\, k \in \mathcal{N},\, s \in S_k \\
0 \quad \leq & f_{ij}^{ks} & \leq \quad & \vec{x}_{ij}^k & & \forall\, (i, j) \in A,\, k \in \mathcal{N},\, s \in S_k \\
& \vec{x}_{ij}^k + \vec{x}_{ji}^k & \leq \quad & x_{\{i,j\}}^k & & \forall\, \{i, j\} \in E,\, k \in \mathcal{N} \\
& x_v^k & \in \quad & \{0, 1\} & & \forall\, v \in V,\, k \in \mathcal{N} \\
& x_e^k & \leq \quad & x_v^k & & \forall\, v \in e \in E,\, k \in \mathcal{N} \\
& \sum_{k \in \mathcal{N}} x_v^k & \leq \quad & 1 & & \forall\, v \in V
\end{aligned}
$$

Fig. 10. Core mixed integer program modeling a Steiner tree packing problem.

In this basic formulation, no additional constraints, especially with respect to distances between shapes, are taken into consideration.

C. Conditional Constraints

In order to implement complex design rules, we need to model logical implications to conditionally enable constraints. More specifically, consider a linear inequality of the form $\sum_{i\in I} a_i x_i \leq b$, and let x_{cond} be a binary variable. We want to model

$$(x_{\text{cond}} = 0) \implies \left(\sum_{i\in I} a_i x_i \leq b\right). \tag{15}$$

To this end, we use the following standard approach: Let U be a constant that is an upper bound on $\sum_{i\in I} a_i x_i - b$, which can be derived from the variable bounds. For this, we require that all involved variables x_i have finite variable bounds, which is the case in our application. Then, the constraint

$$\sum_{i\in I} a_i x_i - U x_{\text{cond}} \leq b \tag{16}$$

satisfies our needs: If $x_{\text{cond}} = 0$, then the constraint equals the original constraint, and if $x_{\text{cond}} = 1$, then $\sum_{i\in I} a_i x_i - U \leq b$ is always satisfied by the choice of U.

Of course, a similar approach works in order to condition on $x_{\text{cond}} = 1$. By replacing constraints with equality by two inequalities, the same approach can be applied to these constraints as well. Finally, in case we need to condition on multiple such binary conditions, we can recursively apply the procedure above.

D. Mapping DRC Constraints

For the wiring within a cell, design rules used to fall into the two basic categories of *same-net rules* and *diff-net rules*. Same-net rules are in place to avoid specific geometric configurations of wiring shapes of a single net, while diff-net rules require a certain minimum distance between wires that belong to different nets. However, in 7nm technology, all wires on layers used for cell-internal routing are generated as the complement of trim shapes, which are not associated with any particular net, and constraints on wire shapes are entirely expressed in terms of constraints on trim shapes. Hence, the routing model contains additional variables and constraints that model the trim shape configuration, and the only additional constraints on non-via edge usage variables are consistency constraints with the trim shape model. A detailed description of the trim mask and its manufacturing process is given in [2] and [12].

All features are manufactured using multiple masks in order to increase packing density: Shapes on different masks are allowed to have a smaller distance than shapes on the same mask. Hence, a valid routing does not only consist of a disjoint Steiner tree packing, but also requires features on such layers to be assigned to masks such that certain design rules are met. We call this assignment *coloring*. In the 7nm node, this only affects vias, since wire and trim shapes use a fixed predefined coloring scheme.

E. Trim Shape Model

Recall that on each routing layer, the routing grid graph consists of parallel routing tracks. Except on the layer TS, each routing track is associated with a fixed color, representing the mask that is used to manufacture trim shapes on this track. Since trim shapes on tracks of different color are independent, we do not consider colors in the remainder of this section. The full model is then obtained by applying the following, for each layer and color, to all tracks of that color.

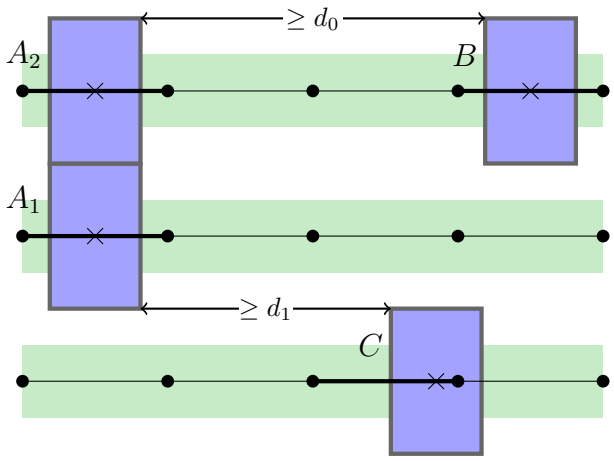


Fig. 11. A trim shape configuration with relevant trim spacing distances, resulting wires and the assignment of trim shapes to grid graph edges. Each trim shape is assigned to an edge containing its center.

Now, fix a layer and without loss of generality assume that routing tracks on that layer are horizontal. Figure 11 shows a configuration with four trim shapes A_1, A_2, B and C. Note that two trim shapes A_1 and A_2 will result in a single trim shape A during manufacturing. Trim shapes on the same track must satisfy a certain minimum horizontal distance, indicated by d_0, while trim shapes on neighboring tracks must either align to the same coordinate (as in case of A) or again satisfy a minimum horizontal distance, indicated by d_1. Note that the minimum same track trim shape spacing rule via d_0 also encodes a minimum area constraint on the wire in between. Since trim shapes have a fixed width W_{trim}, any trim shape configuration is uniquely represented by the set of their center coordinates, indicated by crosses.

Then, we can assign each trim shape to an edge that contains the trim shape's center. This assignment is illustrated in Figure 11, where edges that have a trim shape assigned to them are highlighted. Since d_0 is sufficiently large, at most one trim shape can be assigned to any edge.

Hence, we can model a trim shape configuration as follows: For each edge e, we add a binary variable t_e^{active} which specifies whether there is a trim shape with its center on e, and an integral variable t_e^{pos} that specifies the exact x-coordinate of the trim shape's center in case there is one, where

$$\text{xmin}(e) \leq t_e^{\text{pos}} \leq \text{xmax}(e). \tag{17}$$

The distance constraints on trim shapes are then modeled as follows. Let e and f be two edges on the same track with $\text{xmax}(e) \leq \text{xmin}(f)$. There are three possible cases: If $\text{xmax}(f) - \text{xmin}(e) - W_{\text{trim}} < d_0$, i.e., there is no feasible trim shape configuration with both a trim shape on e and on

f, then we add the constraint

$$t_e^{\text{active}} + t_f^{\text{active}} \leq 1, \tag{18}$$

modeling that at most one of the two trim shapes may be active. Otherwise, if $\text{xmin}(f) - \text{xmax}(e) - W_{\text{trim}} < d_0$, i.e. there are both feasible and infeasible trim shape configurations with trim shapes on e and f, we add the constraint

$$\left(t_e^{\text{active}} = 1 \wedge t_f^{\text{active}} = 1\right) \tag{19}$$

$$\implies \left(t_f^{\text{pos}} - t_e^{\text{pos}} - W_{\text{trim}} \geq d_0\right), \tag{20}$$

which guarantees that trim shapes on e and f are sufficiently far away from each other if present.

Distance constraints on pairs of edges e, f on neighboring tracks are modeled similarly if the x-intervals e and f are disjoint. Otherwise, the only valid configuration with a trim shape on both e and f requires these to be aligned, which we model as

$$\left(t_e^{\text{active}} = 1 \wedge t_f^{\text{active}} = 1\right) \implies \left(t_f^{\text{pos}} = t_e^{\text{pos}}\right). \tag{21}$$

Finally, we need to ensure consistency of the edge usage model and the trim shape model. Used edges may not contain a trim shape, so for each edge e, we add the constraint

$$t_e^{\text{active}} + x_e \leq 1. \tag{22}$$

Furthermore, let e be an edge and $k \in \mathcal{N}$ be a net. If e is used by net k, then adjacent edges must also be used by net k unless cut off by a trim shape. Hence, if f is an edge adjacent to e, add the constraint

$$\left(x_e^k = 1\right) \implies \left(t_f^{\text{active}} + x_f^k \geq 1\right). \tag{23}$$

F. Vias

On via layers, we need to assign colors to used edges. For each via $e \in E$, let M_e be the set of possible colors for e. Then, for each via edge $e \in E$, net $k \in \mathcal{N}$ and color $m \in M_e$, we add a binary variable ${}^m x_e^k$ and enforce

$$x_e^k = \sum_{m \in M_e} {}^m x_e^k. \tag{24}$$

Moreover, for each such edge e and color m, we add a binary variable ${}^m x_e = \sum_{k \in \mathcal{N}} {}^m x_e^k$, representing whether edge e is used with color m by any net.

Then, if two close via edges e, f are not allowed to be used by the same color m, add the constraint

$${}^m x_e + {}^m x_f \leq 1. \tag{25}$$

Similarly, if two via edges e, f are even too close to be used by different colors, require

$$x_e + x_f \leq 1. \tag{26}$$

Minimum required spacings between vias and trim shapes are implemented analogously to trim-trim-spacings.

G. Routing Oracle During Placement

As discussed in Sections II-G and II-H, the routing engine is queried during the placement algorithm to prune partial placements that cannot be completed to routable placements. These queries are not performed using the full routing model, but instead either use the *FEOL* (front end of line) or *FET-access* phase. These phases only respect design rules below M0 or up to M0, respectively.

Instead of forcing net connectivity using flow variables, for each FET contact, it is determined whether a connection to M0 is required, and in that case, constraints enforcing such a connection are added. This results in a much simpler model which we can afford to solve many times during placement, and, albeit its limited set of constraints, detects many unroutable placements.

When routing partial placements, we determine the *unplaced area* where future FETs might be placed. Then, we only add constraints for placed FETs, and skip all constraints that depend on the presence of a FET in the unplaced area.

H. Post Processing

Design for manufacturability (DFM) rules are soft constraints that are not strictly required and aim at increasing yield by avoiding configurations with a higher failure risk. DFM rules include increased trim-via and trim-trim spacings, preferred coordinates for trim shape locations and preferred via colors at specific locations.

After computing a full routing, we perform a post processing which aims at satisfying as many DFM rules as possible while not increasing wire length and via count significantly.

For each DFM rule and each location, we add a binary variable that determines whether the rule is satisfied at that location. Then, we add a constraint modeling the DFM rule, conditioned on that variable, and add the binary variable to the objective function, rewarding all satisfied DFM rules. Finally, we restrict all flow variables (cf. Section III-B) in the model to their current value, which fixes the structure of the routing solution, and reoptimize.

IV. Experimental Results

We have implemented all algorithms described in this paper and tested them on real-world 7nm instances. We did all experiments single-threaded on a 2.20GHz Intel Xeon E5-2699 v4 machine using CPLEX 12.7.1 as MIP-solver.

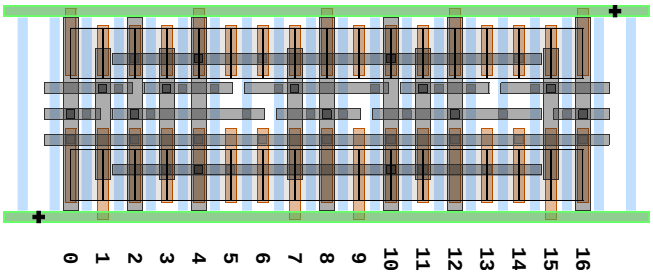
A. Results for a Standard Cell Library

As a first test set, we chose a standard cell library containing 120 cells. See Table I for details on this library. The number of FETs for these cells is between 2 (for the inverters) and 8.

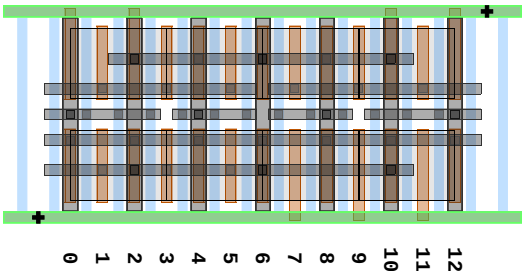
The cells in this standard library have been designed manually and much time has been spent to get a best possible layout. All layouts found by BonnCell were at least as good as the known layouts. For 10 of the 120 cells, BonnCell was able to find better (with respect to area) layouts. In some cases, the BonnCell result improved the library results by more than 22%. Figure 12 shows an example of such a cell. The average running time needed per cell was less than 2 minutes. Table I shows the detailed results of BonnCell on these cells.

TABLE I
BONNCELL RESULTS ON CELLS FROM A STANDARD LIBRARY. WITHIN EACH ROW OF THE TABLE FOR EACH CELL TYPE (COLUMN 2) WE SHOW THE NUMBER OF VARIANTS OF THIS CELL IN THE LIBRARY (COLUMN 1), THE NUMBER OF FETS AND NETS IN THE CELL (COLUMNS 3 AND 4) AND THE AVERAGE RUNNING TIME IN SECONDS NEEDED BY BONNCELL FOR ALL CELLS WITHIN A ROW FOR THE PLACEMENT STEP (COLUMN 5) AND THE ROUTING STEP (COLUMN 6).

cells	cell type	FETs	nets	avg. time placement [s]	avg. time routing [s]
11	AOI21	6	9	48	51
11	AOI22	8	11	61	62
17	INV	2	5	8	9
16	NAND2	4	7	46	53
11	NAND3	6	9	19	24
5	NAND4	8	11	37	59
16	NOR2	4	7	40	50
8	NOR3	6	9	37	38
5	NOR4	8	11	33	50
10	OAI21	6	9	30	32
10	OAI22	8	11	143	40



(a) Layout from the standard library.



(b) BonnCell's layout.

Fig. 12. Layouts of an OAI22 standard library cell. BonnCell's layout reduces the cell's area by more than 22% and total net length on metal layers by more than 33%.

B. Results for Latches

Latches are much larger and more complex cells. Manually finding good layouts for these cells is even for very experienced designers a challenging task. Moreover, it may take a designer several days to find a good layout. We used a testbed of 22 latches ranging in size between 10 and 44 FETs. For 7 of these 22 latches, BonnCell found a provably area optimal solution. For the other latches, BonnCell ran into a timeout and the solution found is therefore not guaranteed to be area optimal. Nevertheless, BonnCell found in all but one case a solution that was either better in area compared to the designer's solution or needed less M2 tracks. The one case in which BonnCell was only able to find an equally good solution is a latch where the designer's solution needs no M2 and is area optimal and therefore cannot be improved. Table II shows the details of BonnCell's results on this latch testbed. An example layout for a latch that BonnCell built using 5% less area than the designer's solution is shown in Fig. 13. Using the approach described in Section II-K BonnCell can also build multi-row layouts of cells. As an example we show in Fig. 14 a 2-row layout for the same latch.

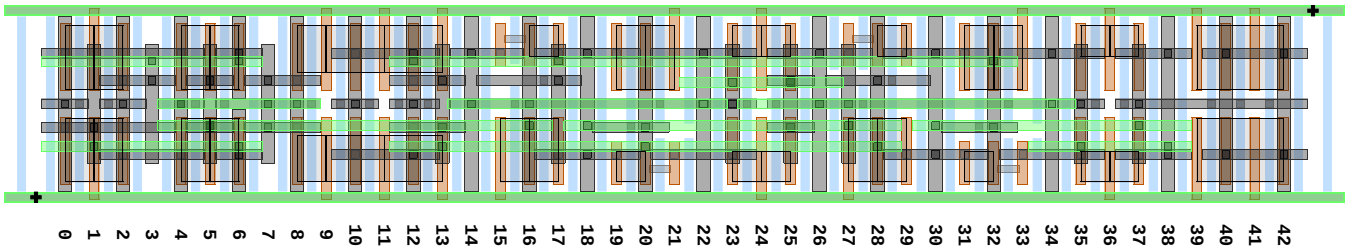
TABLE II
BONNCELL RESULTS ON LATCHES. WITHIN EACH ROW OF THE TABLE WE SHOW THE LATCH NAME (COLUMN 1), THE NUMBER OF FETS AND NETS IN THE LATCH (COLUMNS 2 AND 3), THE WIDTH OF THE SOLUTION FOUND BY BONNCELL IN TRACKS (COLUMN 4) AND BONNCELL'S TOTAL RUNNING TIME FOR PLACEMENT AND ROUTING IN SECONDS (COLUMNS 5 AND 6) AND IN COLUMN 7 WHETHER IT IMPROVED THE BEST DESIGNER'S LAYOUT (YES) OR WAS EQUALLY GOOD (EQUAL).

latch	FETs	nets	tracks	running time [s] place	running time [s] route	improved area or M2
DFFFQ X1	38	27	32	7227	2730	yes
DFFQDICE X1	43	29	38	3243	2756	yes
DFFQ X1	28	21	20	3018	1333	yes
ELATN X1	12	11	12	1218	54	yes
ELATS X1	10	11	10	544	504	yes
ELAT X1	12	11	12	1171	49	yes
ELAT X3	12	11	16	2706	34	yes
ELAT X8	12	11	30	3236	102	yes
ESLATN X1	32	25	36	3237	3492	yes
ESLATS X1	26	25	32	3625	592	yes
ESLAT X1	32	25	36	3623	2140	yes
ESLAT X3	32	25	36	3656	594	yes
L1LATF X1	26	21	22	3966	189	equal
N1LAT X1	38	27	38	3238	4725	yes
N1LAT X3	38	27	44	3239	1247	yes
SDFFQN X1	36	28	28	3636	5444	yes
SDFFQS X1	32	27	24	28837	2793	yes
SDFFQ X1	36	28	28	3644	4016	yes
SDFFQ X3	36	28	42	3650	701	yes
SDFFSRPQ X1	44	34	36	3239	1695	yes
INV ELAT X1	14	12	16	3606	51	yes
INV ELAT X3	14	12	20	603	51	yes

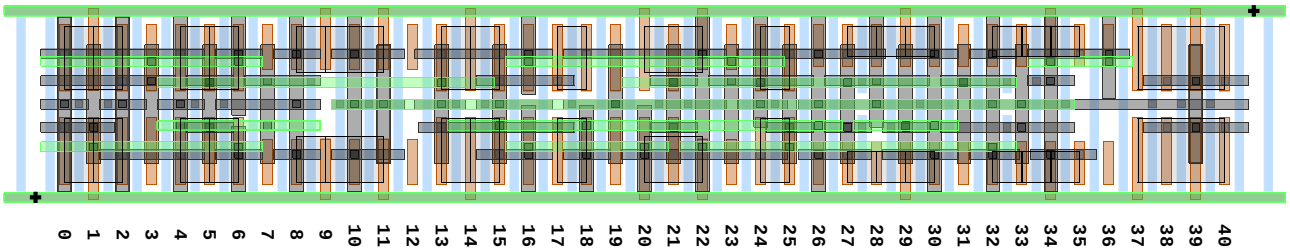
V. CONCLUSION

We have presented BonnCell, a new flow for the automatic cell layout that is able to deal with the challenges arising in 7nm technology. The main features are the global optimization of several design objectives and the full integration of the routing algorithm into the placement algorithm. This allows guaranteeing DRC clean layouts that globally optimize the cell area with given secondary design objectives, e.g. wire length, M2-limitations, pin access and so on.

All solutions found by BonnCell were at least as good as the best designer's solution. Moreover, BonnCell finds these solutions much faster than a designer. On a multi-core machine, the whole library reported in Table I which contains 120 different cells can be processed within five minutes. As BonnCell's layouts are DRC clean by construction no additional post processing is necessary. This allows generating several different layouts for all cells of a library to see the overall impact that these different layouts have on later stages in the layout of the whole chip, e.g. routability, timing, area, power, pin access and so on. It is also possible to add more complex cells to the library, e.g. gates with inverted inputs/outputs or wider AOIs and OAIs which may help to improve the overall timing or power consumption of the chip.



(a) Manual layout by an experienced designer.



(b) BonnCell's layout.

Fig. 13. Layouts of the latch SDFFQ X3. BonnCell's layout needs 5% less area than the best designer's solution while using the same number of M2 tracks.

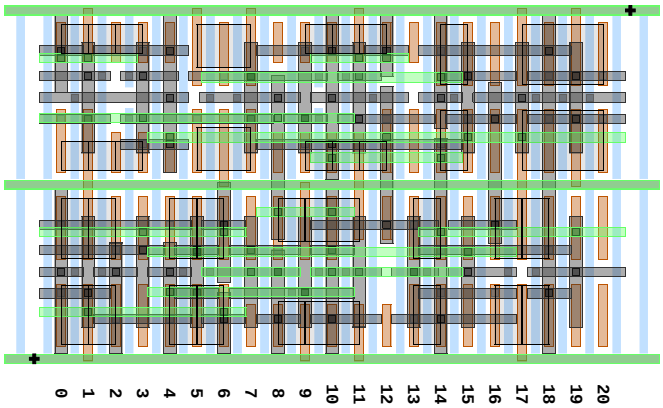


Fig. 14. BonnCell's 2-row layout of the latch SDFFQ X3.

References

[1] B. Basaran and R. A. Rutenbar. An $O(n)$ algorithm for transistor stacking with performance constraints. In *Proc. DAC'96*, pages 221–226, 1996.

[2] J. H. Chen, T. A. Spooner, J. E. Stephens, M. O'Toole, N. LiCausi, B. Kim, S. Narasimha, and C. Child. Segment removal strategy in SAQP for advanced BEOL application. In *Interconnect Technology Conference (IITC)*, pages 1–3. IEEE, 2017.

[3] P. Cremer, S. Hougardy, J. Schneider, and J. Silvanus. Automatic cell layout in the 7nm era. In *ISPD '17*, pages 99–106, 2017.

[4] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

[5] M. Grötschel, A. Martin, and R. Weismantel. The Steiner tree packing problem in VLSI design. *Math. Program.*, 78:265–281, 1997.

[6] A. Gupta and J. P. Hayes. Optimal 2-D cell layout with integrated transistor folding. In *ICCAD'98*, pages 128–135. IEEE, 1998.

[7] N.-D. Hoàng and T. Koch. Steiner tree packing revisited. *Math Meth Oper Res*, 76(1):95–123, 2012.

[8] K. Jo, S. Ahn, T. Kim, and K. Choi. Cohesive techniques for cell layout optimization supporting 2D metal-1 routing completion. In *23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 500–506, 2018.

[9] I. Kang, D. Park, C. Han, and C.-K. Cheng. Fast and precise routability analysis with conditional design rules. In *Proceedings of the 20th System Level Interconnect Prediction Workshop*, Article no. 4, 2018.

[10] C. Lazzari, C. Santos, and R. Reis. A new transistor-level layout generation strategy for static CMOS circuits. In *ICECS'06*, pages 660–663, 2006.

[11] J. Lienig and M. Thiele. The pressing need for electromigration-aware physical design. In *ISPD '18*, pages 144–151, 2018.

[12] H. Liu, T. Han, J. Zhou, and Y. Chen. Layout decomposition and synthesis for a modular technology to solve the edge-placement challenges by combining selective etching, direct stitching, and alternating-material self-aligned multiple patterning processes. In *Design-Process-Technology Co-optimization for Manufacturability X*, volume 9781. International Society for Optics and Photonics, Article no. 25, 2016.

[13] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen. Open cell library in 15nm freePDK technology. In *ISPD'15*, pages 171–178. ACM, 2015.

[14] S. Narasimha, B. Jagannathan, A. Ogino, D. Jaeger, B. Greene, C. Sheraw, K. Zhao, B. Haran, U. Kwon, A. Mahalingam, et al. A 7nm CMOS technology platform for mobile and high performance compute application. In *IEDM'17*, pages 29.5.1–29.5.4, 2017.

[15] C. J. Poirier. Excellerator: Custom CMOS leaf cell layout generator. *IEEE Trans. CAD*, 8(7):744–755, 1989.

[16] T. Polzin. *Algorithms for the Steiner problem in networks*. PhD thesis, MPII Saarbrücken, 2003.

[17] G. Posser, V. Mishra, P. Jain, R. Reis, and S. S. Sapatnekar. Cell-internal electromigration: Analysis and pin placement based optimization. *IEEE Trans. CAD*, 35(2):220–231, 2016.

[18] N. Ryzhenko and S. Burns. Standard cell routing via boolean satisfiability. In *DAC'12*, pages 603–612, 2012.

[19] B. Taylor and L. Pileggi. Exact combinatorial optimization methods for physical design of regular logic bricks. In *DAC'07*, pages 344–349. ACM, 2007.

[20] T. Uehara and W. M. vanCleemput. Optimal layout of CMOS functional arrays. *IEEE Trans. on Computers*, 30(5):305–312, 1981.

[21] R. Vicari. Simplex based graphs yield large integrality gaps for the bidirected cut relaxation. Master's thesis, Research Institute for Discrete Mathematics, University of Bonn, 2018.

[22] S. Wimer, R. Y. Pinter, and J. A. Feldman. Optimal chaining of CMOS transistors in a functional cell. *IEEE Trans. CAD*, 6(5):795–801, 1987.

[23] R. T. Wong. A dual ascent approach for Steiner tree problems on a directed graph. *Math. Program.*, 28(3):271–287, 1984.

[24] P.-H. Wu, M. P.-H. Lin, T.-C. Chen, T.-Y. Ho, Y.-C. Chen, S.-R. Siao, and S.-H. Lin. 1-D cell generation with printability enhancement. *IEEE Trans. CAD*, 32(3):419–432, 2013.

[25] W. Ye, B. Yu, Y.-C. Ban, L. Liebmann, and D. Z. Pan. Standard cell layout regularity and pin access optimization considering middle-of-line. In *GLSVLSI'15*, pages 289–294. ACM, 2015.

[26] B. Yu, X. Xu, S. Roy, Y. Lin, J. Ou, and D. Z. Pan. Design for manufacturability and reliability in extreme-scaling VLSI. *Science China Information Sciences*, 59(6):1–23, 2016.