Algorithms for Cell Layout

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Vorgelegt von

Pascal Cremer

aus

Neuss

Bonn, Januar 2019

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter:	Prof. Dr. Stefan Hougardy
2. Gutachter:	Prof. Dr. Jens Vygen
Tag der Promotion:	5. April 2019
Erscheinungsjahr:	2019

Acknowledgments

This thesis would not have been possible without the kind support of many people. First and foremost, I would like to express my gratitude to Prof. Dr. Stefan Hougardy for his excellent guidance throughout my time at the institute. Our discussions have been invaluable for the research that has gone into BonnCell.

I would also like to thank Prof. Dr. Jens Vygen, whose imaginative ideas highly contributed to the long term vision of the project. My special thanks goes to Prof. Dr. Dr. h.c. Bernhard Korte for providing the excellent working conditions at the institute.

BonnCell would be of much less use if it wasn't integrated so well into the IBM environment. Tobias Werner, Gerhard Hellner, and Dr. Iris Leefken took care of this and much more that made me enjoy our collaboration.

Many students contributed to BonnCell. Thanks go to Lars Friederichs, Alexander Göke, Andreas Gwilt, Thekla Hamm, Silas Rathke, Simon Thomae, and Robert Vicari, who brought in their own ideas and patiently endured countless code reviews. I would especially like to thank Benjamin Klotz, whose work as a student has been invaluable and who seamlessly took over the project so that I had time to write this thesis.

Out of the many great colleagues I had at this institute, I would especially like to thank Jannik Silvanus and Philipp Ochsendorf for our in depth discussions on C++ and their tremendous effort to improve compilation times and create powerful general purpose tools for the entire institute. I also wish to thank Jan Schneider, who wrote the first version of BonnCell and allowed me to experience a smooth transition into the project. I would like to thank Andreas Gwilt and Konstantin Fröhlich for their critical proofreading of parts of this thesis.

Finally, thanks go to my family and to my fiancée Anne, who supported me greatly when writing this thesis.

Contents

Acknowledgments **iii**

1 Introduction **1**

2 Previous Work **3**

2.1 Categories of Cell Layout Generators . . . 3
2.2 Layout Styles . . . 4
2.3 Placement . . . 4
2.4 Design Rules . . . 6
2.5 Routing . . . 6
2.6 Challenges due to Manufacturability . . . 7
2.7 Comparison to Other Tools . . . 8

3 Definitions and Goals **9**

3.1 Problem Definition . . . 9
3.2 Design Rules . . . 12
3.2.1 Legality . . . 12
3.2.2 Routability . . . 13
3.3 Objective Function . . . 14

4 Routing **17**

4.1 Modular Router Structure . . . 18
4.2 Mixed Integer Programming Formulation . . . 18
4.2.1 Grid Graph Construction . . . 18
4.2.2 Steiner Tree Packing . . . 19
4.2.3 Conditional Constraints . . . 20
4.2.4 Mapping Design Rule Constraints . . . 21
4.2.5 Trim Shape Model . . . 22
4.2.6 Vias . . . 24
4.3 Routing Oracle During Placement . . . 24
4.4 Post Processing . . . 25

5 Placement Algorithm **27**
5.1 Placement Algorithm . . . 27
5.2 Phases . . . 28
5.3 Routing of Partial Placements . . . 33
5.4 FEOL Routing Oracle Cache . . . 34
5.5 Cell Width Pruning . . . 42
5.5.1 Combinatorial Approach . . . 42
5.5.2 MIP Approach . . . 49
5.6 Netlength Pruning . . . 55
5.7 Search Tree Ordering . . . 58
5.8 Parallelization . . . 61
5.9 Routing Corridors . . . 63

6 Extensions **71**
6.1 Globally Optimum Routings . . . 71
6.2 Folding . . . 74

7 Big Cell Placement **77**
7.1 Multibit Cells . . . 78
7.2 Divide Placer . . . 79
7.2.1 Routability Guaranty . . . 83
7.2.2 Subcell Splitting . . . 84
7.2.3 Runtime Distribution . . . 85
7.3 Linear Arrangement Placer . . . 86
7.3.1 Min Cut Linear Arrangements . . . 88
7.3.2 Algorithm of Linear Arrangement Placer . . . 92
7.4 Results . . . 94

8 Comparison to Manual Layouts **97**
8.1 Standard Cells . . . 97
8.2 Latches . . . 97
8.3 PLCB . . . 100

9 Summary **107**

A Testbeds **109**
A.1 Standard Cells . . . 109
A.2 Latches . . . 109

Bibliography **113**

Chapter 1

Introduction

The design and layout of a modern computer chip is a huge project. To facilitate independent development of different parts of the chip, several layers of hierarchy are introduced. The lowest level of the hierarchy is the cell level. Cells are logical gates and storage elements consisting of transistors connected with wires. In this way, the higher levels of the hierarchy are almost independent of placement and routing of individual transistors as cells are seen as black boxes with specified input and output wire locations. Although this abstraction greatly reduces the complexity of the layout process, some design and layout flexibility is lost. Cell layouts might not be optimal for the specific location in which they are used, resulting in increased area usage, worse timing properties, and higher power consumption. As a remedy, a large library of cells is used which contains many variants of the same logical function, such that the best variant for the given situation can be chosen. Some parts of the chip, e.g. static random-access memory (SRAM), are so critical that custom cells are created. These cells are used in a single context and can therefore be highly optimized to match the special needs of their environment.

Most processor chips are released on a two year cycle. With each release transistors and wires get smaller and are packed more densely to increase the chip's functionality. Although the logical functions of most cells remain unchanged, there is no simple way to migrate existing layouts from one technology node to the next. The layout properties of a cell are mainly affected by the design rules arising from the manufacturing process. These design rules are largely individual for each technology node, therefore the cell layout process can become a severe bottleneck and layout automation tools are much requested. However, so far human experts have been superior in cell layout in terms of cell quality. Cell layout tools could not realize cell layouts with the same area usage as those crafted by experienced engineers.

BonnCell, the program presented in this thesis, is the first to produce cells with provably minimum area usage in only a fraction of the time needed for manual layout. It guarantees routability of its transistor placements without using pessimistic restrictions of the search space. This is achieved by novel algorithms based on graph theory and other techniques which solve the underlying NP-hard problems in feasible time.

In addition, BonnCell supports the transition to new technology nodes, for which a cell image has to be defined which describes layout properties shared by all cells. These properties, e.g. the cell height, ensure that any two cells can always be placed legally next to each other. They have a huge impact on all cell layouts and assessment of the effect on area usage and other features of the standard library is fundamental when designing the cell image. Therefore, human experts evaluate cell images on a few hand crafted cell layouts. With BonnCell larger sets of cell layouts and cell images can be compared against each other to find the best cell image for the chip.

This thesis is organized as follows. Previous work on cell layout generators is presented next in Chapter 2. In Chapter 3 we define the cell layout problem and the objective function used to evaluate cell quality. BonnCell can be split into two main algorithms for placement and routing of transistors. The routing algorithm is called many times from within the placement algorithm to evaluate the routability of placements and is therefore presented first in Chapter 4. Subsequently, we present the placement algorithm in Chapter 5. Details on the algorithms for runtime optimization and the routability evaluation are provided. In Chapter 6 we discuss two BonnCell features which extend the capability of the placement algorithm. Some cells are too large to be solved by a single call of the placement algorithm. These cells are split into smaller subcells which are placed and routed individually and afterwards merged into a placement and routing of the original cell. The algorithms for these big cells are provided in Chapter 7. Finally, in Chapter 8 we compare the layouts generated by BonnCell to manually crafted ones.

Chapter 2

Previous Work

Although layout automation is widely used in higher layers of the chip hierarchy, cells are usually hand crafted by experienced human layouters. This is a very time consuming process which limits the number of library and custom cells that can be created. Transistor level layouts heavily depend on the design rules and they have to be rebuilt from scratch with every new technology. To reduce the cost of this process, cell layout tools are much desired.

Most modern work on automated cell layout is based on the 1-D layout scheme introduced by Uehara and Cleemput 1979. In this scheme transistors are placed in two rows. N type transistors ($NFETs$) are placed on one row next to the ground voltage power rail and P type transistors ($PFETs$) are placed on the other row next to the supply voltage power rail. Previous work based on different layout schemes is difficult to compare to current approaches due to the large differences in their underlying algorithmic problems. Schneider 2014 gives a detailed review of work on other layout styles.

2.1 Categories of Cell Layout Generators

Lefebvre, Marple, and Sechen 1997 distinguish three categories for cell layout generators: procedural generators, re-compaction methods, and cell synthesis tools. Procedural generators use a domain specific language to describe each cell individually. The description of each cell is usually produced by human experts. Re-compaction methods aim at improving existing layouts. This includes migration of existing layouts to new technologies or improvement of certain layout properties like area usage as done by Fu et al. 2009. Cell synthesis tools build a transistor level layout based on the netlist only. This is the only category of tools with no requirement of previous layout information. Some cell synthesis tools also include a compaction step to post optimize their results, e.g. Ziesemer and Luz Reis 2014 use a MIP based compaction algorithm to improve layouts generated by their cell synthesis tool ASTRAN. We will focus on cell synthesis tools as they require no human interaction. BONNCELL also falls into the category of cell synthesis tools.

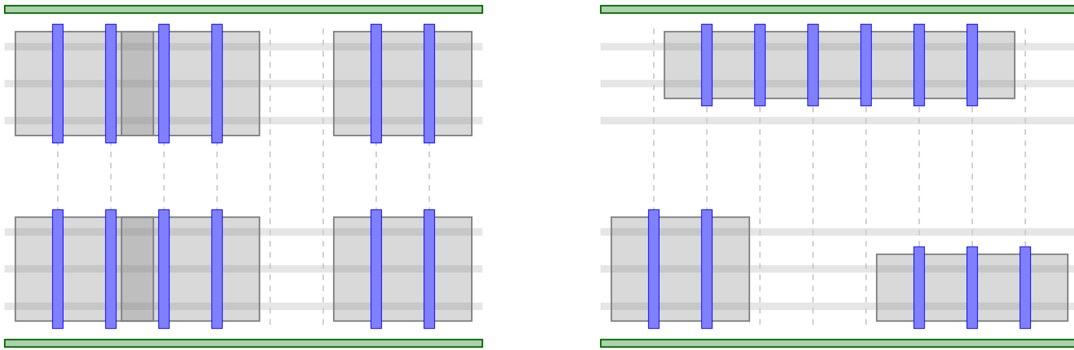


Figure 2.1: 1-D dual cell layout style (left) and 1-D non-dual cell layout style (right). Dual cells are easier to place and route, as PFETs and NFETs appear in pairs which can be placed opposite of each other with the same gate connection. This is not the case for non-dual cells where there might be different numbers of PFETs and NFETs.

2.2 Layout Styles

Many logic circuits can be implemented by dual pull-up and pull-down transistor networks, which means that transistors appear in pairs with the same gate connection. Figure 2.1 shows a dual and a non-dual cell layout. Many algorithms, including the seminal work by Uehara and Cleemput 1981, specialize on dual cells as it simplifies the placement problem, cf. Maziasz and Hayes 1991; Nair, Bruss, and Reif 1983; Nyland and Reif 1996. However, Iizuka 2006 estimates that only about 50% of the required cells are dual. BonnCell is not restricted to dual cells.

Every cell synthesis tool has some restrictions on the layout styles it supports. As written above, many tools only support 1-D dual cells. 1.5-D layout means that PFETs can be extended into the N region and vice versa. This allows more compact placements for cells with unbalanced size of PFETs and NFETs but requires the tool to prevent illegal intersections of PFETs and NFETs. In 2-D layout the cell is placed on multiple circuit rows and the routing problem includes connecting FETs placed on different circuit rows. The 1.5-D and 2-D layout styles are illustrated in Figure 2.2. Finally, free layouts, as used by Riepe and Sakallah 2003, do not make any of these restrictions on position or orientation of FETs. BonnCell supports all these layout styles, except for free layouts.

2.3 Placement

Previous approaches to the transistor placement problem differ in several aspects. For example, tools support different degrees of flexibility for FET folding, i.e. realizing a FET with different numbers of gate fingers. Support for folding is essential to achieve dense layouts without the need for manual intervention. Hill 1985 was the first to allow *static folding*, i.e. choosing one folding configuration before or after the FET placement algorithm is run. Berezowski 2001 was the first to explore different layout and folding options simultaneously, so called *dynamic folding*, in an algorithm based on

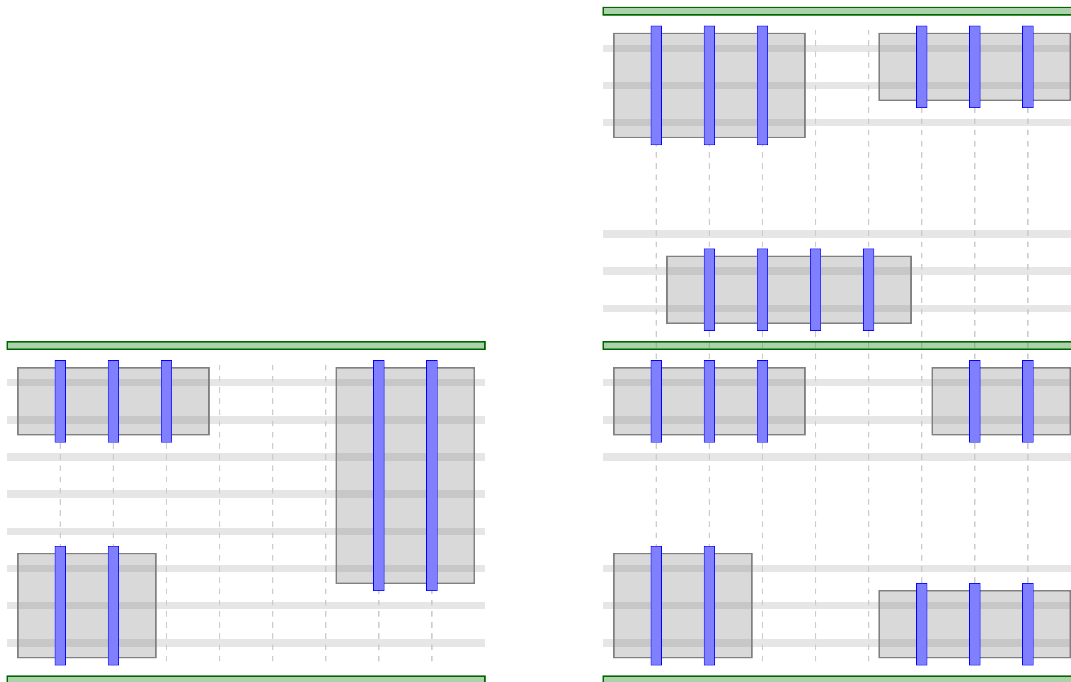


Figure 2.2: Continuation of layouts styles. The left image shows 1.5-D layout style where PFETs can be extended into the N region and vice versa. The right image shows a 2-D layout on multiple circuit rows, where each circuit row can be built in 1-D or 1.5-D layout style.

dynamic programming. BonnCell supports dynamic folding by enumerating different FET widths during the placement algorithm. It also provides an extended folding mode which splits FETs into smaller, individual FETs. These FETs can be placed independently of each other which allows many more placement configurations. In some situations this technique helps to significantly reduce the cell area compared to standard dynamic folding.

The placement objective function of cell synthesis tools does not differ much between different work, cell area is almost always the top priority. Many placement algorithms additionally optimize for routability in some way. This can be in the form of estimating pin density or horizontal length of gate to gate connections, both used by Ziesemer and Lazzar 2007. Some more recent work take manufacturability into account as well, cf. Wu et al. 2013. BonnCell's primary objective is also cell area. As routability is guaranteed by the algorithm it does not have to be optimized in the objective function. Instead, an estimation for routing netlength is minimized.

Introduced by Uehara and Cleemput 1981, Euler chains have been used in many tools to find minimum area placements for dual cells. Since many variants of the placement problem become NP-hard if transistor folding and non-dual cells are allowed, many other algorithms have been developed. Methods can be divided into exact approaches and heuristics. Exact approaches like mixed integer programming (*MIP*), cf. Gupta and Hayes 1998, satisfiability (*SAT*), cf. Iizuka 2006, and dynamic

programming, cf. Bar-Yehuda et al. 1989, have the advantage that optimality of the solution can be guaranteed. Heuristics like simulated annealing, cf. Guruswamy et al. 1997, and threshold accepting, cf. Ziesemer et al. 2014, can be more suitable for larger instances, where the exact algorithms fail to find any solution. BonnCell combines both approaches by solving small and medium sized cells using an exact branch-and-bound approach. Larger instances are heuristically split into smaller subcells which can then be solved optimally. BonnCell enumerates all feasible placements and uses Euler chains to prune infeasible parts of its search tree.

2.4 Design Rules

Most cell synthesis tools use a grid based representation of design rules, cf. Cortadella et al. 2014; Iizuka 2006; Kang et al. 2018; Karmazin, Otero, and Manohar 2013. In these tools a routing grid is used and wires need to end on vertices of the grid with some fixed wire width and via overhang. This has the advantage that wire spacing and other design rules are easy to formulate. However, depending on the design rules, this imposes a restriction on the routing space. For example, the rules might require the wires to be a distance of 1.5 grid edge lengths apart from each other. Due to the coarseness of the grid, the shortest legal distance that could then be modeled is 2 edge lengths. BonnCell is able to represent every legal solution. It avoids a grid based representation of design rules by allowing line end coordinates which are continuous in one direction, whereas the other direction is gridded by the cell image.

2.5 Routing

Rip-up and reroute is a common approach to cell routing, cf. Jo et al. 2018; Karmazin, Otero, and Manohar 2013. Nets are routed sequentially, which means that already routed nets block routing space which might be essential for nets that still have to be routed. If a net cannot be routed given some already routed nets, the wires of some of the routed nets are removed (ripped-up) and rerouted after the conflicting net has been routed. Of course, there is no guarantee that the ripped-up nets can be successfully rerouted. Usually this strategy is combined with a dynamic cost function which penalizes routing in highly congested areas. These approaches are all heuristics and therefore not able to prove that a placement of FETs cannot be routed. Some routing algorithms do not explore the entire routing search space but use a fixed set of heuristically generated routes, e.g. Wu et al. 2013.

More recent work focuses on MIP or SAT based approaches which do not depend on rip-up and reroute but route all nets simultaneously. Cortadella et al. 2014 use a SAT formulation with gridded design rules which supports double-patterning lithography. Kang et al. 2018 present a MIP based routing formulation with a focus on pin accessibility. Design rules are modeled on a grid with each wire segment either fully present or absent. This allows turning the MIP problem into a SAT problem, which is solved within a fraction of the MIP runtime. Compared to BonnCell their approach respects fewer routing layers at the bottom of the layer stack. Transistors and related

front-end-of-line layers (TS, CA, PC, CT) are fixed during routing and the grid based approach does not allow to place line ends continuously. A very different SAT formulation has been used by Ryzhenko and Burns 2012. Their approach enumerates different routes for each net using a maze algorithm and detects conflicts between routes. The SAT formulation ensures that exactly one route is chosen for each net and no two routes are in conflict.

2.6 Challenges due to Manufacturability

Multiple patterning techniques are now widely used to overcome the limitations of photo lithography, cf. Arnold 2009; Liebmann, Chu, and Gutwin 2015; Lin 2009; Pan 2009. These techniques allow to continuously produce smaller feature sizes with each technology node, despite being limited by 193nm immersion lithography. Layout automation flows need to adapt to multiple patterning and the resulting tool requirements.

There exist many different multiple patterning techniques, each one with different advantages and disadvantages, cf. Pan, Yu, and Gao 2013. To allow for some depth of the discussion, we focus on manufacturing of transistor gates using self-aligned double patterning (SADP) as described by Haffner et al. 2007, 2008; Lai et al. 2008. The gates are produced in two steps. First, a regular, unidirectional pattern of poly silicon (PC) is generated by using sidewall spacers. In a second step, unwanted PC features are cut off using the cut mask (CT). This technique is successful in limiting line end roughness and allowing smaller gate to gate distances, cf. Burkhardt et al. 2009; Lai et al. 2008; Sarma et al. 2008.

These manufacturing requirements bring new challenges for cell synthesis tools. Haffner et al. 2007 described the layer decomposition of desired gate features with given dimensions into a PC and CT mask. However, they fix the gate lengths before layer decomposition which often turns out to eliminate all legal decompositions. Our approach is to fix the transistor positions but leave their gate lengths variable within given boundaries. After transistor placement, gate lengths are determined simultaneously for the entire cell in a SADP compliant manner. The boundaries for the gate lengths are chosen such that the transistor's electrical properties, e.g. number of covered fins, do not change.

Double patterning (DP) and multi patterning (MP) aware routing has been intensely studied for different techniques. Litho-etch-litho-etch (LELE), cf. Pan, Yu, and Gao 2013, is an alternative to SADP. LELE uses two exposures and etches to create two coarse patterns. They are combined into a fine pattern which has twice the resolution of the individual patterns. Instead of creating features and cutting off extraneous parts as in SADP with a cut mask, both masks are used to generate independent features. Due to the differences between LELE and SADP, LELE aware routing techniques cannot be applied to SADP masks. Many approaches to handle SADP in automated design have been suggested, cf. Du et al. 2013; Fang 2015; Gao and Pan 2012; Kodama et al. 2013; Mirsaeedi, Torres, and Anis 2011; Pan, Yu, and

Gao 2013; Xu et al. 2015. Our work differs to previous work for SADP in several aspects. First, we allow variable transistor gate lengths which gives more flexibility to find solutions without design rule violations. Second, our algorithm guarantees to find solutions without design rule violations, if existent.

2.7 Comparison to Other Tools

The main contribution of this thesis is that BonnCell is able to produce provably minimum area layouts without design rule violations. Other tools cannot guarantee routability of their placements or use pessimistic restrictions during the placement phase, thereby wasting area. This is the first tool which is able to guarantee routability of its placements without sacrificing cell area. This is achieved by sophisticated lower bounds based on graph theory and other speed up techniques.

Previous versions of BonnCell have been described in Hougardy, Nieberg, and Schneider 2013 and Schneider 2014. The version presented in this thesis has many improvements compared to the previous ones. The routing algorithm is built upon a new MIP formulation with better theoretical performance guarantee and is able to route all nets simultaneously with exact representation of the design rules. It can guarantee routability of its placements and supports an extended folding mode. Finally, the initial version of the big cell mode has been extended and improved to give denser placements with guaranteed routability, even for very large cells spanning multiple circuit rows. A preliminary version of the work in this thesis has already been published in Cremer et al. 2017.

Chapter 3

Definitions and Goals

In this chapter we formalize the Cell Synthesis Problem (Section 3.1) and give an overview over the design rules and their origin (Section 3.2). Furthermore, we describe the objective function we use when solving the Cell Synthesis Problem (Section 3.3).

3.1 Problem Definition

BonnCell's tasks can be divided into two parts. Placement of *field-effect-transistors* (FETs) and routing of their connections. In this process many constraints, so called *design rules*, need to be respected. Some design rules are motivated by manufacturability restrictions, others are conventions to ensure that different cells can be placed next to each other without conflict.

A FET has three contacts, *source*, *drain*, and *gate*. The function of a FET is to connect its source and drain contacts if and only if a certain voltage is applied at its gate. It can be thought of as a voltage controllable switch. Furthermore, it acts as an amplifier, as a small gate current suffices to generate a large drain current. FETs exist in two types: *p-FET* and *n-FET*. For n-FETs, the source and drain contacts are connected if and only if supply voltage (VDD) is connected to the gate. Conversely, p-FETs connect source and drain if and only if ground voltage (VSS) is connected to the gate. These two devices suffice to build standard logic gates like inverters, NANDs, and NORs, but also storage cells like latches. Figure 3.1 shows the CMOS implementation of a NAND gate.

FETs can be built with different *VT levels*. The VT level determines the trade off between power consumption and switching speed (timing) of a FET. Cells with the same logical functionality are usually built in different versions s.t. later design stages can choose the optimum trade off between cell area, power consumption, and timing.

Throughout the chip horizontal power rails ensure the connection of FETs to *VSS* and *VDD*. The cells are placed in so called *circuit rows* in between these power rails. The circuit rows are also called *bits* and large cells can be built as *multi bit* cells, spanning multiple bits in vertical direction. For single bit cells, the transistors are arranged in two *stacks*, one next to each power rail. One stack consists of the cell's

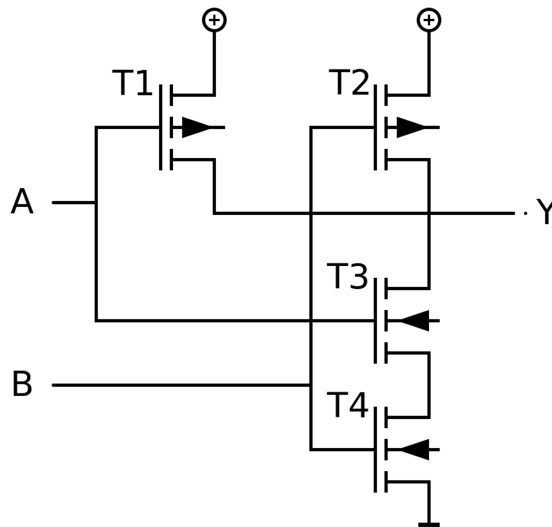


Figure 3.1: CMOS implementation of a NAND gate. Output Y reads 1 (VDD) if one or both of the inputs A and B are 0 (VSS), otherwise it reads 1. The implementation uses four FETs, two p-FETs (T1, T2) and two n-FETs (T3, T4). If A or B is connected to VSS, at least one of the p-FETs, T1 and T2, connects VDD to Y and at least one of T3 and T4 will disconnect VSS from Y. If both A and B are connected to VDD, T1 and T2 are blocked and T3 and T4 connect Y to VSS. Image source: Wikipedia.

n-FETs and is placed directly next to the lower power rail (VSS), whereas the other stack contains the p-FETs and is placed directly next to the upper power rail (VDD).

There are different areas of the chip with different purposes and different requirements. These requirements are captured by the *cell image* definition. The cell image describes the general structure of a cell. For example, different cell images have different heights determined by the distance of power rails.

A FET can be built with different sizes. The larger the FET, the lower its resistance and the more current it can supply. The 7nm technology node uses *FinFETs*. For these devices, the size of a FET is discretized in the number of *fins* it intersects. Fins are horizontal shapes parallel to the power rails. Number and positions of fins are given by the cell image. Figure 3.2 shows an image of a cell with placed FETs.

A FET F is defined by a tuple $(S_{min}, S_{max}, N_g, N_s, N_d, t, \nu)$, where

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

A legal realization of a FET must intersect $S \in [S_{min}, S_{max}]$ fins. We allow FETs to be realized with different numbers of *fingers*. Therefore, solving the placement problem

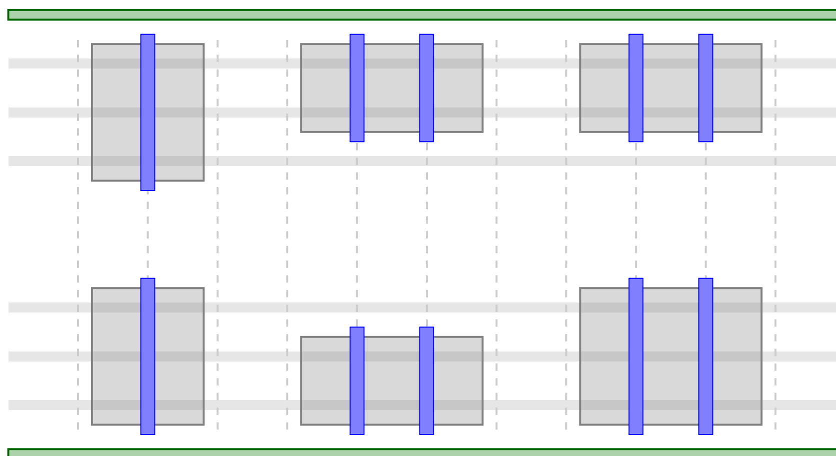


Figure 3.2: Placement of a single bit cell with 6 FETs, 3 n-FETs (lower stack) and 3 p-FETs (upper stack). FETs are covering 2 or 3 fins (thin horizontal rectangles). This cell image allows FETs to have a height of at most 3 fins. Green rectangles represent power rails. Blue rectangles represent the gates of the FETs. Some FETs are built using 1 finger, others using 2 fingers.

does not only include the assignment of locations to each FET 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 a number of 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 $S_{min} = S_{max} = 6$, 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. all 7nm 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 contacts of the FET exchange their places. Figure 3.3 shows the same FET realized in three different ways.

Definition 3.1. The configuration c of a FET is defined as the tuple (x, f, h, s) where

- $x \in \mathbb{N}$ is the FET's x location (measured in PC tracks),
- $f \in \mathbb{N}_{>0}$ is the number of fingers,
- $h \in \mathbb{N}_{>0}$ is the height measured in fin intersections per finger, and
- $N_L \in \{N_S, N_D\}$ is the leftmost contact net.

We call a FET F *swapped* if $N_L = N_D$. 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.

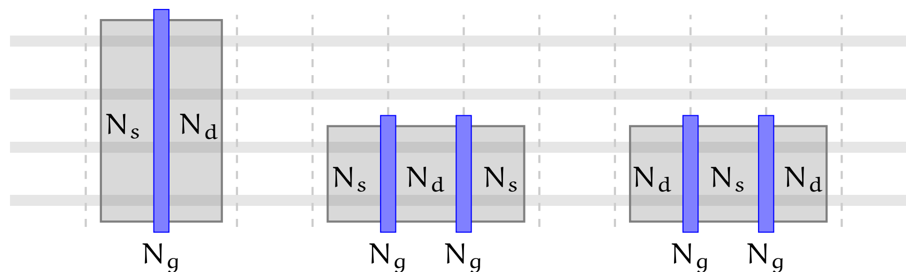


Figure 3.3: A FET of size 4 realized with 1 finger, 2 fingers, and 2 fingers swapped. The heights are 4, 2, and 2 fins respectively. Gates are shown in blue, source and drain contacts in gray.

Definition 3.2. A placement C is defined as a tuple $C = (c_1, \ldots, c_n)$, where c_i is the configuration for FET F_i.

3.2 Design Rules

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

3.2.1 Legality

Due to the location of the fins, the cell image guarantees that FETs do not overlap vertically, cf. Figure 3.2. However, depending on their nets, some FETs will not be able to be placed opposite of each other with a height of 3 fins. In horizontal direction FETs need to obey some minimum distance depending on their configuration but this rule only applies to neighboring FETs. Two FETs of a placement are neighbors, if there is no other FET in between them. Two neighboring FETs are allowed to share contacts if they have

- the same height,
- same VT level, and
- the facing contacts belong to the same net.

More formally, this gives the following definition.

Definition 3.3. Two neighboring FETs F_1, F_2 with configurations c_1, c_2 and F_2 placed to the right of F_1, i.e. $x(c_2) > x(c_1)$ are allowed to share if

- $h(c_1) = h(c_2)$,
- $VT(F_1) = VT(F_2)$, and
- $N_R(F_1, c_1) = N_L(F_2, c_2)$,

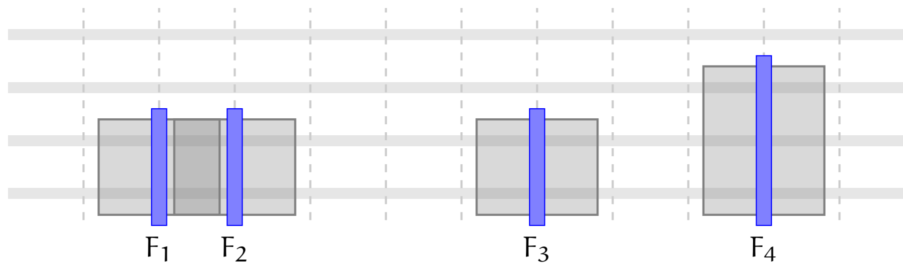


Figure 3.4: Illustration of placement legality rule. FETs F_1 and F_2 have the same VT level, same height and same contact net facing each other. Thus they are allowed to share their diffusion regions. FETs F_3 and F_4 have different height and must therefore be separated by two empty gate tracks.

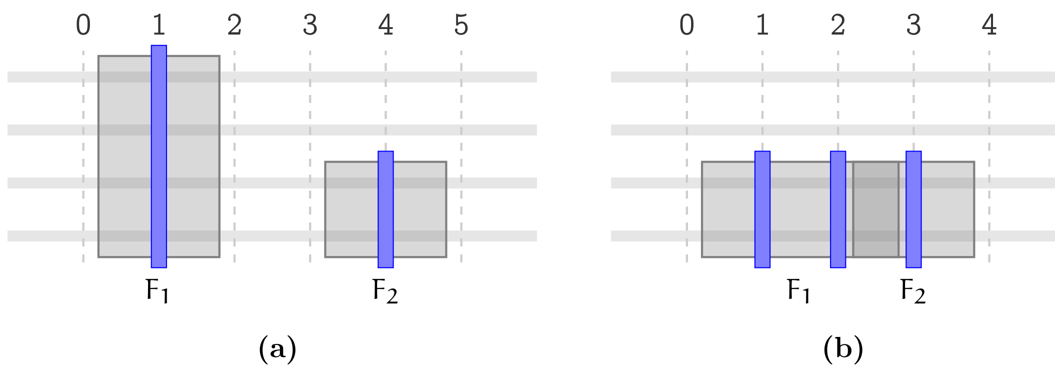


Figure 3.5: Placing a FET with larger width can result in denser placements. F_1 has size 4 and can be built with a single finger intersecting 4 fins, or 2 fingers intersecting 2 fins each. Although wider itself, the 2 finger option results in a more compact layout as it allows sharing with F_2. **(a)** Densest placement if F_1 is built with height 4. **(b)** With F_1 having 2 fingers, F_1 and F_2 have the same height and can share contacts.

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 $x(c_2) \geqslant x(c_1) + f(c_1)$. Otherwise it is legal if $x(c_2) \geqslant x(c_1) + f(c_1) + 2$.

For sharing FETs the diffusion regions overlap and the contact is used simultaneously by both FETs. If sharing is not allowed, the gates must have a minimum distance of 3 PC pitches. Figure 3.4 gives an illustration of this rule. Placements which obey this rule are called *legal*. This distance requirement makes the placement problem complicated, even for a single stack. As illustrated in Figure 3.5, increasing the width of a FET can allow a denser placement.

3.2.2 Routability

We call a placement *legal* if it obeys the FET sharing rules described above. A legal placement might not be manufacturable, as there are many complicated rules, so called design rules, that need to be obeyed. For example, all gates are manufactured using the

self aligned double patterning (SADP) technique. 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. Placements which pass the routability check, cf. Chapter 4, are called *routable*.

The placement legality rules will be obeyed by our placement algorithm by construction. Routability rules are checked by using the routing algorithm as a black box oracle. Only routable placements are returned by our algorithm. This is very important as has been seen with past technologies when BonnCell wasn't able to guarantee routability. In many cases the solution found by BonnCell was violating design rules, i.e. all FETs were placed and nets routed without shorts or opens but the solution was violating design rules. These remaining violations had to be fixed manually which was time consuming and not always possible without changing the placement. The main contribution of this thesis is that BonnCell is able to produce layouts without design rule violations and without artificial restrictions of the placement or routing search space.

3.3 Objective Function

We are only interested in placements which are routable under consideration of all design rules. However, there may be many routable placements for a given instance. While each of these solutions is acceptable, some of them are more preferable than others. Minimizing cell area plays a key role as it allows more compact placements of cells. The smaller the cells, the more functionality one can pack on the limited chip area. Therefore, the most important criteria in BonnCell's objective function is area. Since the cell height is given by the cell image, this means we minimize the cell width.

There might still be many solutions with minimum cell width. From these solutions we would like to choose the one with minimum routing objective value. This means for a given placement, we find the optimum routing w.r.t. the routing objective function Ψ, which mainly measures weighted netlength but also takes other metrics, e.g. M2 track usage, into account. Finding a placement which admits a routing with globally minimum routing objective value is only achievable for small cells and the topic of Section 6.1. For medium to large sized cells, the runtime to calculate optimum routings for each legal placement is too large. Therefore, by default, we optimize the *weighted bounding box netlength* which serves as a proxy for the routing objective value. We calculate lower bounds for the weighted bounding box netlength for partial placements which allows us to prune parts of the search tree for which we can prove that they cannot contain the optimum solution.

To summarize, BonnCell returns the solution P which minimizes

$$\Phi(P) = (W(P), \phi(P))$$

lexicographically, where $W(P)$ is the placement width and $\phi(P)$ the weighted bounding box netlength of P. More details on the weighted bounding box netlength of P will be given in Section 5.6. The final problem definition reads

Cell Synthesis Problem

Instance: Nets $\mathcal{N}$, FETs $\mathcal{F}$, a cell image and a set of design rules.
Task: Find a *legal* and *routable* placement P which minimizes the placement objective function Φ. Furthermore, find a routing of P, minimizing the routing objective function Ψ.

Chapter 4

Routing

In previous versions of BonnCell the program flow was divided into two steps. In the first step, a placement P was computed which minimized some auxiliary objective function $\phi(P)$. In the second step, this placement was routed by the routing algorithm. Since many placements are not routable, the placement objective function ϕ measured features which empirically correlated with routability, e.g. gate to gate netlength. The current version of BonnCell still uses these two steps. However, the main difference is that we no longer use a placement objective function which tries to prefer routable placements but check routability directly during the placement. This is done by using the main routing algorithm already during the placement step. This approach guarantees that the placement algorithm will return a routable solution. The approach also has a second advantage. Additional routing constraints, like blockage of certain tracks or routing pins at certain positions, are essential for cell synthesis in real world situations. 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 unroutable nodes. This simple approach is too slow in practice and several speedup techniques are used, cf. Sections 5.3 and 5.4. In this chapter we introduce the routing engine. The placement algorithm and the speed up techniques to make it fast enough will be presented in Chapter 5.

In Section 4.1 we describe our modular implementation approach to routing. Then, we give the description of our MIP model in Section 4.2, containing the grid graph construction in Section 4.2.1 and the Steiner tree packing problem in Section 4.2.2. We explain how design rules, in particular trim shape rules and via coloring rules, are incorporated into that model in Sections 4.2.3 to 4.2.6. For most of this chapter, we assume that all FETs have already been placed. In Section 4.3, we describe how the router can handle partial placement input which allows us to prune unroutable parts of the placement search tree. Finally, post optimization of routings is discussed in Section 4.4.

4.1 Modular Router Structure

Our router is based on a mixed integer programming (MIP) approach. Our goal is to find a legal routing that minimizes weighted netlength 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 to optimize arbitrary linear objective functions.

At its core the router solves the VERTEX DISJOINT STEINER TREE PACKING PROBLEM problem. Additionally, many design rules have to be obeyed. Different design rules exist for each layer and change with every new technology. BONNCELL uses a modular approach to implement these constraints. Sets of MIP variables and constraints which implement a certain set of MIP rules are bundled into a *group*. For example each layer has its own group which contains all variables to describe shapes on this layer. Additionally these groups contain some basic constraints to ensure basic relations between these variables. For example, the TS mask is described by an interval for every TS track. GROUP-TS has two variables for each TS track t, the lower l_t and upper bound u_t of the TS interval. Additional constraints guarantee that the upper bound is always above the lower bound $u_t \geqslant l_t$ and that both bounds are within the bit boundary. Shapes of neighboring layers also have to obey design rules. These rules are gathered in a separate group, e.g. GROUP-TS-GO models rules between TS and GO.

This approach has several advantages. First, it helps organizing the code in a clear way and allows different developers to modify the MIP formulation of different groups in parallel. This is important as the amount of design rules is too large to be implemented by a single developer within the required time frame. Second, the groups make it easy to activate only parts of the design rules. This will be interesting in Section 5.3 where we improve the runtime of routability checks. Third, given a MIP solution s, we can post optimize the solution while keeping a subset of variables fixed to the values given in s. We will use this in Section 4.4 to optimize our routings for *design for manufacturability* (DFM).

4.2 Mixed Integer Programming Formulation

4.2.1 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.

Edge costs are obtained by multiplying their geometric length by a layer-, track-, and net-dependent value. This allows to trade off netlength 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.

4.2.2 Steiner Tree Packing

First, we define the VERTEX DISJOINT STEINER TREE PACKING PROBLEM and describe the core MIP we use to solve it. Then, in Sections 4.2.4 to 4.2.6, we explain how design rules are incorporated into the model.

> **VERTEX DISJOINT STEINER TREE PACKING**
>
> *Instance:* A graph $G = (V, E)$ and edge costs $c : E \to \mathbb{R}_{\geqslant 0}$, a set of net indices $\mathcal{N} \subseteq \mathbb{N}$ with a set of terminals $T_k \subseteq V$ for each net $k \in \mathcal{N}$.
>
> *Task:* For each net $k \in \mathcal{N}$ find a Steiner tree S_k, i.e. S_k contains a path from s to t for each pair of vertices $s, t \in T_k$. Furthermore, the vertex sets of the Steiner trees $V(S_k)$ must be pairwise disjoint and the total cost $\sum_{k\in\mathcal{N}} c(E(S_k))$ minimized.

We solve this problem using a MIP formulation. 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 Grötschel, Martin, and Weismantel 1997, 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.

$$\sum_{e\in\delta(X)} x_e^k \geqslant 1\,.$$

However, the undirected cut relaxation has an integrality gap of 2 (Goemans and Williamson 1995), 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 $\vec{x}^k_{ij}$ 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.

$$\sum_{(i,j)\in\delta^+(X)} \vec{x}^k_{ij} \geqslant 1\,.$$

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 $\frac{6}{5}$ (Vicari 2018), 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 multi-commodity flow relaxation, first introduced by Wong 1984. This relaxation is equivalent to the bidirected cut relaxation (Polzin 2003) and was already used in by Hoàng and Koch 2012 to solve Steiner tree packing problems. We will also use the multi-commodity flow relaxation:

For each net k, we denote the set of sink terminals $T_k \setminus \{r_k\}$ by S_k. Then, the multi-commodity 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^{ks}_{ij} that is upper bounded by $\vec{x}^k_{ij}$ is introduced, representing the flow of the commodity associated to net k and sink s 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 the commodity associated to k and 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^k_v, 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 can be seen in Figure 4.1, where we denote by

$$b^{ks}(v) := \sum_{(i,j)\in\delta^+(v)} f^{ks}_{ij} - \sum_{(i,j)\in\delta^-(v)} f^{ks}_{ij}$$

the flow balance of the commodity associated to k and s at a vertex $v \in V$. In this basic formulation, no additional constraints, especially with respect to distances between shapes, are taken into consideration.

4.2.3 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 $\sum_{i\in I} a_i x_i \leqslant b$, and let x_{cond} be a binary variable. We want to model

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

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

Figure 4.1: MIP formulation to solve VERTEX DISJOINT STEINER TREE PACKING by multicommodity flows.

To this end, let U be 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

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

satisfies our needs: If $x_{\text{cond}} = 0$, then the constraint equals the original constraint, and if $x_{\text{cond}} = 1$, then $\sum_{i \in I} a_i x_i - U \leqslant b$ 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. This technique is known as the *big M* method, cf. Griva, Nash, and Sofer 2009.

4.2.4 Mapping Design Rule Constraints

For the wiring within a cell, design rules used to fall into the two basic categories of *same-net* 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

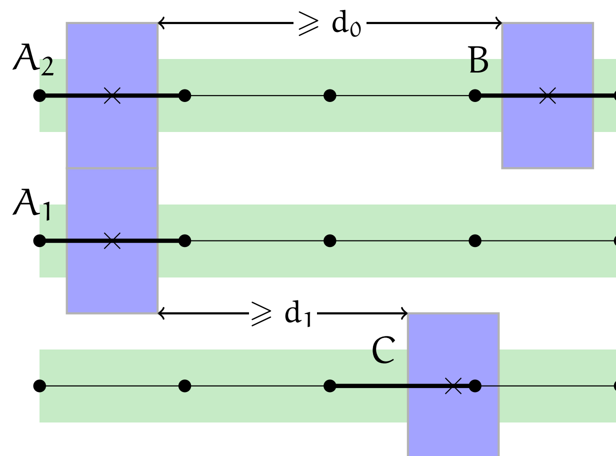


Figure 4.2: A trim shape configuration with relevant trim spacing distances, resulting wires and the assignment of trim shapes to grid graph edges. Each trim shape is assigned to an edge containing its center.

configuration, and the only additional constraints on non-via edge usage variables are consistency constraints with the trim shape model.

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.

4.2.5 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.

Now, fix a layer and without loss of generality assume that routing tracks on that layer are horizontal. Figure 4.2 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 which contains the trim shape's center. This assignment is illustrated in Figure 4.2, 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

$$\min_x(e) \leqslant t_e^{\text{pos}} \leqslant \max_x(e).$$

For two adjacent edges e, e', we have $\min_x(e') = \max_x(e) + \epsilon$, where ϵ is the base unit of the technology[1]. The distance constraints on trim shapes are then modeled as follows. Let e and f be two edges on the same track with $\max_x(e) \leqslant \min_x(f)$. There are three possible cases: If

$$\max_x(f) - \min_x(e) - W_{\text{trim}} < d_0,$$

i.e. there is no feasible trim shape configuration with both a trim shape on e and on f, then we add the constraint

$$t_e^{\text{active}} + t_f^{\text{active}} \leqslant 1,$$

modeling that at most one of the two trim shapes may be active. Otherwise, if

$$\min_x(f) - \max_x(e) - W_{\text{trim}} < d_0,$$

i.e. there are both feasible and infeasible trim shape configurations with trim shapes on e and f, we add the constraint

$$\left(t_e^{\text{active}} = 1 \wedge t_f^{\text{active}} = 1\right) \implies \left(t_f^{\text{pos}} - t_e^{\text{pos}} - W_{\text{trim}} \geqslant d_0\right),$$

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

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

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

$$t_e^{\text{active}} + x_e \leqslant 1.$$

[1] For 7nm technology, we have $\epsilon = \frac{1}{4}$nm. All coordinates have to lie on a regular grid with spacing ϵ.

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

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

4.2.6 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

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

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

$${}^m x_e + {}^m x_f \leqslant 1.$$

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

$$x_e + x_f \leqslant 1.$$

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

4.3 Routing Oracle During Placement

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.

4.4 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 netlength or via count. 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 4.2.2, in the model to the value found in the main routing step. This fixes the structure of the routing solution, i.e. which vias and edges have been used but not the exact positions of vias and trim shapes. We solve the MIP with these additional constraints and obtain a solution with the same structure as the old solution but fewer DFM violations. Since we restricted the flow variables to fixed values, most of the MIP complexity is gone and solving these MIPs takes a fraction of the time needed to solve the unconstrained MIP.

Chapter 5

Placement Algorithm

This chapter starts with the description of the core placement algorithm of BonnCell in Section 5.1. This core version is capable of finding optimum layouts but only for small cells. The combinatorial explosion of the number of possible placements makes it impossible to enumerate all of them. Most of this chapter focuses on showing how this simple placement framework is extended to prune large parts of the search tree and find optimum solutions even for larger cells.

The raw core placement algorithm spends the vast majority of the runtime in checking each fully placed instance for routability. There are three natural approaches to improve the runtime: speeding up the routability check, reducing the number of instances one has to check for routability, and parallelizing the entire algorithm. We developed several techniques which covered all of these three items. For infeasible instances, the routability check can be sped up by solving subproblems which are already infeasible (Section 5.2). We also reduced the number of fully placed nodes one has to visit with several techniques presented in Sections 5.3 to 5.7. Finally, Section 5.8 shows results for parallelization of the placement algorithm.

5.1 Placement Algorithm

The placement algorithm is based on a branch and bound approach. The basic skeleton of the algorithm is very simple and easily implemented. It guarantees routability of its returned placement by using the routing oracle which has been explained in Chapter 4. For all but very small cells the algorithm in this form will be too slow. The speed up techniques described in the following chapters are therefore essential for good results on larger cells.

The basic placement algorithm consists of two parts, PlaceCell (Algorithm 1) iterates over the cell width in increasing order and calls PlaceCellFixedWidth (Algorithm 2) for each fixed width. PlaceCellFixedWidth solves the placement problem for fixed cell width and returns the 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 of position, number of fingers, height, and swap status are tried. The resulting search tree is illustrated in Figures 5.1 and 5.2. For some FET configurations, the resulting partial placement is illegal (e.g. violating FET distance requirements) 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. Since we iterate over the cell width in increasing order, we know that the first routable placement found in this way has minimum cell width. The second objective ϕ is optimized by enumeration of all routable placements P with minimum width.

Algorithm 1: PLACECELL

input : FETs $\mathcal{F}$ to be placed, nets $\mathcal{N}$, cell image
output: Routable placement P which minimizes $\Phi(P)$

```
1  for W_cell := 1, 2, ... do
2      P := PLACECELLFIXEDWIDTH(F, W_cell)
3      if P ≠ null then
4          return P
```

Algorithm 2: PLACECELLFIXEDWIDTH

input : FETs $\mathcal{F}$ to be placed, nets $\mathcal{N}$, cell image, fixed cell width W_{cell}.
output: Routable placement with width W_{cell} which minimizes $\Phi(P)$, or null if no such placement exists.

```
1  P_best ← null
2  Q ← {P_∅}                        // P_∅ denotes the empty placement
3  while Q is not empty do
4      P ← POPNODE(Q)
5      if P is not feasible then
6          continue
7      if P is fully placed then
8          P_best ← BEST(P_best, P)
                   // minimum w.r.t. objective value (cf. Section 3.3)
9      else
10         Q ← Q ∪ PLACENEXTFET(P)
11 return P_best            // will be null if no routable placement exists
```

5.2 Phases

Due to the large complexity of the routing problem, routing oracle calls are very expensive. In many cases placements are illegal even when only a restricted set of rules

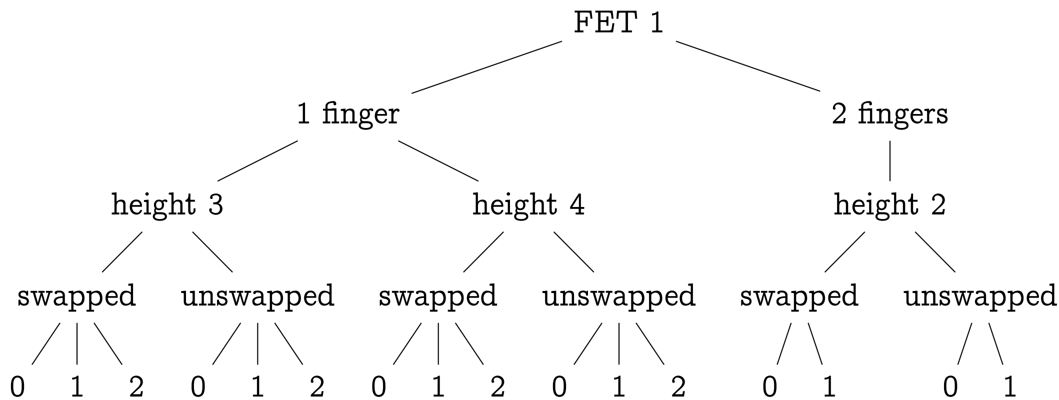


Figure 5.1: Example of a configuration search tree for a single FET. Properties are set in the following order: number of fingers, height in fins, swap status, and x position.

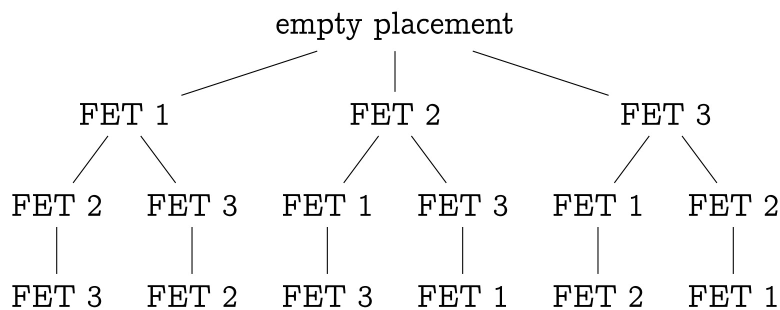


Figure 5.2: FET permutation search tree for 3 FETs. FETs are placed from left to right in all possible permutations and all configurations. In the full placement search tree, permutations of FETs are enumerated for both stacks individually and each of the FET nodes is expanded by the configuration tree (cf. Figure 5.1) for the corresponding FET.

is considered. Checking only this restricted set of rules can speed up the oracle call substantially. We distinguish between three sets of rules, called *phases*:

1. *FEOL* (front-end-of-line). Contains only rules below M0. This includes but is not limited to floating gates, RX coloring, fin trim shapes, and PC trim shapes.

2. *FET-access*. Contains rules up to M0. Some nets can be routed below M0 by FET sharing. For all other nets we know that all their terminals must somehow 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. M0 trim shapes, via coloring, and all FEOL rules are contained in this group.

3. *Full*. The entire routing. Also honoring net connections to user-specified pin tracks and respecting forbidden tracks.

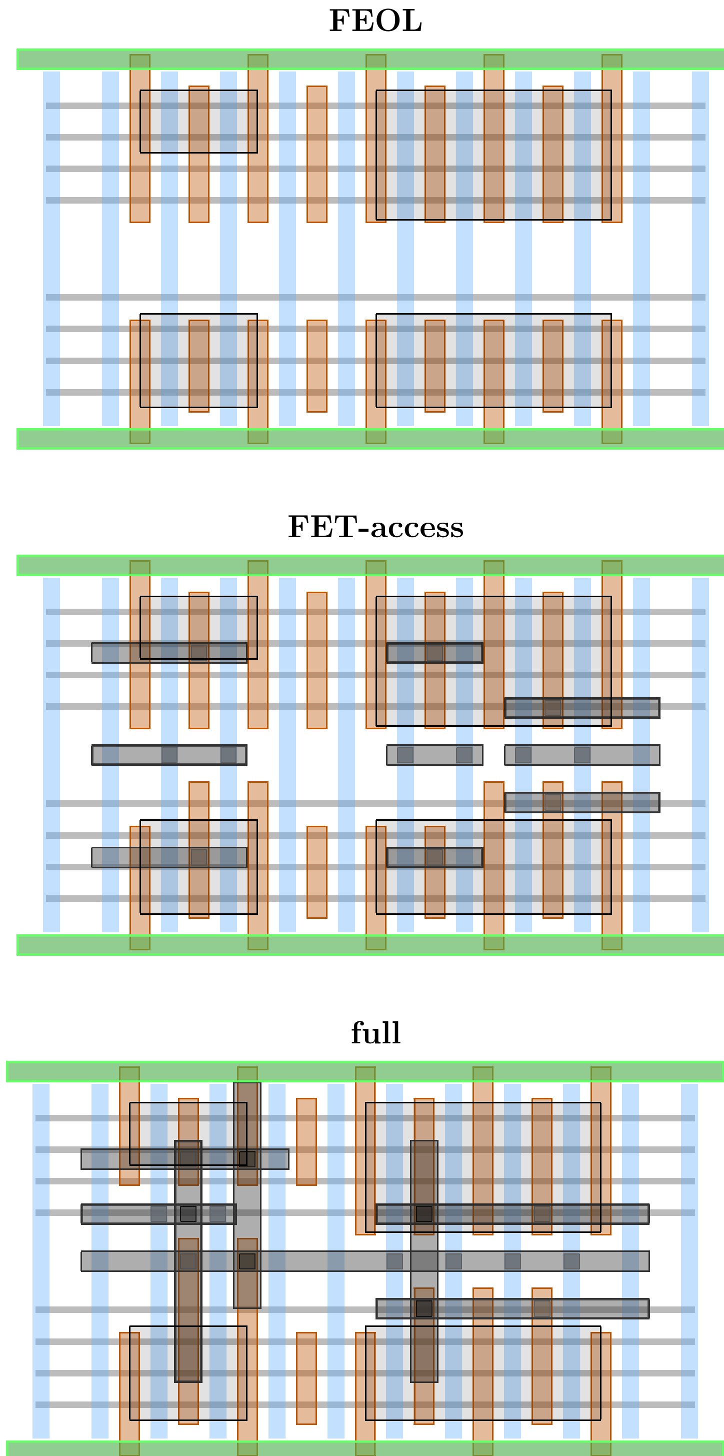


Figure 5.3: Routing solutions of FEOL, FET-access, and full phase. Many relevant layers for the FEOL phase are not shown in the plots. Routings of the FET-access phase connect each FET terminal to M0, but only for nets which cannot be fully connected below M0. The full phase runs an entire routing.

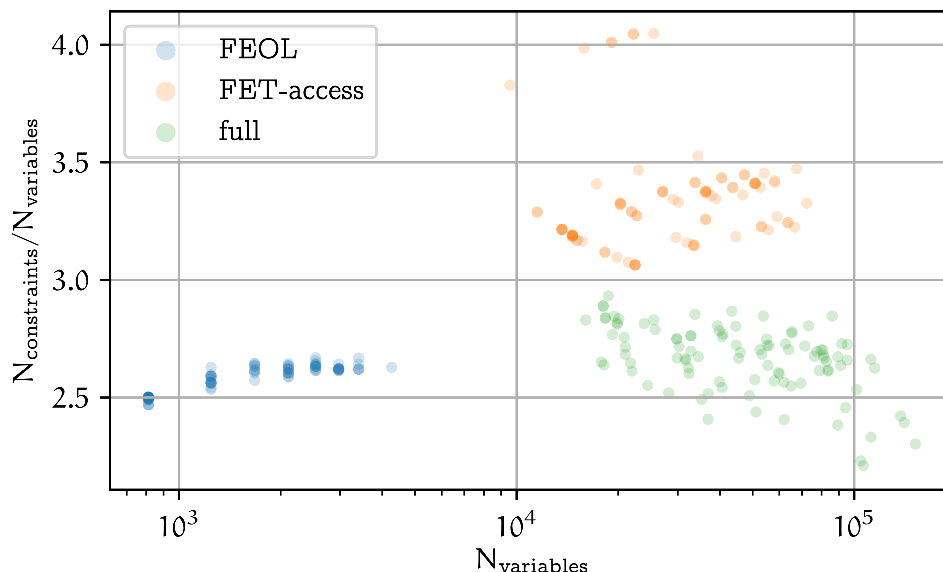


Figure 5.4: Size of MIP for different phases. Each point represents the routing MIP of the final placement of one cell of the LATCH SUBCELLS testbed in a given phase. There are about $2 - 4$ times as many constraints as variables, for all phases and placement sizes. The FEOL phase is about 20 times smaller than FET-access and full. Information about the LATCH SUBCELLS and other testbeds are given in Appendix A.

Figure 5.3 shows routing solutions of the three phases for the same instance. 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. As soon as one phase proves that the placement is illegal, we stop since we know that the full placement is also illegal. Figure 5.4 shows number of variables and constraints of the MIPs for different phases.

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 hours on large instances. For placements which are already illegal w.r.t. FEOL rules, the phase based approach is much faster compared to directly checking full routability. Figure 5.5 shows the runtime distribution of routability oracle calls for STANDARD CELLS and LATCH SUBCELLS. The results contain the routing corridors feature which will be the topic of Section 5.9. Routing corridors artificially restrict the search space of the full phase. Instances which are feasible with routing corridors are guaranteed to be feasible without routing corridors as well. For instances which are infeasible with routing corridors, an additional phase without routing corridors has to decide upon routability. A few interesting things can be observed in Figure 5.5. First, FEOL checks are much faster than FET-access and full routability checks. Furthermore, their runtime is much more consistent. The [25, 75] percentile interval is very narrow and

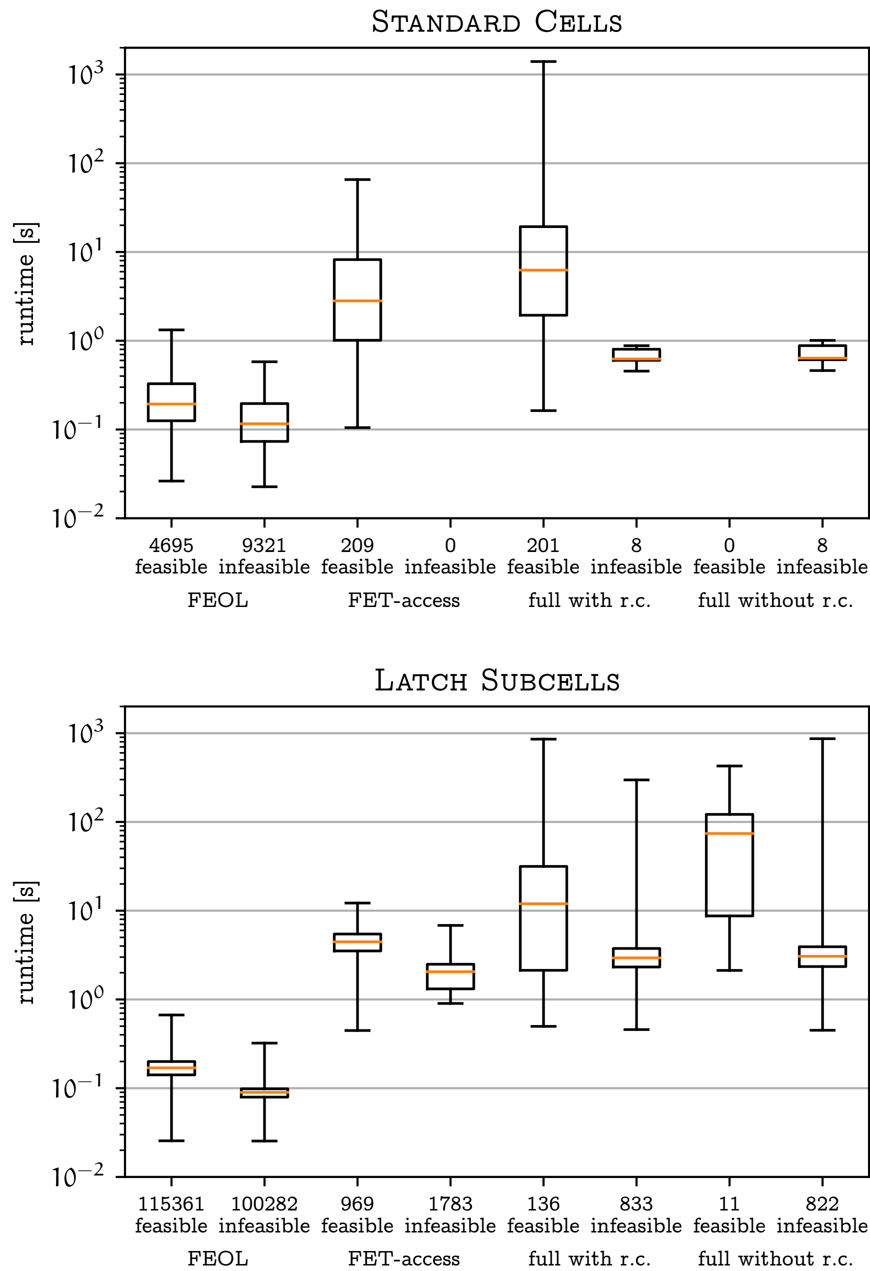


Figure 5.5: Box and whisker plot for runtime distribution of routing oracle calls on Standard Cells and Latch Subcells testbeds. Each of the 8 columns summarizes runtimes for a specific phase (FEOL, FET-access, full with routing corridors, and full without routing corridors) and MIP result (feasible and infeasible). The orange line denotes the median of the runtimes within each category. The lower and upper bound of the box represent 25th and 75th percentile, respectively. The lower and upper whisker represent the minimum and maximum runtime. The respective number of data points is given as x label. Note that there are more feasible FEOL checks than FET-access checks, as FEOL checks are applied on partial placements, whereas FET-access is only checked on fully placed nodes.

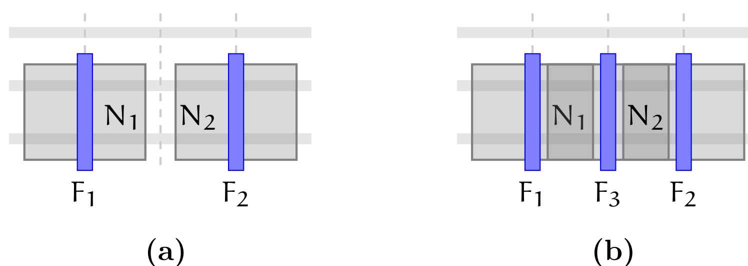


Figure 5.6: Adding a FET to an illegal placement can make it legal. **(a)** FETs F_1 and F_2 are too close to each other. Since their nets N_1 and N_2 are not equal, the gates need to be at least 3 tracks apart from each other. **(b)** Adding FET F_3 with contacts N_1 and N_2 legalizes the situation but also changes the electrical properties.

the minimum and maximum do not deviate from the median as much as they do for the other phases. Proving infeasibility is about twice as fast than finding a feasible solution. For the other phases the results depend on the testbed. In the Standard Cells testbed all FET-access and almost all full instances were feasible. Exactly the opposite is true for Latch Subcells instances, where 65% of the FET-access and 86% of the full instance are infeasible. Runtimes fluctuate by a large amount for these phases. Although only called 209 times, runtimes of the full routability check with routing corridors span 4 orders of magnitude. Runtimes of cells with and without routing phases can be seen in Figures 5.7 and 5.8.

5.3 Routing of Partial Placements

For some instances, most leaves of the placement search tree are unroutable. Many of these instances are unroutable due to a common part of their placement, i.e. there is a common ancestor in the search tree for which all children are unroutable. If we can detect that a node only has unroutable children, the enumeration and feasibility check of those children can be skipped. The key difficulty here is the non-monotonicity of partial placements w.r.t. to routability. Counter-intuitively, a placement can be unroutable but the same placement with an additional placed FET is routable. The same is true for the FET distance rules. Figure 5.6 shows an example where the addition of a FET to an illegal placement yields a legal placement.

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

- *used*, FET placed with configuration c
- *empty*, no FET will be placed here in the full placement
- *unknown*, potentially empty, but some FET might be placed here in the full placement

The routing formulation contains constraints only for cases where the presence or absence of a FET is known but not for the unknown case.

Checking partial placements for routability gives a large speedup compared to simply checking full placements as can be observed in Figures 5.7 and 5.8. These figures also show the speedup achieved by using multiple routing phases.

5.4 FEOL Routing Oracle Cache

In contrast to the previously presented technique, routing oracle caching does not affect the search tree. For the LATCH SUBCELLS testbed, 85% of the routability oracle runtime is spent in the FEOL phase (cf. Figure 5.12 on page 39). Many partial placements look similar from a routability perspective and calling the oracle twice for similar instances can be avoided. Two full routing instances are equivalent if, for example, they can be transformed into each other by a permutation of nets. The routing instances still originate from different placements, but their routability oracle outcome is guaranteed to be the same. During the FEOL phase there is much more redundancy which can be exploited. For example, connectivity information is not required, since we do not want to fully route the nets. This means two placements might give different routability results in the full phase but are both legal in the FEOL phase due to the FEOL rules being less strict.

In cases where we can prove that the FEOL instances are equivalent, a single routing oracle call suffices. To detect these situations, we transform partial placements into a data structure called *FEOL routing instance* which contains all information needed by the FEOL routing oracle but hides other information contained in the placement, like net names and FET names. This routability data is the only input to the FEOL routing algorithm. For example, for each track and stack it contains a bool which denotes whether there exists a FET which must be connected at this position. If two partial placements are transformed into the same routability data we know, by construction, that the routability oracle will give the same answer for both. For every finished call to the routing algorithm, we store the routability data and the oracle result in a cache. Figure 5.11 shows two partial placements which are transformed into the same routability data.

The effectiveness of the FEOL cache is shown in Figures 5.13 and 5.14. Figure 5.12 shows that for LATCH SUBCELLS instances the relative routability check runtime spent in the FEOL phase can be reduced from 85% to 60% by using the FEOL cache.

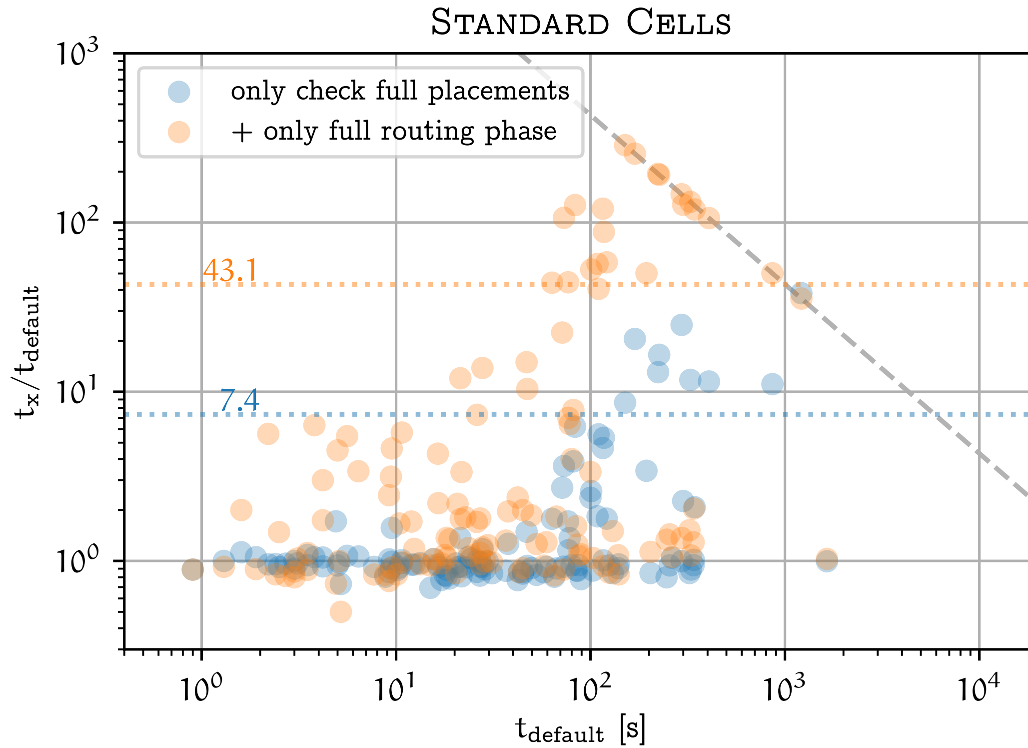


Figure 5.7: Evaluation of runtime benefits due to partial placement checks and routing phases on STANDARD CELLS testbed. Runtime of default mode $t_{default}$, including features presented in the following sections, is shown on the x axis. Relative runtime of default mode without specific feature t_x compared to runtime of default mode is shown on the y axis. For blue markers the deactivated feature is the routability check of partial placements, i.e. only full placements are checked for routability using the routing phases FEOL, FET-access, and full. For orange markers only full placements are checked but only using the full routing phase. The dashed gray line shows points for which t_x hits the runtime limit of 12h. This means that all cells represented on this line would have taken longer, i.e. larger value of t_x, if not aborted. The entire testbed is factor 7.4 faster due to the partial placement checks and additionally factor 5.8 faster due to using three routing phases.

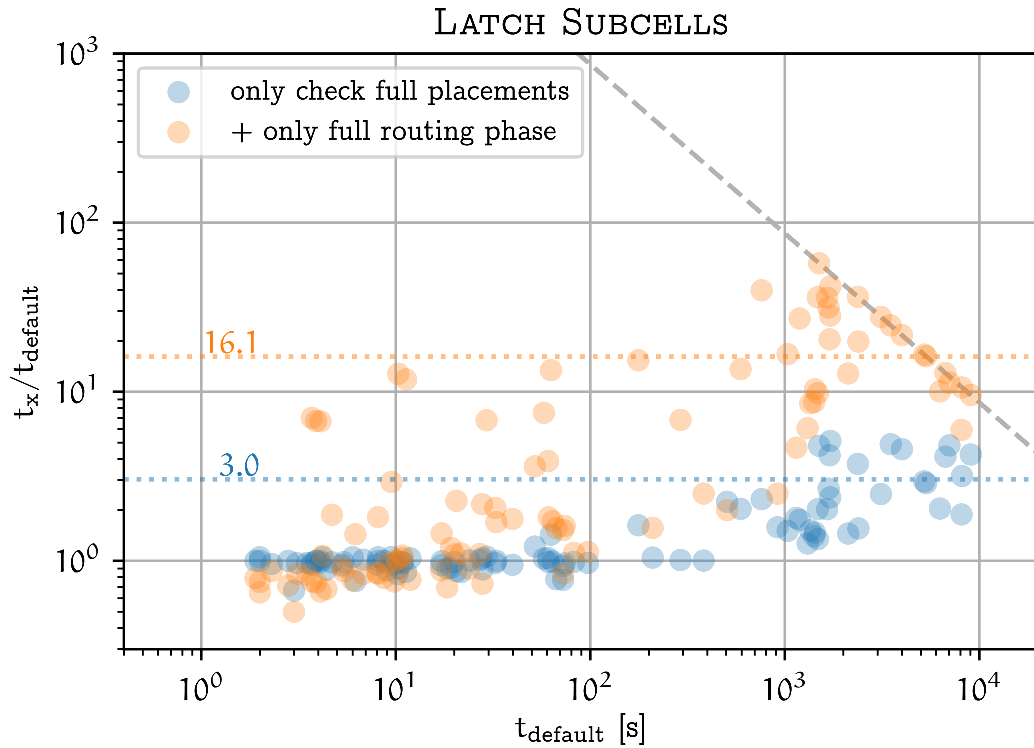


Figure 5.8: Same as Figure 5.7 but for LATCH SUBCELLS testbed. For this testbed, the runtime limit was 24h. Runtime savings are not as large as for STANDARD CELLS instances. Many instances have a larger runtime $t_{default}$, s.t. $t_x/t_{default}$ cannot be very large before the timeout is reached. It is obvious, however, that these features are mandatory to achieve good performance.

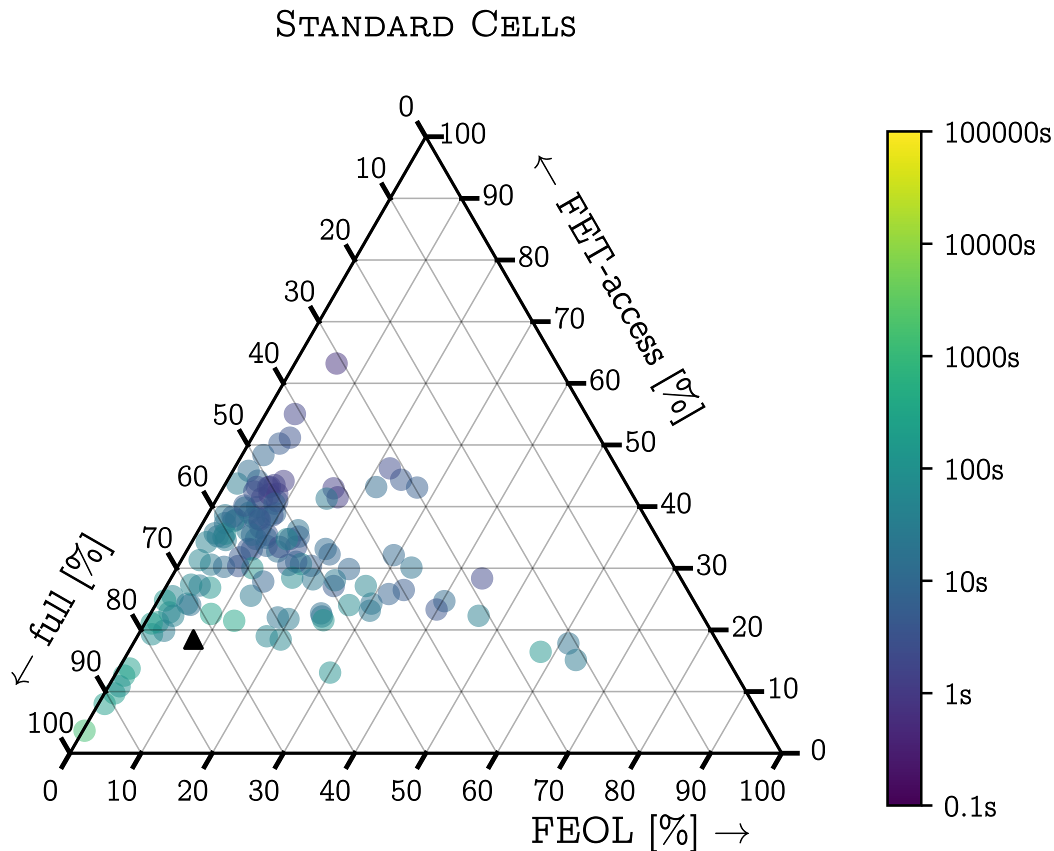


Figure 5.9: Ternary plot of relative runtimes of FEOL, FET-access, and full phase. Each point represents one instance of the Standard Cells testbed and shows which portion of the routability check runtime was spent in which phase. The bottom left corner contains instances with 100% runtime spent in full and 0% in the other two phases. Instances with 100% spent on FET-access would be in the top corner and instances with 100% on FEOL phase in the bottom right corner. Instances with equal portions in each phase are exactly in the center of the triangle. Horizontal lines from bottom to top represent points with 0%, 10%, . . . , 100% relative runtime in the FET-access phase. Most instances use large parts of their runtime in the full phase with small contributions from FET-access and FEOL. There are some outliers with more than 60% FEOL and FET-access runtime though. Colors encode the total routability check runtime used by a cell. There is a tendency for cells with higher runtime to use larger runtime portions in the full phase. The black triangle shows relative phase runtimes for the total runtime of all cells summed up. The plot was created using the python package ternary (Harper et al. 2015).

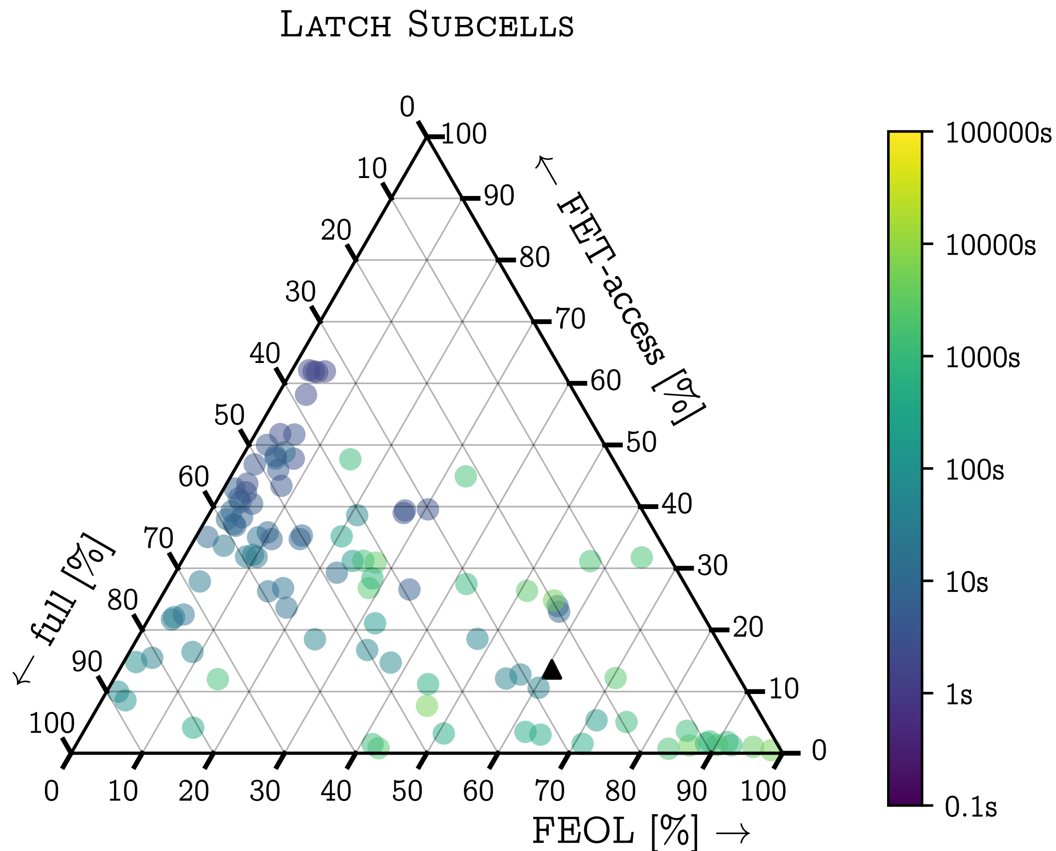


Figure 5.10: Ternary plot of relative MIP runtimes of FEOL, FET-access, and full phase. Same plot as Figure 5.9 but on the Latch Subcells testbed. In comparison to the Standard Cells testbed, many more instances use the majority of their runtime in the FEOL phase, some even more than 90%. The instances are in general much more spread out over the diagram. In contrast to the Standard Cells results, higher routability check runtime correlates with a larger fraction of time spent on the FEOL phase. The relative runtime summed over all cells uses about 60% of its runtime in the FEOL phase.

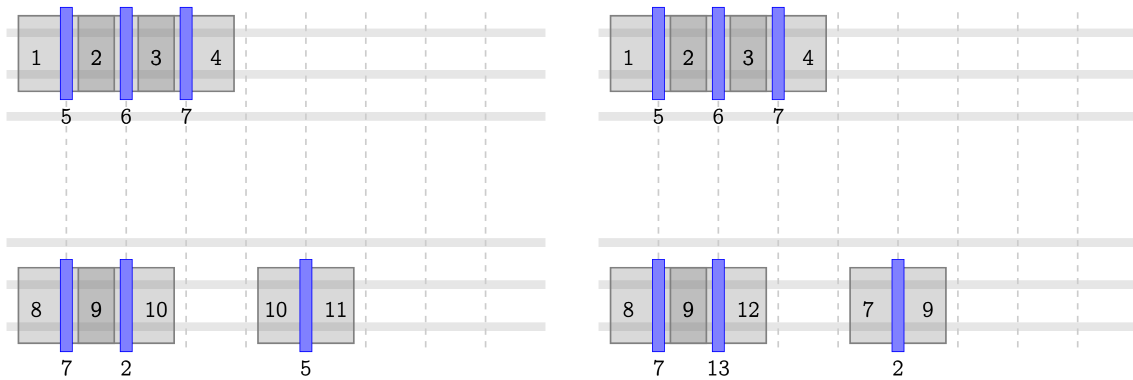


Figure 5.11: Two partial placements with equivalent FEOL cache key. The placements differ in two FETs on the bottom stack. Note that a simple renaming of net names is not sufficient to transform one placement into the other. Both instances can be checked for FEOL routability by a single oracle call.

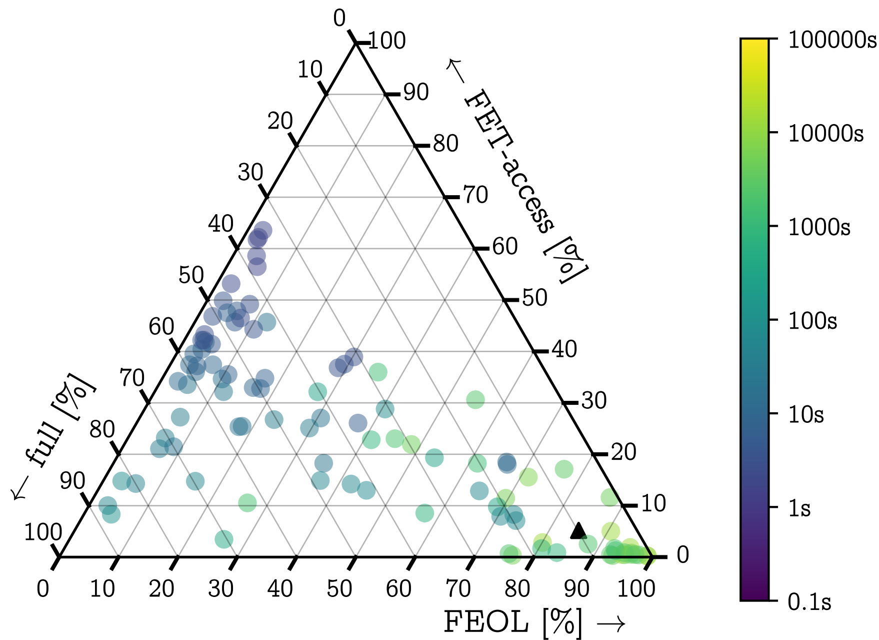


Figure 5.12: Ternary plot of relative MIP runtimes of FEOL, FET-access, and full phase. Same plot as Figure 5.10 but without the FEOL cache presented in Section 5.4. Compared to Figure 5.10 more cells are in the bottom right corner with most of their runtime spent on the FEOL phase, especially for instances with high total runtime. The total average uses 85% in the FEOL phase, compared to 60% with FEOL caching.

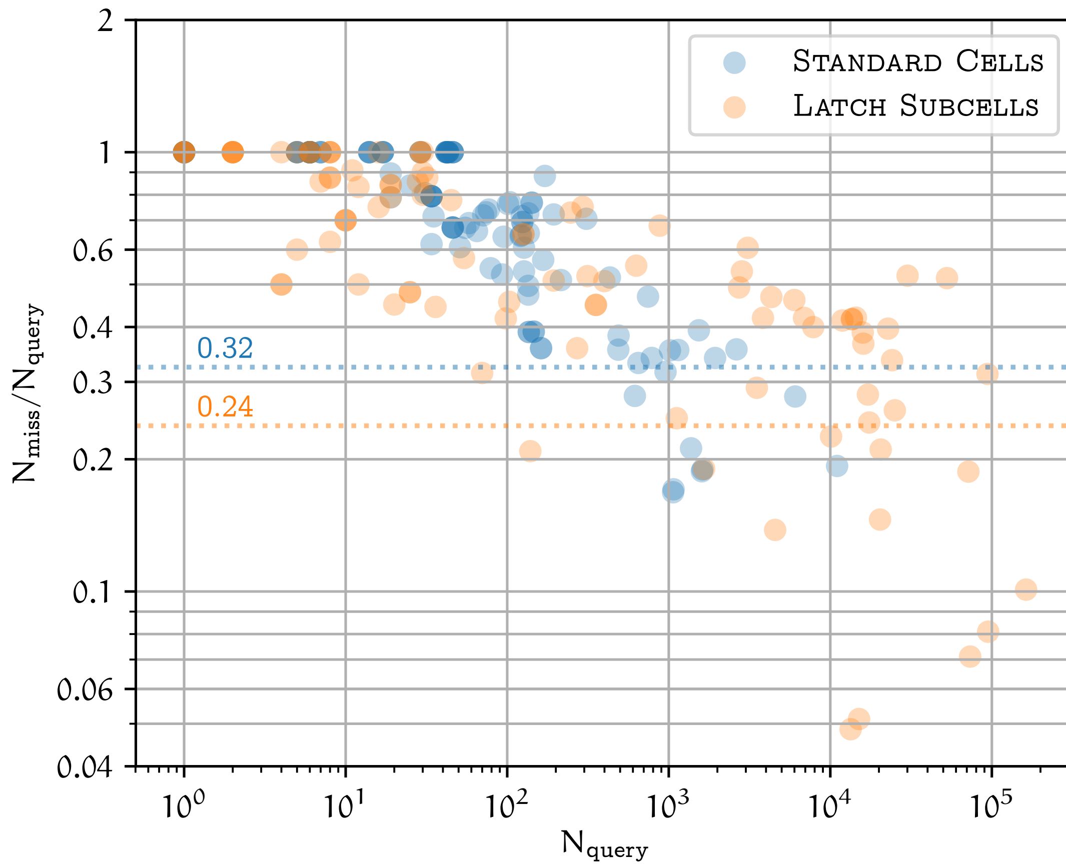


Figure 5.13: FEOL routability oracle cache misses and queries for STANDARD CELLS and LATCH SUBCELLS testbeds. Each point represents one cell. N_{query} denotes the number of cache queries for this cell and N_{miss} the number of cache misses, i.e. instances that were not found in the cache and needed a MIP call. The ratio N_{miss}/N_{query} gives the number of instances that on average need to be solved by the MIP for each FEOL instance. A value of 1 means that the cache did not give any improvements. Cells with only few FEOL instances do not profit from caching. The more FEOL instances are solved, the hotter the cache, i.e. more FEOL cache keys are already present. For two LATCH SUBCELLS instances, only 5% of the queries could not be found in the cache. Dotted lines show the ratio of cache misses and hits summed over all instances of the respective testbed.

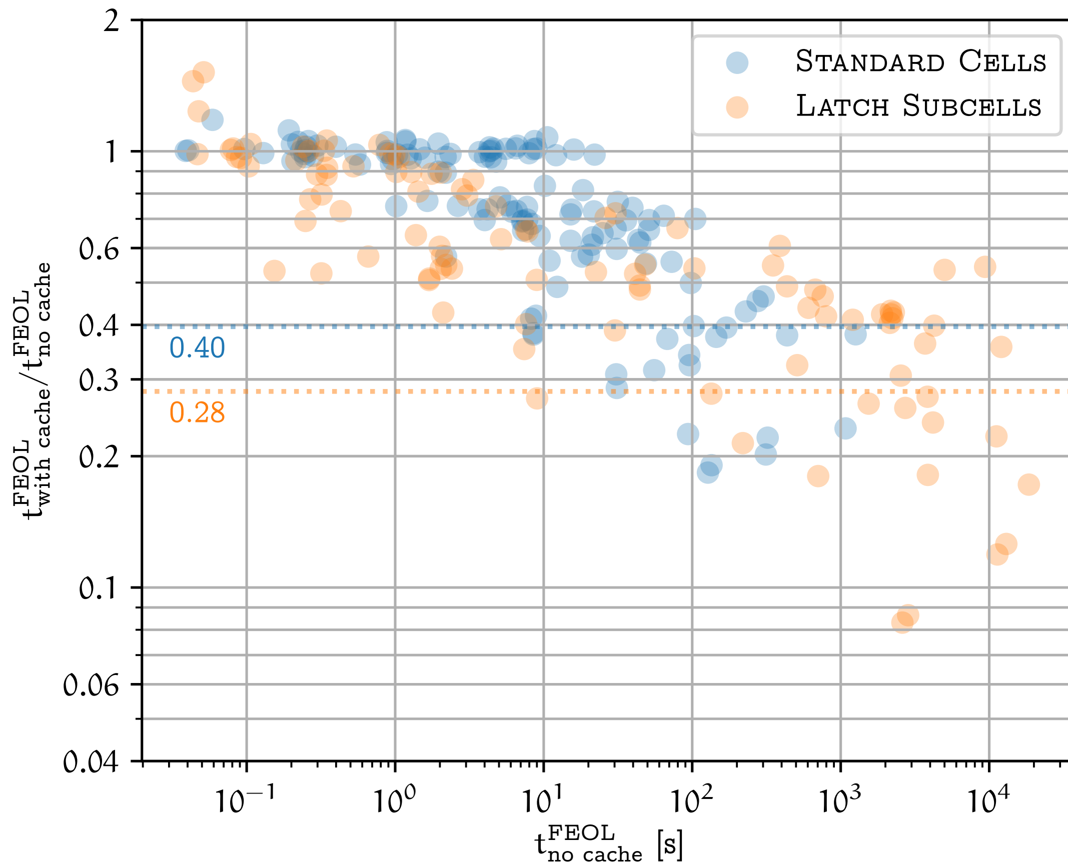


Figure 5.14: Similar plot compared to Figure 5.13. Total runtime for FEOL instances t^{FEOL} is shown with and without caching. If MIP runtime were the only contribution to the total runtime and all instances took the same amount of time for solving, this plot would look exactly the same as Figure 5.13. MIP runtimes are relatively constant over different instances, especially for Latch Subcells instances cf. Figure 5.5. However, MIP solver runtime is not the only contribution to total FEOL runtime. Instance construction takes some amount of runtime. Especially for those instances which are easy to solve by the MIP the relative contribution of instance construction becomes significant. Therefore, runtime improvements are slightly worse than would one would expect from Figure 5.13. Dotted lines show the runtime ratio for total runtime of all instances of the respective testbed. As expected, their values are slightly larger compared to those of Figure 5.13.

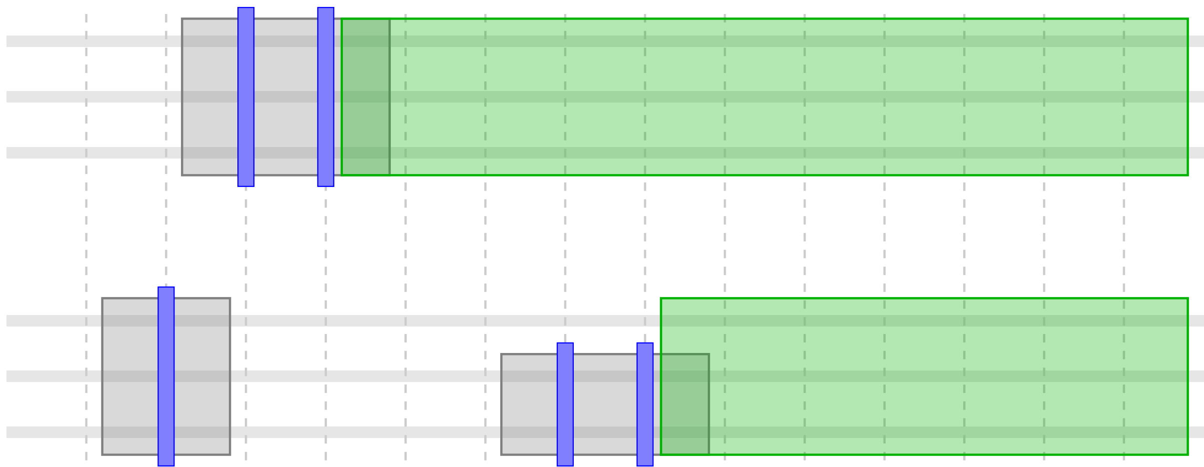


Figure 5.15: Partial placement with three placed FETs. The remaining FETs can only be placed in the green area to the right of the placed FETs.

5.5 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 descendants are either not routable or not optimal. In cell width pruning this means that given some partial placement we prove that for any configuration of the remaining FETs the resulting placement is illegal. Note that at this point we are inside the cell width loop (cf. Algorithm 1 on page 28). Therefore, the cell width is already fixed, i.e. all FETs must be placed in 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 as illustrated in Figure 5.15. 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. Routability will be checked at a different point of the algorithm.

5.5.1 Combinatorial Approach

Our cell width pruning is based on Euler chains which have already been used in previous work on automated cell design. Uehara and Cleemput 1981 introduced them in their seminal work on placement of dual cells using the 1-D layout style. Weyd 2011 discusses many variants of the placement problem for non-dual cells and analyzes its complexity for different restricted versions of the problem. The most general case is not

discussed but most ideas from the restricted cases can be applied. We will focus on the most general case directly since it is the one which is relevant in practice. Compared to previous work our contribution is an algorithm which handles non-dual cells, allows gaps between FETs, and can be used in practice. As the problem is NP-hard it does not have polynomial worst case runtime but we show that the runtime is negligible in practice.

Placement Restrictions

As we will see, the Euler chains method requires restrictions on the search space of all possible configurations of the FETs which are not placed yet. 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 5.1. 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 5.2. Given a placement C and a restriction R, we say that $C = (c_1, \dots, c_n)$ obeys $R = (r_1, \dots, r_n)$, if c_i obeys r_i for all i. The set of all legal placements obeying R is defined as $\mathcal{C}(R) := \{C \mid C \text{ legal, obeying } R\}$.

Note that for odd number of fingers, the swap status is not controlled by a restriction. The reason for this will become apparent later. The outer loop of the algorithm described above is shown in Algorithm 3 (CellWidthPruning). The function Restrictions($\mathcal{F}$) returns the set of all possible restrictions R s.t. a placement C obeying R exists. The function MinWidth($\mathcal{F}, C_L, R$) 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($\mathcal{F}, C_L, R$) for $C_L = \text{null}$. $C_L = \text{null}$ means that no FET has been placed on this stack yet. This algorithm can then easily be extended for $C_L \neq \text{null}$.

Algorithm 3: CellWidthPruning

input : FETs $\mathcal{F}$ to be placed, fixed leftmost FET C_L (null if no FET is fixed), maximum allowed placement width W_{max}

output: Does a placement with width at most W_{max} exist?

1 **for** $R \in$ Restrictions($\mathcal{F}$) **do**
2   **if** MinWidth($\mathcal{F}, C_L, R$) $\leqslant W_{max}$ **then**
3   **return** true
4 **return** false

Minimum Width for Restricted Placements

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 5.3. The width of a placement $C = (c_1, \dots, c_n)$ is defined as

$$W(C) := \max_i (x(c_i) + f(c_i)) - \min_i x(c_i)$$

This allows us to formulate our central lemma.

Lemma 5.4. *Let* R *be a placement restriction. Then,*

$$\min_{C \in \mathcal{C}(R)} W(C) = \min_{C \in \mathcal{C}(R)} \left(\sum_{i=1}^{n} f(c_i) + 2N_{\textit{no-share}}(C) \right), \tag{5.1}$$

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

Proof. "$\leqslant$": 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

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

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

$$\begin{aligned} W(C) &= x(c_n) + f(c_n) - x(c_0) \\ &= \sum_{i=1}^{n-1} (x(c_{i+1}) - x(c_i)) + f(c_n) \\ &= \sum_{i=1}^{n-1} (G(c_i, c_{i+1}) + f(c_i)) + f(c_n) \\ &= \sum_{i=1}^{n} f(c_i) + \sum_{i=1}^{n-1} G(c_i, c_{i+1}) \end{aligned}$$

We have $G(c_i, c_{i+1}) \geqslant 0$ if F_i and F_{i+1} are allowed to share and $G(c_i, c_{i+1}) \geqslant 2$ otherwise. Therefore,

$$W(C) \geqslant \sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C) \tag{5.2}$$

From Inequality (5.2) we immediately get

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

"$\leqslant$": Equality in Inequality (5.2) 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 $\sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C)$. 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

$$\begin{aligned} \min_{C \in \mathcal{C}(R)} \left(\sum_{i=1}^{n} f(c_i) + 2N_{\text{no-share}}(C) \right) &= W(C') \\ &\geqslant \min_{C \in \mathcal{C}(R)} W(C), \end{aligned}$$

which concludes the proof. □

The left hand side of Equation (5.1) is $\textsc{MinWidth}(\mathcal{F}, C_L, R)$ for $C_L = \text{null}$. Note that the first summand 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, which is possible since the height of the FETs is fixed by the restriction. There needs to be a gap between two FETs of different groups, which means that the number of gaps is minimized if

all FETs within one group are placed directly next to each other. We therefore get $N_{group} - 1$ gaps between the groups, where N_{group} is the number of groups.

Within each group two FETs can share if they have the same source or drain net, cf. Figure 3.4 on page 13. For FETs with an even number of fingers the outward facing nets are equal on both sides. If the FET is not swapped it is the source net on both sides, if the FET is swapped it is the drain net. Since the swap status is fixed for all FETs with an even number of fingers, it is known which nets belong to the outward contacts and we know which pairs of FETs are allowed to overlap. The remaining part of the placement configuration space which is not restricted are the relative x positions of the FETs and the swap status of FETs with an odd number of fingers. In the following we present a linear time algorithm which determines the minimum number of needed gaps.

Walk Partitions

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 the *sharing graph* $G = (V, E)$, where V corresponds to the set of nets and E to the set of FETs. For each FET with an odd number of fingers, we have 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 have a loop $e = \{v, v\}$, where v is the net which belongs to the leftmost and rightmost contact of the FET. An illustration of the sharing graph is given in Figure 5.16. The idea of the sharing 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 of a subset of FETs without gaps a *chain*.

Definition 5.5. Let G be a graph. We denote by $P(G)$ the size of a minimum partition of the edges of G into walks.

Theorem 5.6. *Let F be a non-empty set of FETs with equal VT level, equal height, fixed number of fingers, and fixed swap status for FETs with an even number of fingers. For any placement C, let $C_{|F}$ be the sub placement of C consisting of the FETs F. Then given a placement C it holds that*

$$\min_{C \in \mathcal{C}(R)} N_{\textit{no-share}}(C_{|F}) = P(G(F)) - 1\,.$$

Proof. "$\leqslant$": Let $k := P(G(F))$, and $P_1, \ldots, P_k$ 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.

"$\geqslant$": 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 placement yields a partition of G into $N_{\text{no-share}}(C_{|F}) + 1$ walks. $\square$

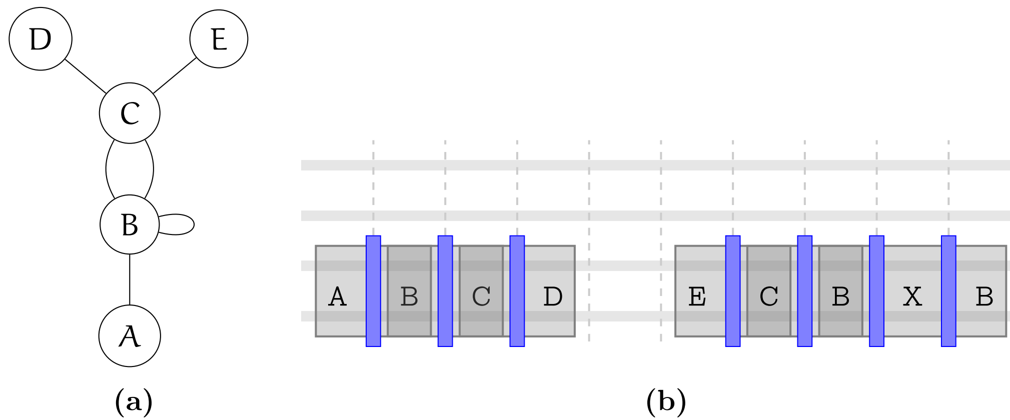


Figure 5.16: **(a)** Sharing graph with 5 nodes corresponding to nets (A, B, C, D, E) and 5 edges, corresponding to FETs with an odd number of fingers and 1 loop corresponding to a FET with an even number of fingers. **(b)** Placement with minimum number of gaps. This corresponds to the walks A, B, C, 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.

The size $P(G)$ 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. This result can easily be extended.

Theorem 5.7. *Let G be a connected graph and N_{odd} the number of vertices with odd degree. If G has no edges, i.e. G is the graph consisting of a single unconnected vertex, then $P(G) = 0$. Otherwise,*

$$P(G) := \max\left(\frac{N_{odd}}{2}, 1\right).$$

To obtain $P(G)$ for a potentially unconnected graph G, one has to sum the number of walks for each connected component.

Corollary 5.8. *Let G be a connected graph and $C_1, \ldots, C_k$ its connected components. Then*

$$P(G) = \sum_{i=1}^{k} P(C_i).$$

The connected components of a graph can be computed in linear time. Therefore, using Theorem 5.6 and Corollary 5.8 we are able to compute the minimum number of gaps in a placement restricted by R in linear time.

Algorithm 3 has to iterate over all restrictions in order to decide whether a placement with width at most W_{max} exists. In practice, we only consider restrictions R for which $\sum_i f(r_i) \leqslant W_{max}$. This means that we start with the minimum number of fingers for each FET and incrementally distribute additional fingers. For some instances

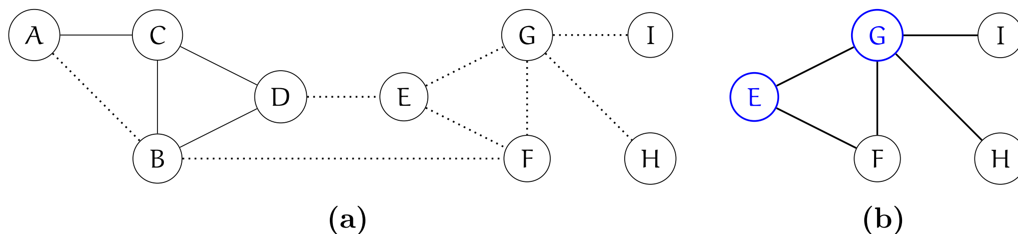


Figure 5.17: **(a)** Generalized sharing graph with 9 vertices (nets) and 3 edges for FETs with an odd number of fingers (E_{odd}, solid edges) and 8 edges for FETs with an even number of fingers (E'_{even}, dotted edges). **(b)** Corresponding vertex cover graph. A minimum vertex cover of size 2 is denoted by blue vertices.

a large number of FETs are restricted to an even number of fingers and iterating over all swap states for these FETs becomes a runtime bottleneck. In the following, we show how the swap states of these FETs can be chosen more efficiently.

Choosing Swap States Efficiently

Given a restriction R, the edges of the sharing graph of R can be divided into two sets E_{odd} and E_{even}, where E_{odd} (E_{even}) consists of the edges corresponding to odd (even) finger restrictions. We notice that all edges in E_{even} are loops and therefore their presence does not influence the parity of the vertex degree. Changing the swap state of a FET restriction with an even number of fingers removes a loop at one vertex (e.g. corresponding to the FETs source net) and adds a loop at another vertex (e.g. corresponding to the FETs drain net). This changes neither N_{odd} in Theorem 5.7, nor the vertex sets of connected components in Corollary 5.8. However, it might change whether a connected component consisting of a single vertex has any loops or not. If the minimum width of a restriction R_1 is guaranteed to be no larger than the minimum width of a restriction R_2, there is no need to iterate over R_2 in Algorithm 3.

Definition 5.9. An *undecided swap placement restriction* is a placement restriction without specifying the swap status for even numbers of fingers. A placement restriction is consistent with an undecided swap placement restriction if it specifies the same number of fingers and FET height for each FET.

We construct the *generalized sharing graph* $G = (V, E)$ of an undecided swap placement restriction as follows. Similar to the sharing graph, let the vertex set V correspond to the nets. For each FET restricted to an odd number of fingers we have an edge connecting the vertices corresponding to source and drain. This gives the edge set E_{odd} as for the sharing graph. However, unlike for the sharing graph, for each FET restricted to an even number of fingers we have an edge connecting the vertices corresponding to source and drain as well. This gives the edge set E'_{even}. An example for a generalized sharing graph is shown in Figure 5.17(a).

Definition 5.10. Let R_U be an undecided swap placement restriction and R a consistent placement restriction. Then we say the generalized sharing graph of R_U is consistent with the sharing graph of R.

Given an undecided swap placement restriction, we construct a consistent placement restriction which minimizes its corresponding minimum width over all consistent placement restrictions. In terms of graphs this means that given a generalized sharing graph, we construct a consistent sharing graph G with minimum $P(G)$. Consistent sharing graphs are obtained from generalized sharing graphs by replacing each edge $(v, w) \in E'_{\text{even}}$ by the loop (v, v) or (w, w). We denote the vertices of the non-empty connected components of the graph (V, E_{odd}) by $V_{>0}$. Adding a loop to an existing connected component of (V, E_{odd}) does not increase the size of a minimum walk partition. Therefore, we can greedily replace all edges in E'_{even} which are incident to $V_{>0}$ by loops adjacent to the non-empty connected components. The remaining edges in E'_{even} connect empty connected components of (V, E_{odd}). Each vertex which gets assigned at least one additional loop will result in an additional non-empty connected component. We want to minimize the number of additional non-empty connected components, which is equivalent to the VERTEX COVER problem on the graph $(V \setminus V_{>0}, E'_{\text{even}})$. Figure 5.17(b) gives an illustration of the VERTEX COVER problem resulting from the generalized sharing graph in Figure 5.17(a).

Although the VERTEX COVER problem is NP-hard (Karp 1972), this approach still results in a runtime improvement compared to the brute force approach. Given an undecided swap placement restriction the brute force approach would run the linear time minimum width computation for each of the $2^{|E'_{\text{even}}|}$ different consistent placement restrictions. With the VERTEX COVER approach, we only have one minimum width computation and need to solve one VERTEX COVER instance on a graph which often has fewer than $|E'_{\text{even}}|$ edges. In practice the VERTEX COVER instances are very tiny s.t. a full enumeration of possible cover sets with increasing cardinality is fast enough. Figure 5.18 shows the size of vertex cover instances in practice.

Combined, these techniques make our cell width pruning so fast that for most instances its running time is negligible compared to other parts of the algorithm like routability checks (cf. Figure 5.19). The resulting speed up of the placement algorithm is very large, especially for LATCH SUBCELLS instances. Figure 5.20 shows the effects of cell width pruning on the number of search tree nodes as well as resulting placement runtimes.

5.5.2 MIP Approach

The Euler chains method presented above works well if both stacks can be placed independently of each other. In practice, however, design rules on the FEOL layers forbid certain placements which would be seen as legal for both stacks individually. Efficiently incorporating these additional constraints into the Euler chains method seems to be difficult, at least if tried in an technology independent manner. An alterna-

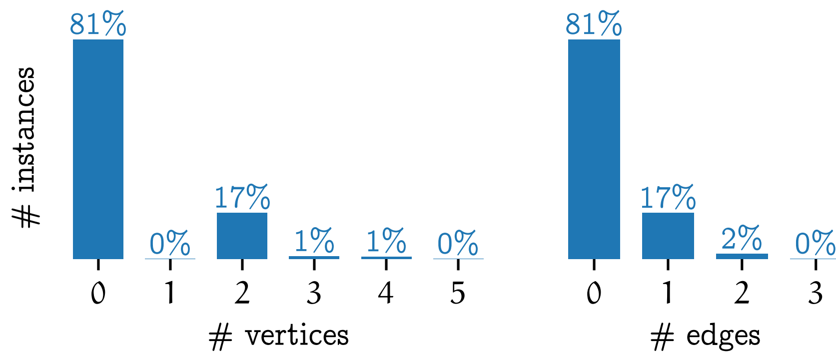


Figure 5.18: Vertex cover instances are very small in practice and can be solved by full subset enumeration. Only instances with up to 5 vertices and up to 3 edges have been observed on the STANDARD CELLS and LATCH SUBCELLS testbeds. Slightly larger instances have been observed when trying to solve small latches without big cell modes but even there the largest instances have at most 10 edges.

tive approach is to model the placement problem in a MIP formulation. Additional constraints can easily be added and adapted to new technologies.

The main motivation for improving the lower bounds are instances with a small number of large FETs. Figure 5.21 illustrates an example with 6 FETs. Three of these, one on the P stack and two on the N stack, cannot be placed opposite of each other if built with maximum height due to the FEOL rules. The width of the placement is therefore dominated by the sum of the width of both of these FETs and much larger compared to an independent placement of both stacks. The Euler chains method underestimates the width of the placement which leads to checking many placement nodes with the two large FETs being illegally placed opposite of each other.

The MIP approach checks the following additional rules. Two FETs placed opposite of each other with different gate nets are unroutable if

7 track image

- they both have at least 3 fingers and none of their contact nets are connected to power, or
- any of them have a height of 3 fins.

9 track image

- they both have at least 3 fingers and none of their contact nets are connected to power, or
- any of them have a height of 4 fins.

12 track image

- no additional rules are checked.

These rules are derived from design rules from other layers, mainly CT. Unit tests ensure that these rules stay valid even if design rules change.

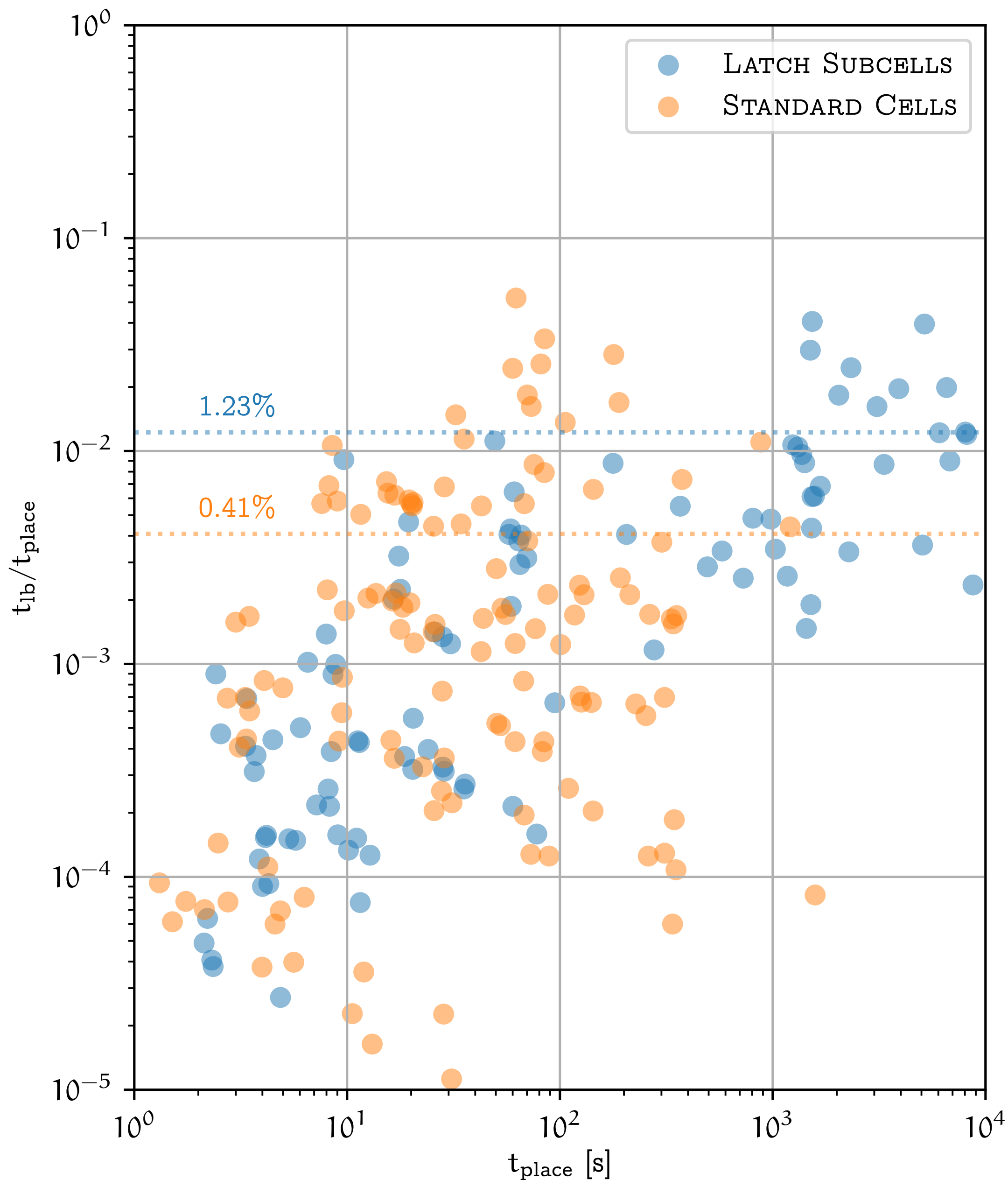


Figure 5.19: Runtime of combinatorial lower bounds for the placement width t_{lb} relative to total placement runtime t_{place} on Standard Cells and Latch Subcells testbeds. MIP lower bounds (Section 5.5.2) are not used here. For most instances the runtime of the lower bound calculation is negligible compared to the total placement runtime and for every instances it takes less than 10% of the total placement runtime. There is a trend that instances with larger placement runtime use a higher fraction of this runtime for lower bound calculation but there is still some gap before the lower bound runtime could become the runtime bottleneck if this trend continues. The dotted lines show the relative runtimes of lower bounds for all testbed runtimes summed up.

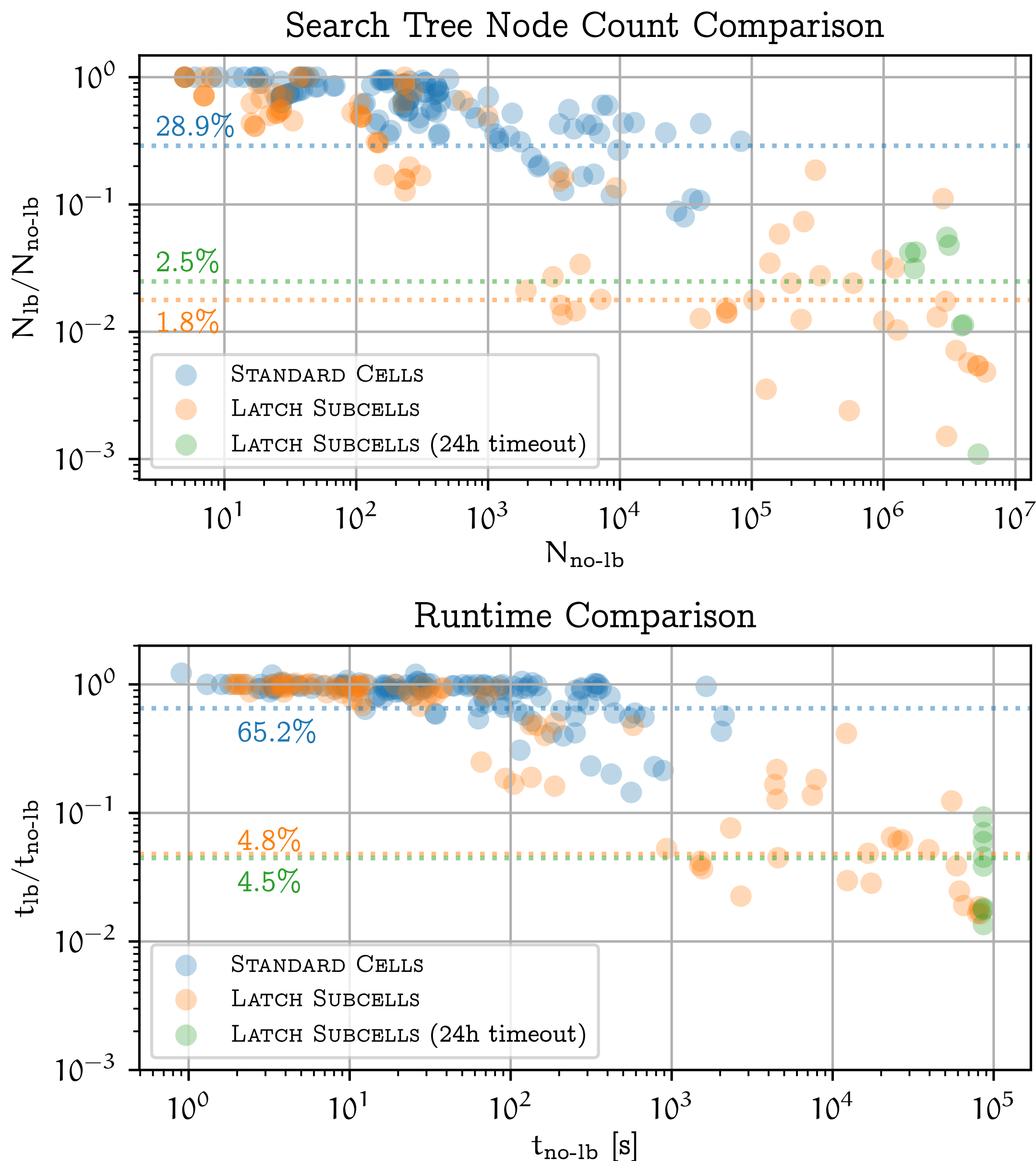


Figure 5.20: Pruning with lower bound on placement width is very effective for larger instances. Results are shown for Standard Cells and Latch Subcells testbeds. N_{lb} denotes the number of nodes in the placement search tree that are considered with active lower bounds. $N_{no\text{-}lb}$ measures the same number when lower bounds are not used. t_{lb} and $t_{no\text{-}lb}$ measure the respective total placement runtimes, including lower bound calculation for t_{lb}. Reduction of the number of search tree nodes translates into shorter placement runtimes but not linearly. Both figures show the same y range to allow for a comparison. For instances with more than 10.000 nodes more than 50%, in many cases roughly 95%, can be pruned by cell width lower bounds. Again, the runtime reduction is slightly less. Some Latch Subcells instances run into a 24h timeout but only if lower bounds were not used. Those are shown in green. The dotted lines show relative number of nodes and relative runtime for all testbed cells combined.

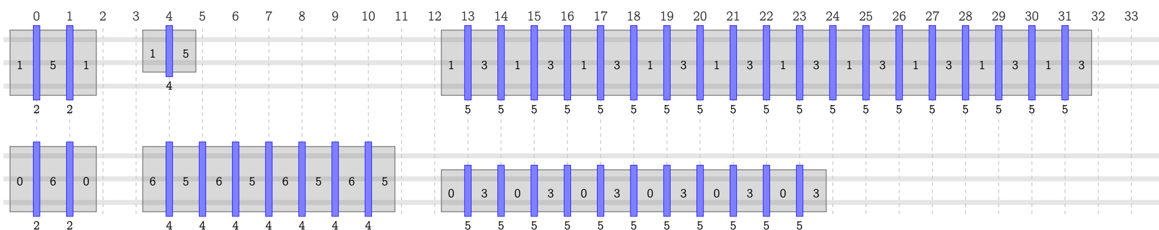


Figure 5.21: Instance with 6 FETs whose smallest placement is substantially larger than the lower bound on placement width for each stack individually. Note that the PFET (upper stack) with gate net 5, if built with height 3, cannot be placed opposite of the NFETs with gate nets 2 and 4.

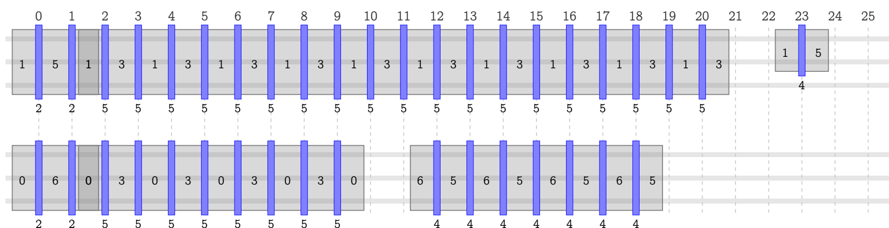


Figure 5.22: Illegal placement of instance of Figure 5.21. This placement is a valid solution with respect to the Euler walks lower bounds which assumes that both stacks can be placed independently of each other. The resulting cell is 8 tracks smaller than the smallest legal layout.

MIP lb. for $\leqslant \ldots$ FETs	0	4	all (6)
number of nodes	599601[a]	244997	244889
$\leqslant$ 30 tracks (runtime) [min]	562	2.2	1.9
$\leqslant$ 32 tracks (runtime) [min]	-	60.6	60.6
34 tracks first sol. (runtime) [min]	-	65.4	64.8
34 tracks proven opt. sol. [min]	-	396	41

Table 5.1: Runtimes and number of visited placement nodes for combinatorial lower bounds on placement width and MIP lower bounds on placement width. Timeout was set to 600min, at which point the algorithm without MIP lower bounds tried to find a solution with 32 tracks. For proving that no solution with $\leqslant$ 30 tracks exists, a factor $\sim$ 300 speedup has been achieved by using the MIP lower bounds.

[a]Number of nodes visited after 600min runtime.

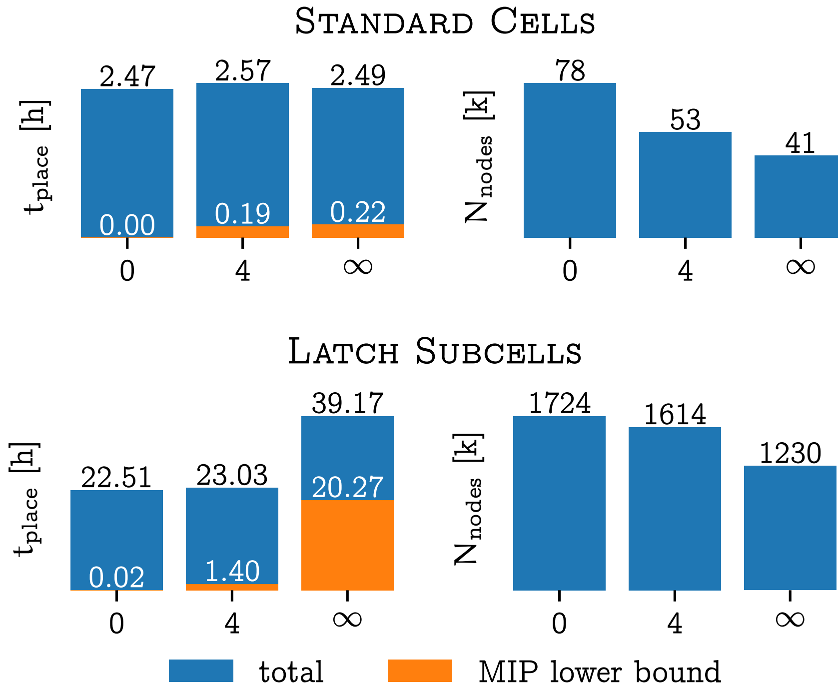


Figure 5.23: Effect of MIP lower bounds on placement width on Standard Cells and Latch Subcells testbeds. The MIP lower bounds are only used if the partial placement has at most k unplaced FETs. Otherwise, the combinatorial lower bounds are used instead. Results for $k = 0, 4, \infty$ are compared against each other, where ∞ means that the MIP lower bounds are always used. The number of search tree nodes N_{nodes} can be reduced by MIP width lower bounds, especially for Standard Cells, but this does not yield an improved total runtime.

MIP formulation

The MIP formulation contains an integer variable x^F_{pos} for each FET F which denotes the position of F. Furthermore there are integer variables for the width, height and swap status of each FET. Constraints make sure that only placements without illegal overlaps are considered. Additionally, there are constraints which model the rules given above. The lower bounds are calculated in the setting where a partial placement is present and some FETs still have to be placed. A specific FET F_L is chosen which is to be placed next. The goal is to determine the rightmost possible position for this FET s.t. the other FETs can still be placed to its right. Therefore, the objective function is to maximize $x^{F_L}_{pos}$, the x coordinate of F_L. See Klotz 2018 for a detailed description of the MIP formulation. The basic formulation can be improved upon by adding MIP cuts which are not needed for correctness of the formulation but exclude fractional solutions from the LP relaxation, resulting in lower MIP solver runtimes. Klotz 2018 also gives a detailed description which cuts have been added and how this affects the runtime.

Overall the MIP lower bounds on placement width have not been very successful in practice. Figure 5.23 shows that for the STANDARD CELLS and LATCH SUBCELLS instances, which are all 9 track, the number of visited placement nodes can be reduced to almost 50% of the number needed with the combinatorial lower bounds. For the STANDARD CELLS, this does not significantly reduce the runtime. For the latch instances this even significantly increases the total running time. The situation is different for the cell mentioned in the motivation earlier (Figure 5.21). Within a runtime limit of 10h, the cell can only be solved using the MIP lower bounds. Table 5.1 summarizes the results.

5.6 Netlength Pruning

Once a routable placement has been found, the algorithm continues to search for placements with a better objective value. Since the cell width is fixed at this point, the only criterion is weighted bounding-box netlength. To reduce the number of nodes which need to be checked checked before the best solution found is proven to be an optimum solution, we calculate lower bounds on the weighted bounding-box netlength for partial placements. In this way we are able to prune parts of the search tree which might be routable but would be worse solutions compared to the one we have already found.

Definition 5.11. Given a placement C and net N, let $T(N, C)$ denote the set of x positions of the source, drain and gate terminals of net N in the placement C. Given a weight w_N for each net N, the weighted bounding-box netlength $L_{BB}(C)$ of a placement C is defined as

$$L_{BB}(C) := \sum_{N \in \mathcal{N}_{\text{signal}}} w_N(\max(T(N, C)) - \min(T(N, C))),$$

where $\mathcal{N}_{\text{signal}}$ is the set of signal nets, i.e. all nets except the power nets.

FAST-LB is an easy to compute lower bound for L_{BB} which estimates $T(N, C)$ for partial placements. It considers the x coordinates of already present terminals and assigns all not yet placed terminals to their leftmost possible position. Nets for which no terminal has yet been placed do not contribute to the lower bound. MIP-LB is a more sophisticated lower bound which estimates the additional netlength of yet unplaced FETs more precisely. The placement MIP in Section 5.5.2 is also used here to determine stronger lower bounds on L_{BB}. However, calculation of these stronger lower bounds is relatively slow. To achieve a good trade off between the saved runtime by pruning parts of the search tree and runtime spent by calculation of improved lower bounds, the MIP lower bounds are only calculated if at most 4 FETs remain unplaced. Figures 5.24 to 5.27 show how many placement nodes can be saved with the lower bounds and how this affects total placement runtime. One can see that the stronger MIP based lower bounds achieve additional pruning of nodes. However, the runtime saved by pruning only just compensates for the additional runtime cost of computing these lower bounds.

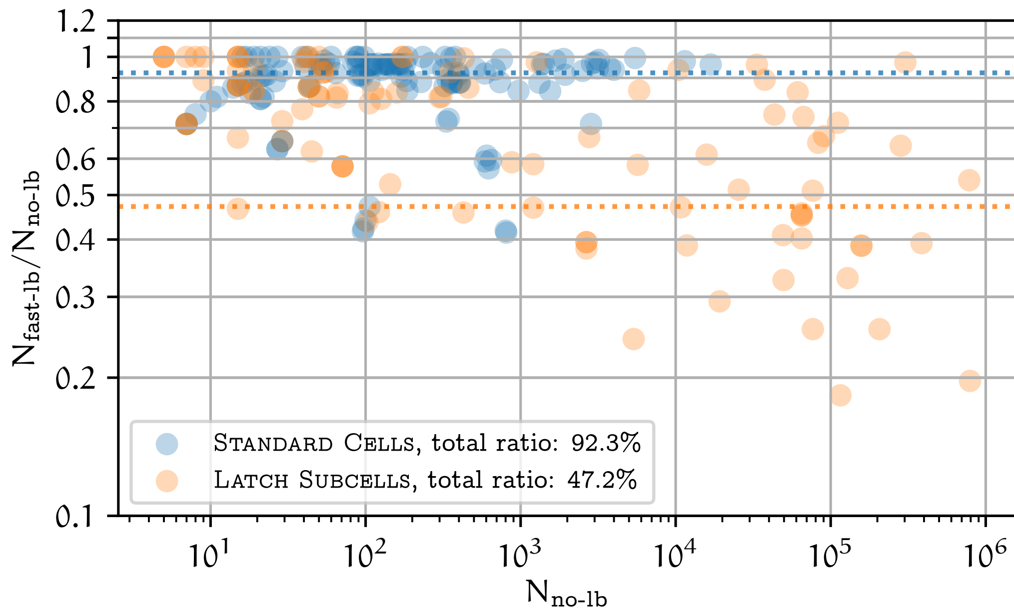


Figure 5.24: Effect of Fast-lb on the number of placement nodes. The number of nodes without lower bounds is shown on the x axis. The relative number of nodes needed with Fast-lb is shown on the y axis. Node savings get more significant for large instances with a high number of nodes. The dotted lines show the ratio over total counts. The exact ratio value is given in the legend.

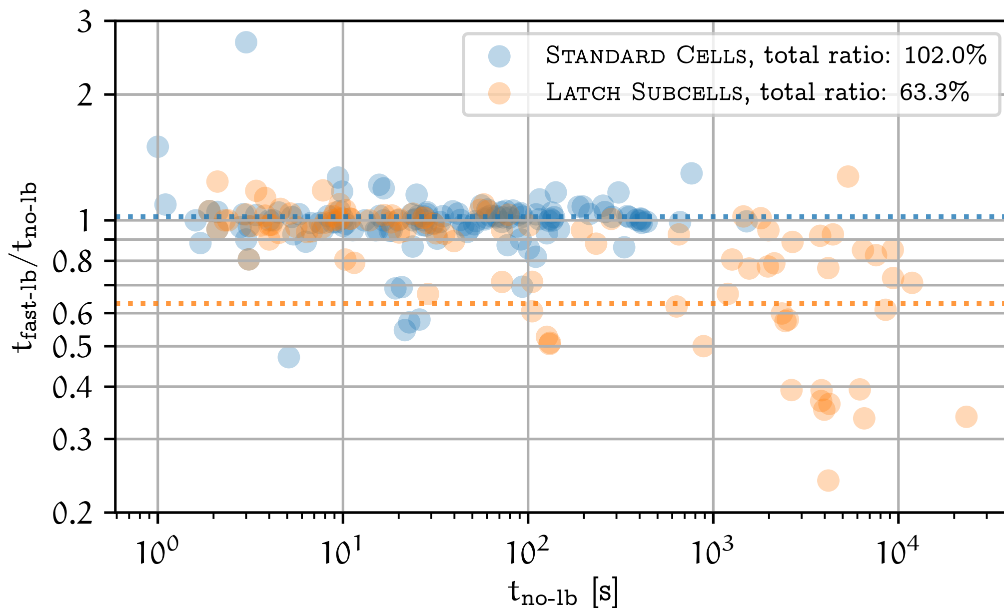


Figure 5.25: Runtime comparison of Fast-lb vs. no lower bounds, similar to Figure 5.24. Relative runtimes can be above 1 due to the additional cost of lower bound computation and, especially for small instances, fluctuations in runtime measurement. Improvements are slightly less distinct compared to node savings but visually correlated.

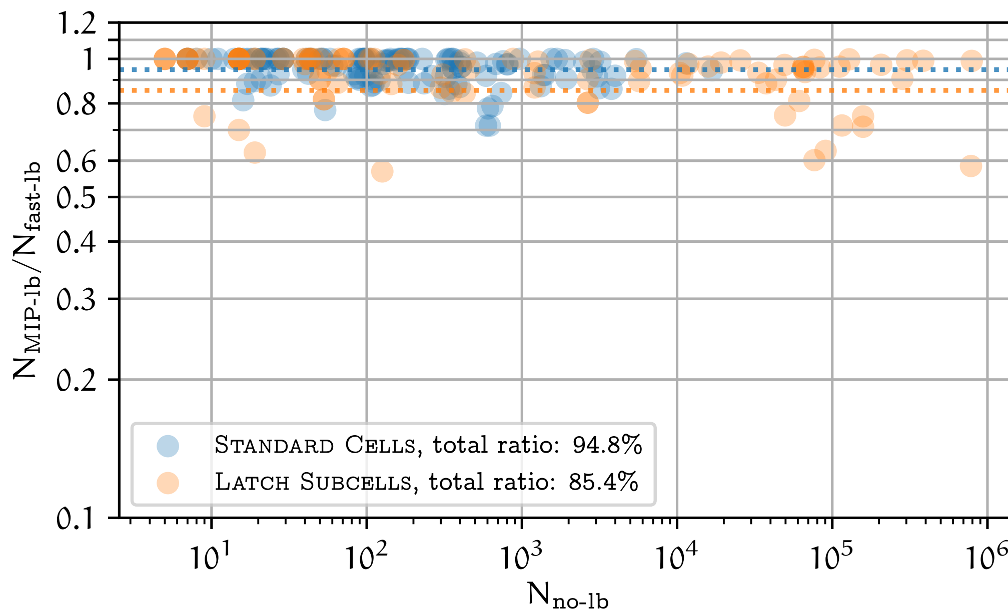


Figure 5.26: Effect of MIP-LB compared to FAST-LB on the number of placement nodes. To allow for a comparison with Figure 5.24, the same x and y range is shown. MIP-LB is never worse than FAST-LB, i.e. no point is above 1. Improvements are not as large as they were for FAST-LB compared to no lower bounds.

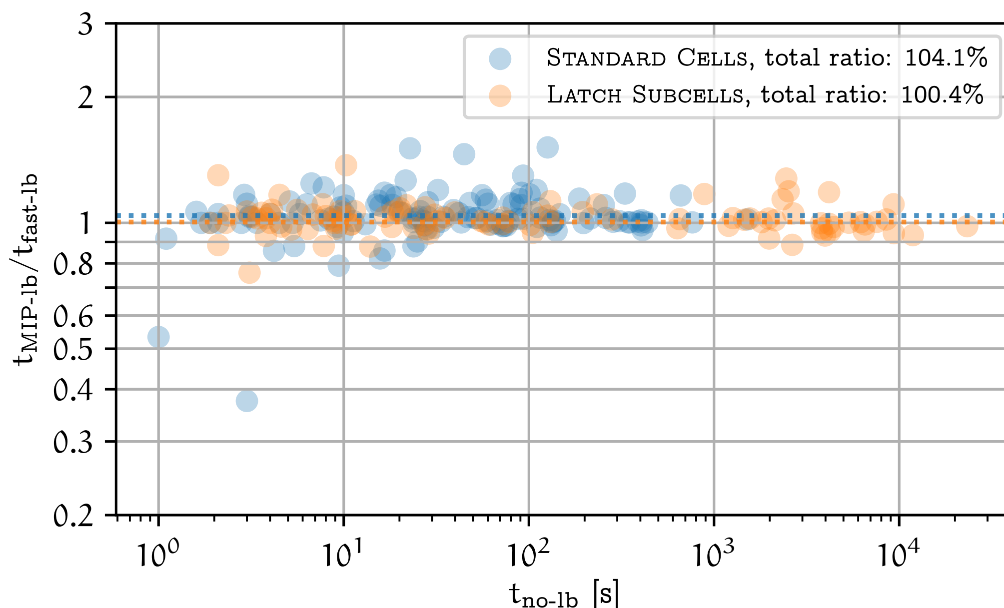


Figure 5.27: Runtime comparison of FAST-LB vs. MIP-LB. Similar to the number of nodes comparison in Figure 5.26 the runtime does not differ much. The slight node savings seem to compensate for the additional runtime cost of solving the MIP.

5.7 Search Tree Ordering

The main goal of our placement algorithm is to find optimum solutions with respect to cell width and netlength as quickly as possible. Usually a solution with optimum cell width but not necessarily optimum netlength can be found faster than an optimum solution. In the context of Algorithms 1 and 2, finding a placement with provably optimum width means finding any routable placement inside the cell width loop. Let t_{width} be the time needed to find a solution with provably optimum cell width and t_{nl} the time needed to find a solution which minimizes our objective function Φ defined in Section 3.3. Since the tool is also used to answer queries such as "find the best placement in time t", we are interested in small values for both t_{width} and t_{nl}. The search tree enumeration strategy influences the times t_{width} and t_{nl}. Enumeration strategy here means which node is picked when we pop from the queue (cf. Line 4 of Algorithm 2). This analysis assumes that each node takes a certain amount of time to be processed and that other runtimes, e.g. popping a certain element off the queue, do not contribute to the total runtime at all. In practice this is a reasonable assumption. We will describe the *Two Stage Strategy* used in BonnCell and analyze its properties. Klotz 2018 gives a more detailed comparison of different strategies and rigorous analysis of their runtime behaviors.

Assume we are given lower bounds l for the netlength for each node $n \in N$. Furthermore, let the netlength of an optimum solution be ν_{opt}. To simplify the analysis, we assume that there is only a single node with netlength ν_{opt}. An easy observation is that we need to visit at least the set of nodes

$$N_{min} := \{n \in N \mid l(n) \leqslant \nu_{opt}\}$$

to prove optimality of the solution found. The Dijkstra algorithm used to find shortest paths in graphs can also be applied in this setting. We keep a list of nodes we want to visit, initially containing only the root vertex. At each step, we choose the next node n to be the one with the lowest value of $l(n)$. Once we have processed a full placement node which is routable, we know that this is the optimum solution as we have already processed all nodes N_{min}. Due to our enumeration strategy we have actually visited exactly the nodes N_{min}. Therefore, the Dijkstra Strategy is an optimum strategy w.r.t. minimizing t_{nl}. Let $\hat{t}_{nl}$ be the runtime needed by the Dijkstra Strategy. The first solution found by the Dijkstra Strategy is already provably optimal, i.e. $t_{width} = t_{nl}$. As described above, a shorter time until a first solution is found is desirable and achievable in practice, with more or less minor sacrifices on t_{nl}. The idea is that the placement is run in two stages. The first stage finds any solution quickly, using a search heuristic. The second stage uses the Dijkstra Strategy to find an optimum solution. Ideally, the first stage should only take a fraction of the entire runtime, s.t. t_{width} is much smaller than $\hat{t}_{nl}$. In the second stage the Dijkstra Strategy is applied which processes the set N_{min}. Some of the nodes in N_{min} were already processed in the first stage and do not need to be processed again. The runtime of the second stage is therefore smaller

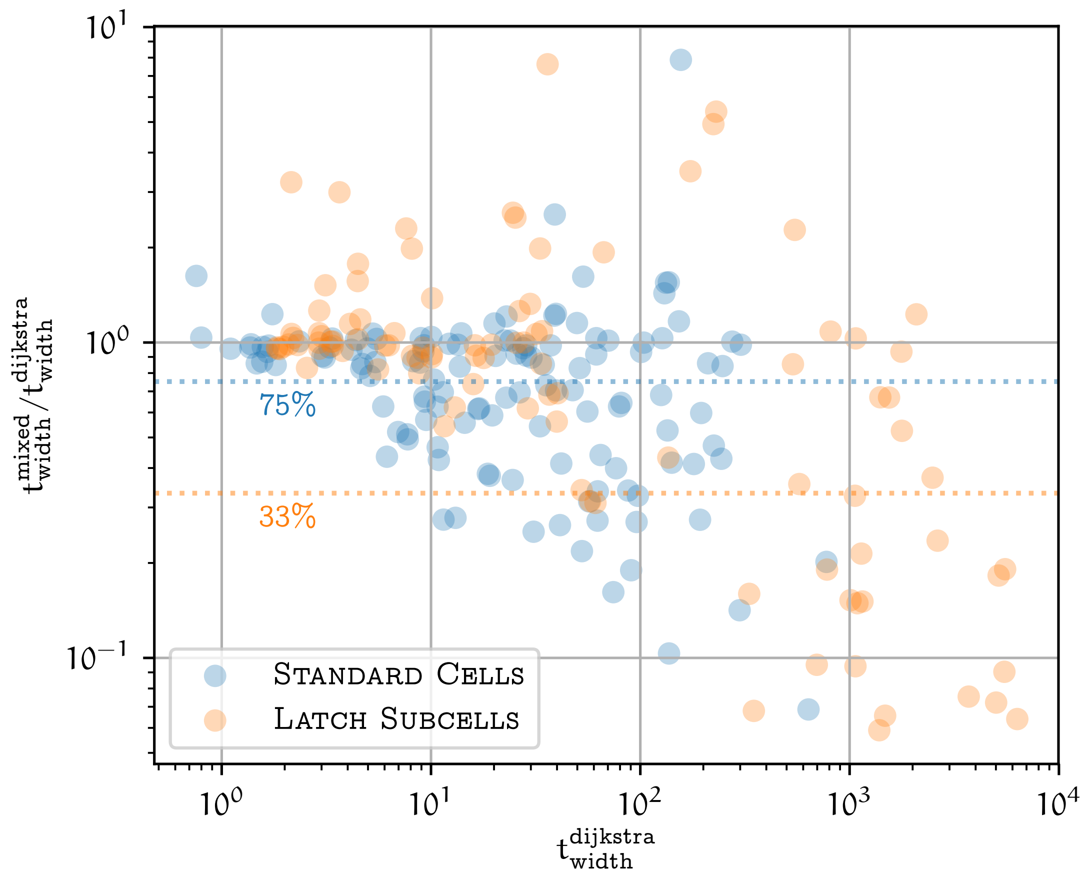


Figure 5.28: Placement runtime until first solution with provable minimum cell width is found for Standard Cells and Latch Subcells instances. Relative runtime of Mixed Strategy $t_{\text{width}}^{\text{mixed}}$ compared to Dijkstra Strategy $t_{\text{width}}^{\text{dijkstra}}$ is shown on the y-axis. The average over the total runtime shows that the mixed strategy faster finds a width minimum solution, it uses 75% runtime for Standard Cells and 33% runtime for Latch Subcells.

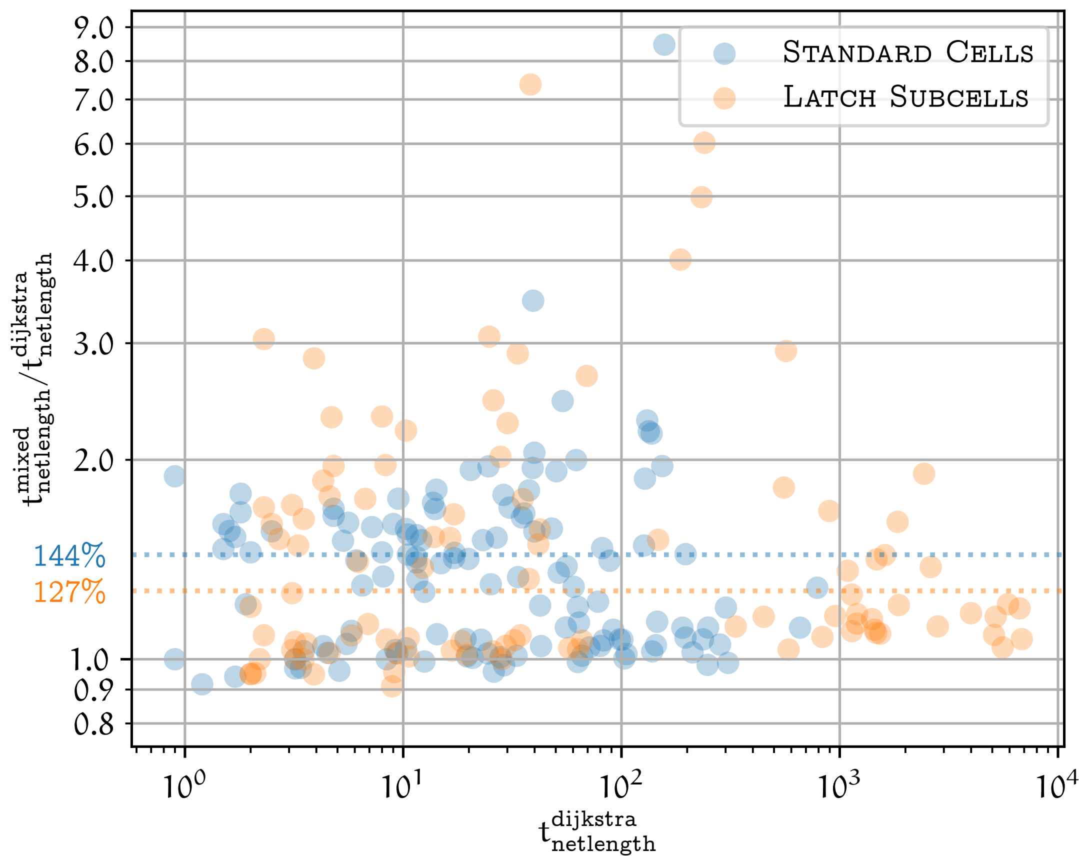


Figure 5.29: Entire placement runtime for finding an optimum placement and proving its optimality. Relative runtime of Mixed Strategy $t^{mixed}_{netlength}$ compared to Dijkstra Strategy $t^{dijkstra}_{netlength}$ is shown on the y-axis. Note that in our model the Dijkstra Strategy should never be slower which is not the case in our data. The set of processed nodes of the Dijkstra strategy is indeed a subset of the nodes processed by the Mixed Strategy. The discrepancy could be explained by fluctuations in processor speed and server loads. Our model also does not take into account runtime needed to sort the nodes in the Dijkstra strategy.

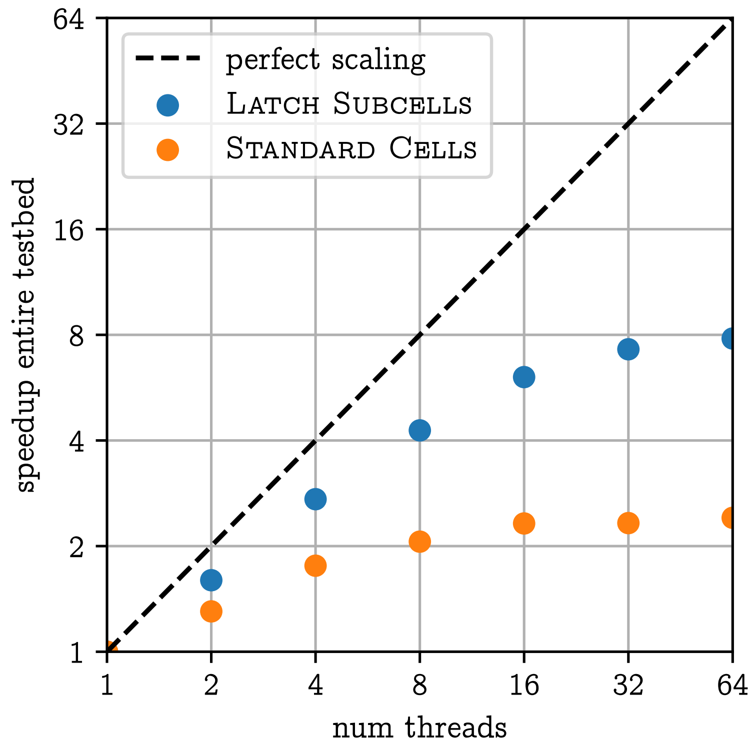


Figure 5.30: Multithreading speedup of entire testbed for Latch Subcells and Standard Cells instances. Speedup for k threads is measured by t_1/t_k, where t_x measures the runtime for the entire testbed run with x threads. The Latch Subcells scale better, see also Figure 5.31. Single threaded the entire Latch Subcells testbed took 24.3h, the Standard Cells took 3.2h.

than $\hat{t}_{nl}$. The total runtime t_{nl} is therefore bounded by $t_{width} + \hat{t}_{nl}$. If t_{width} is small compared to $\hat{t}_{nl}$, this is a good trade-off. Although there is no mathematical guarantee that $t_{width}/\hat{t}_{nl}$ is small, this is often observed in practice.

5.8 Parallelization

Modern processors are implemented with many cores. For a library of cells this allows us to process many cells in parallel by assigning each cell to a core. For large latch instances we usually want to solve a single instance as quickly as possible, potentially using the entire machine. This kind of memory sharing parallelization is more difficult to implement, as different threads need to communicate. However, potential gains can be large. A factor 10 speedup means that an instance which used to take a week and was therefore too large to be solved for practical purposes can now be solved in less than a day. Given enough computing resources, the entire Latch Subcells testbed can be solved in three instead of 24 hours using 64 threads per instance (cf. Figure 5.30). This is less relevant for the Latch Subcells testbed as used here, since it can be parallelized by running each instance independently in a separate process but more important for individual very large instances, e.g. the PLCB presented in Section 8.3.

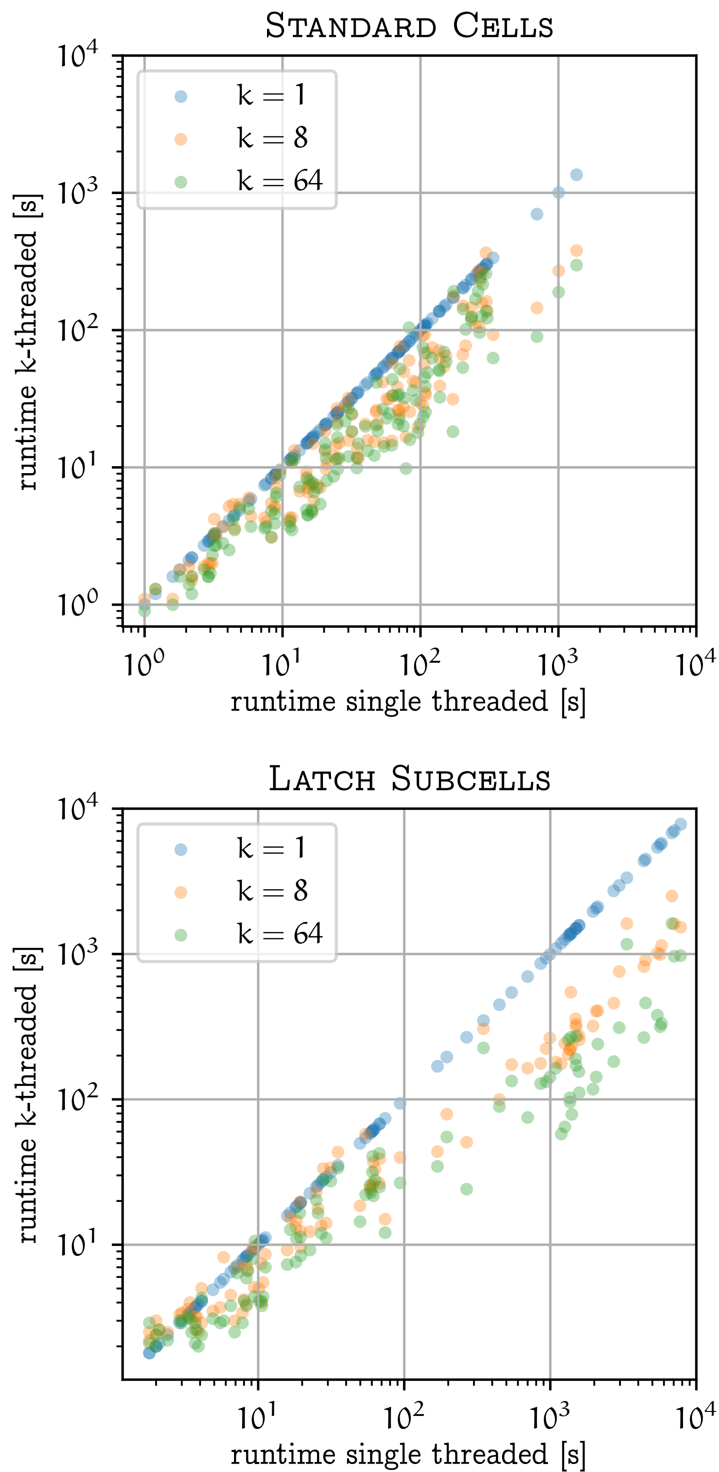


Figure 5.31: Multithreading runtime of all Standard Cells and Latch Subcells instances. Every circle denotes one instance with single threaded runtime as x coordinate and k-threaded runtime as y coordinate. Note that the standard cell runtimes for multiple threads improve less compared to the Latch Subcells runtimes. For Latch Subcells instances, the speedup gets significantly larger for instances with longer runtime. Scaling for the entire testbed has been reported in Figure 5.30.

The placement algorithm (Algorithm 2) is very suitable for parallelization. The search tree consists of independent partial placement nodes which need to be checked for legality and expanded to child nodes. Both of these operations can be done independently of other nodes. Multiple threads simultaneously pop items off the queue, check for legality, and push new children to the queue. Access to the queue needs to be synchronized but is no bottleneck since checking for legality is much slower than queue access. Details of the implementation haven been described in Klotz 2018.

If not done carefully, parallelizing the placement in this way can lead to increased placement runtime. As we have seen in Section 5.7, for some search tree strategies, the order in which nodes get processed determines the set of nodes that need to be processed. With multiple threads processing nodes at the same time additional effects come into play. For instance, one needs to take special care to make sure that the search tree order follows the single-threaded search tree order. It can also happen that a time consuming routability check in one thread is started just before the optimality of another node is shown. A simple implementation would then wait until all routability checks are finished, even though the result is no longer of any interest. We have implemented multiple strategies to avoid these kind of situations, s.t. our multi threaded runs are guaranteed to be faster (only counting time needed to process nodes, as opposed to e.g. management of the queue) than the single threaded version. This can be observed in Figure 5.31. Only very few multi threading runs are slightly slower than the single threaded version, which is unavoidable due to fluctuations in runtime measurements.

5.9 Routing Corridors

One of BonnCells key design aspects is that given a placement, it is able to find an optimum routing. To increase user acceptance of the tool we always want to be able to find a solution which is at least as good as a manual layout. This means that any legal solution must be contained in our search space. However, this comes at a cost. Many legal solutions use large detours which are not necessary and allowing these detours in the MIP formulation slows down the solver.

To overcome this issue, we have the option to restrict the routing to *routing corridors*. These are rectangular areas on the cell, defined for each net, which the router is allowed to use. Edges outside of the routing corridor are forbidden. This reduces the size of the MIP in terms of variables and constraints and also its complexity, which results in shorter solver runtimes.

Definition 5.12. Routing Corridor Let N be a net with terminals $t_1, \dots, t_n$ at x coordinates $x(t_1), \dots, x(t_n)$ in bits $b(t_1), \dots, b(t_n)$. Let

$$\underline{x} := \min_{i \in \{1,\dots,n\}} x_i, \qquad \underline{b} := \min_{i \in \{1,\dots,n\}} b_i,$$
$$\overline{x} := \max_{i \in \{1,\dots,n\}} x_i, \qquad \overline{b} := \max_{i \in \{1,\dots,n\}} b_i,$$

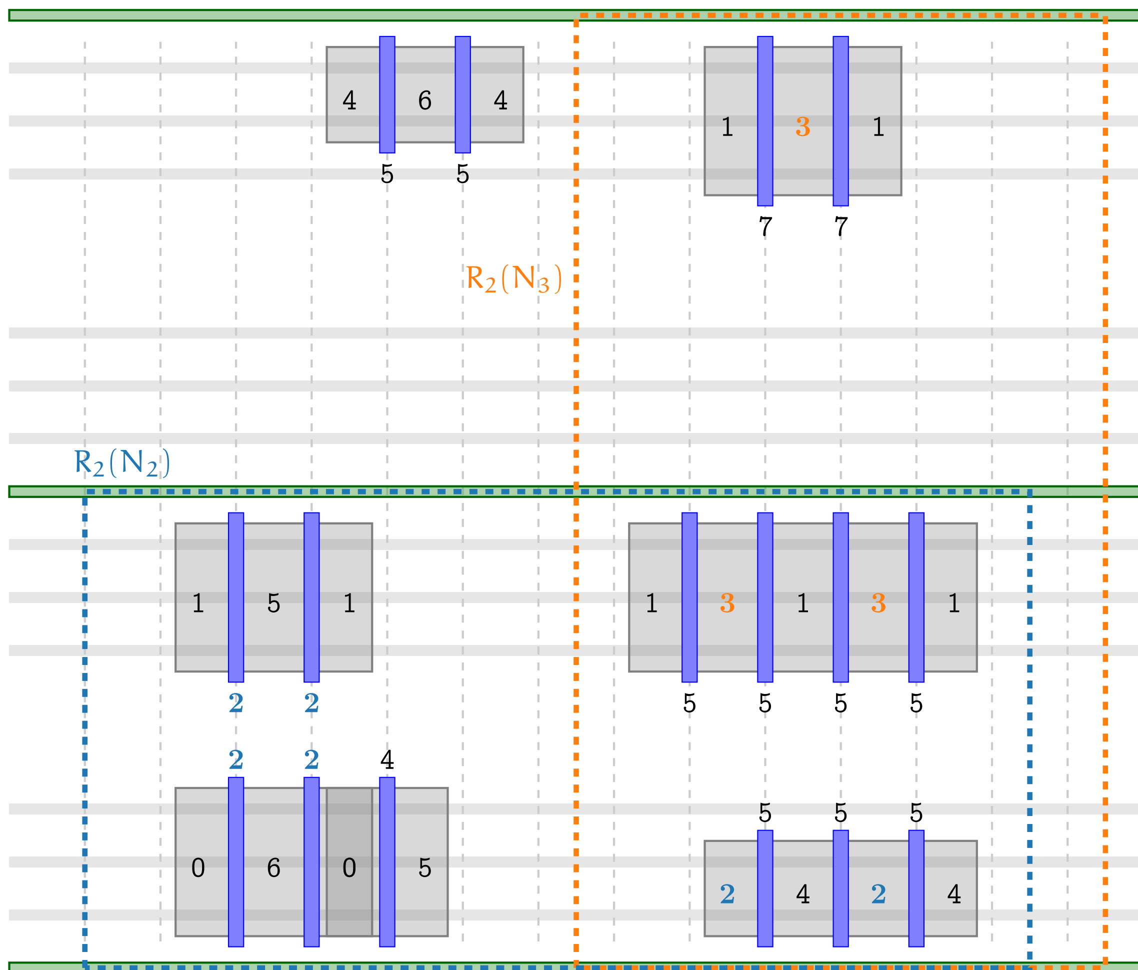


Figure 5.32: Two bit instance with routing corridors for nets N_2 (blue) and N_3 (orange). $k = 2$ has been used for both routing corridors. Note that routing corridors always span entire bits, even if a net does not have any terminals on one of the stacks — see N_3 in the shown example.

and $k \in \mathbb{R}^{\geqslant 0}$. Then the routing corridor R_k of net N is defined as

$$R_k(N) := [\underline{x} - k, \overline{x} + k] \times [\underline{b}, \overline{b}].$$

The parameter k allows some routing space to the left and right of the terminals. This is needed as terminal connections often need overhangs in horizontal direction. In practice the value $k = 2x_{CPP}$, where x_{CPP} is the poly pitch, i.e. the distance between two gates, showed a good trade off between MIP runtime and loss of good routing solutions. Figure 5.32 shows routing corridors for a two bit instance.

Some instances are so congested that large detours for some nets are necessary. Confined to routing corridors, these instances become infeasible. This is critical as the placement feasibility check uses the routing engine to decide routability. We use a two step approach to solve this problem. When we check a placement for routability, we first run the routing confined to routing corridors. If this has been successful, we know

that the placement is routable. If the placement is infeasible with applied routing corridors, we run the original routing without any routing corridors. Especially for STANDARD CELLS, most of the placements are routable, s.t. only the fast routability checks with routing corridors are run. The resulting runtime improvements for the full routing phase and total placement runtime are shown in Figures 5.33 and 5.34.

After the placement algorithm returned a placement, the router is run to find an optimum routing of this placement. It is possible that restricting the solution space to routing corridors discards all optimum routings and we can no longer find an optimum routing solution. A method to overcome this problem could iteratively confine the routing space. Initially we search for a solution without routing corridors. As soon as any, not necessarily optimum, solution is found, we apply routing corridors R_k, where k is chosen s.t. any solution which uses an edge outside of R_k will have larger objective value than the solution already found. This is repeated until we have proven optimality of the routing. This approach has not been implemented as it is difficult to chose the value k for the routing corridor. The routing objective function not only measures weighted netlength but also includes terms penalizing track usage. For example, two nets using the same M2 track are preferred to two nets using different M2 tracks, as they will block less M2 resources for connections outside of the cell.

Next, we present results of routing corridors on placements of the DOUBLE BIT LATCH testbed containing 11 instances. Placements were generated by the multibit placer (Section 7.1) and LINEAR ARRANGEMENT PLACER (Section 7.3) with 12h runtime limit. Therefore, all placements are guaranteed to be routable without routing corridors. There are 4 possible outcomes of the routing algorithm:OPT, SUBOPT, INF, and ABORT. OPT means an optimum solution within 2% optimality gap has been found, SUBOPT means a solution has been found but the optimality gap is larger than 2%, INF means the instance has been proven to be infeasible, and ABORT means that no solution has been found within the runtime limit of 24h.

With routing corridors \ Without routing corridors	OPT	SUBOPT	INF	ABORT
OPT	0	1	0	0
SUBOPT	0	0	0	6
INF	0	0	0	0
ABORT	0	1	0	3

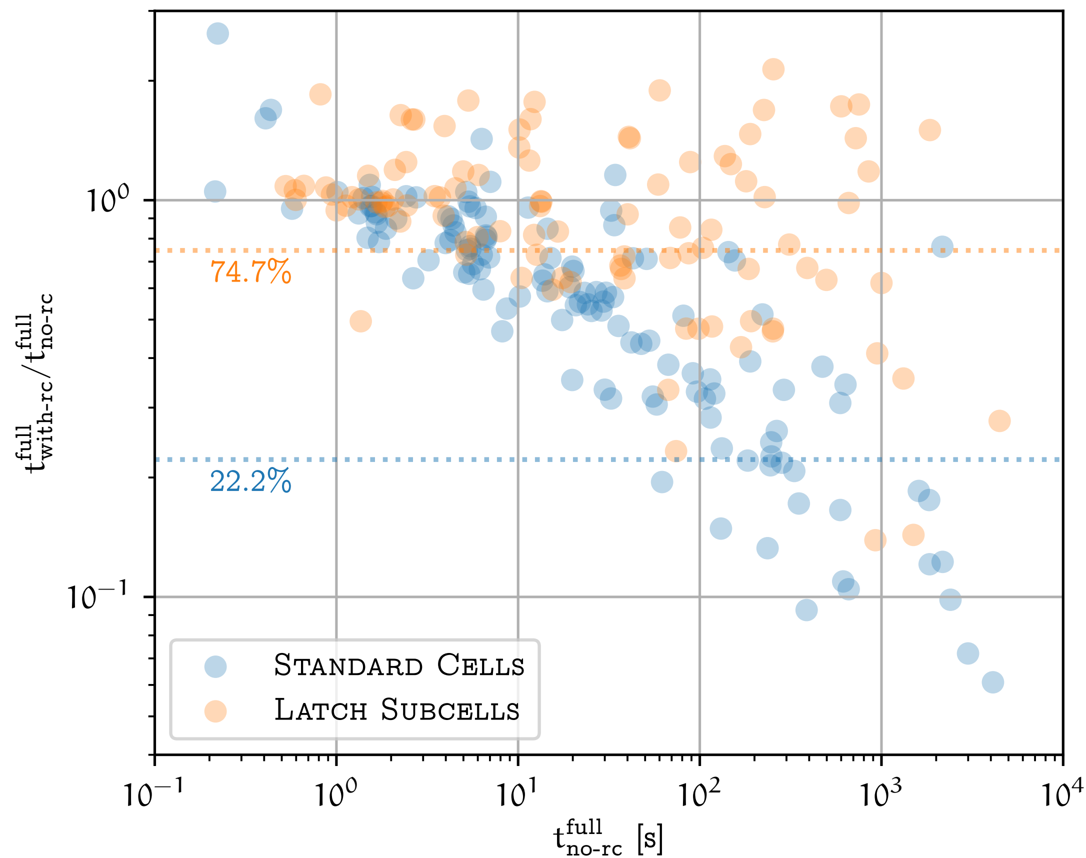


Figure 5.33: Runtime improvement of full phase for routing corridors on Standard Cells and Latch Subcells testbeds. Each point represents one instance with full phase runtime without routing corridors t^{full}_{no-rc} on the x axis and the relative runtime used in the full phase with routing corridors $t^{full}_{with-rc}$ compared to t^{full}_{no-rc}. Full phase runtime with routing corridors here means that the first routability check uses routing corridors. If it finds the placement to be unroutable a second full phase without routing corridors is run. This means that runtime improvements can only be achieved for instances which are feasible even with routing corridors. It can be observed that Standard Cells instances show larger improvements compared to Latch Subcells instances. This is due to the fact that input placements of the full routing check are much more often feasible than infeasible for Standard Cells. For Latch Subcells exactly the opposite is the case. This can be observed in Figure 5.5.

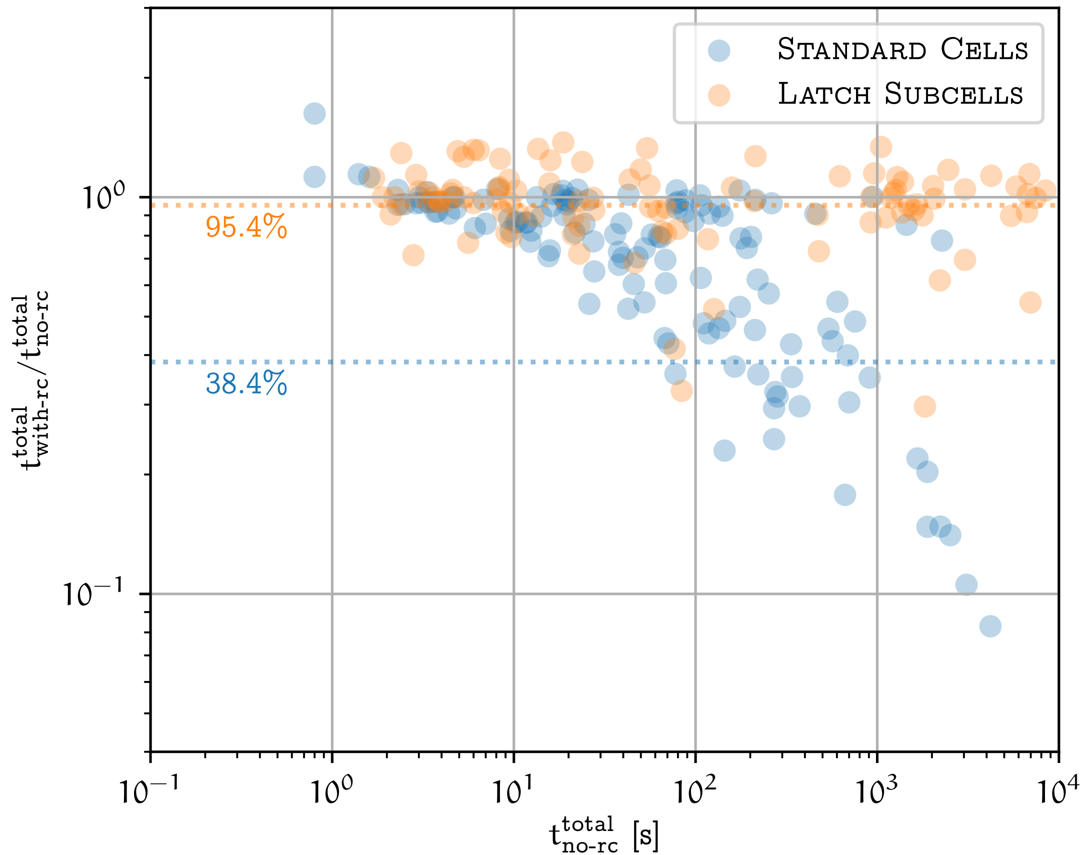


Figure 5.34: Same plot as Figure 5.33 except that total placement runtime t^{total} is shown instead of full phase runtime t^{full}. Same x and y axis range is shown to allow for comparison. Results look quite similar to those of Figure 5.33, Latch Subcells runtimes are effected even less by introducing routing corridors as points are closer to 1 on the y-axis. Same is true for Standard Cells instances but to a lesser extend. Standard Cells instances with higher total runtime are greatly improved by usage of routing corridors with few exceptions. This is consistent with Figure 5.9 which shows that most of the MIP runtime for Standard Cells is spent in the full phase, especially for instances with high runtime. For Latch Subcells more runtime is spent in the FEOL phase, as seen in Figure 5.10.

For 6 instances the routing corridors have been successful as they provided a solution where no solution could be found without routing corridors. In only one case both algorithms found a solution. The result without routing corridors has slightly better objective value (36,200 compared to 36,338 with routing corridors). The optimality gaps were 5.7% and 2% respectively. Routing corridors restrict the search space, therefore it is expected that the resulting objective value of optimum solutions will be larger compared to optimum routings without routing corridors. For one instance no solution could be found with routing corridors but a solution has been found without routing corridors. In three cases both algorithms were not able to find any solution. In total using routing corridors allows finding a solution for 6 out of 9 instances where the algorithm without routing corridors was not able to find any solution. Figure 5.35 shows an example of a multibit routing found with routing corridors.

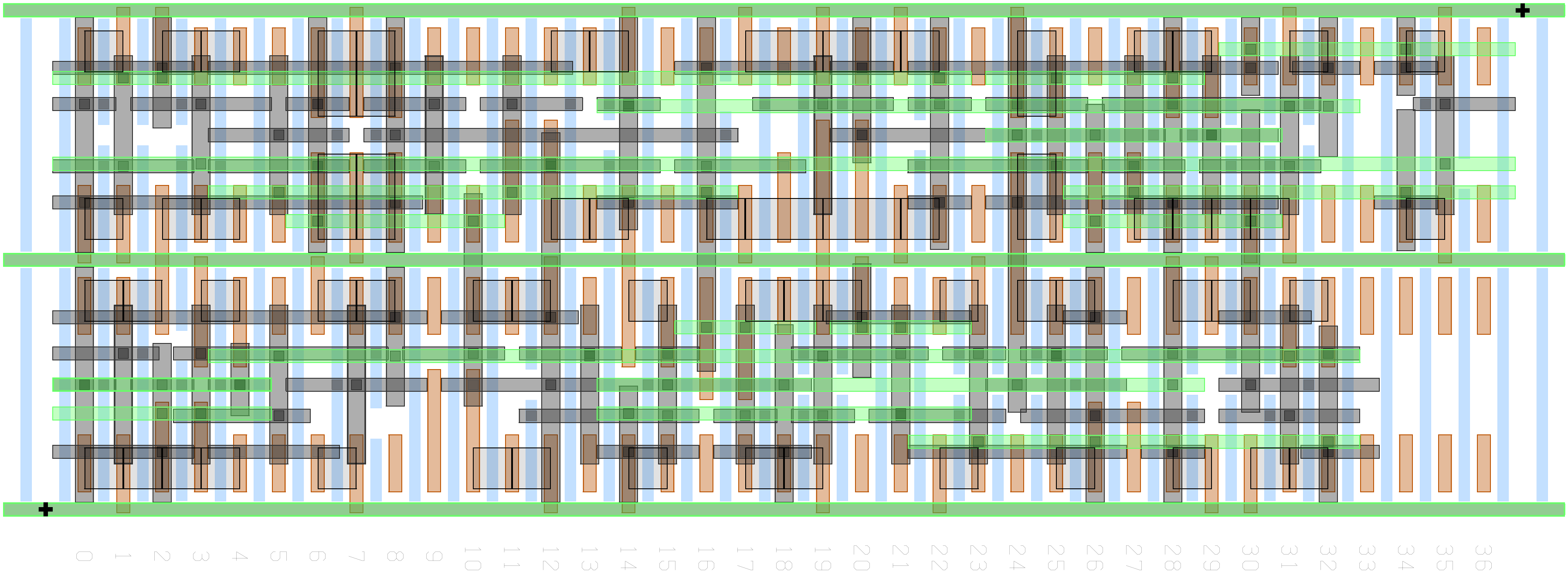


Figure 5.35: Multibit routing of SDFFQDICE_X1M which has been found with routing corridors within 24h and 11% optimality gap.

Chapter 6

Extensions

This chapter contains two extensions of the placement algorithm. Section 6.1 aims at finding placements with better routing solutions, the Folding Placer in Section 6.2 extends the placement search space to find more compact placements.

6.1 Globally Optimum Routings

The placement objective function introduced in Section 3.3 uses the weighted bounding box netlength of a placement. This is not the exact objective we are interested in, which is the routing objective function. It merely serves as an approximation which can quickly be calculated and estimated with lower bounds for partial placements. Since routability is already checked during placement in form of running the router on the placement, we can use the same router call to optimize the routing and use the resulting objective value as the objective value of the placement. Finding an optimum placement then means finding the placement which yields the globally optimum routing objective value. This approach is much slower compared to the objective function introduced in Section 3.3 for two reasons.

1. Full optimization of the routing is slower than deciding routability.
2. Lower bounds for the weighted bounding box netlength of partially placed instances are much easier to compute than lower bounds for the routing objective. The routing objective function contains terms such as track costs, which are hard to estimate. Currently, no lower bounds for the routing objective of partial placements are implemented in BonnCell.

Results for globally optimum routing experiments are shown in Figures 6.1 and 6.2. They show that the objective value improvements for Standard Cells are relatively small, up to 8%. For Latch Subcells the results are stronger as improvements reach up to 38%. Running times increase dramatically in both cases as expected. Slowdowns go up to two orders of magnitude. However, BonnCell is the first tool which is able to find globally optimum placements and routings of cells in 7nm technology. Figure 6.3 compares the placement and routing of a cell whose routing objective value improved by 28%.

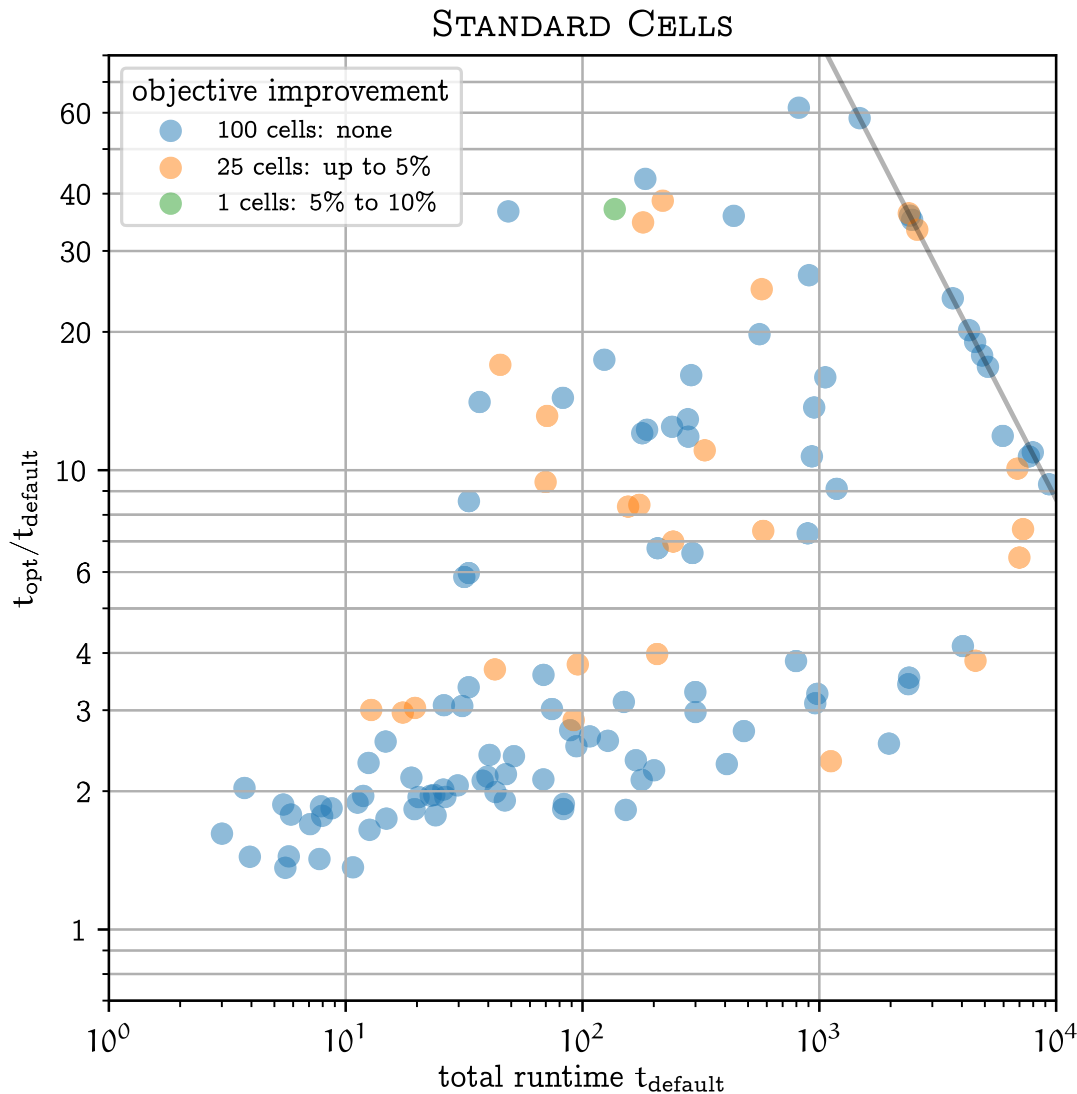


Figure 6.1: Results for globally optimum routings on Standard Cells testbed. Each point represents one cell. Colors indicate how much the routing objective value improved for the globally optimum routing run compared to default settings. 15 instances ran into a timeout in globally optimum routing mode. They lie on the gray line in the upper right of the plot. For 100 cells the placement found by the default run allowed a globally optimum routing or no better routing could be found within 24h. For 25 cells, we could improve the routing objective by up to 5% and for a single cell it has been approved by 5% to 10%. There are two main reasons why most routings are already optimal with default settings. First, this indicates that the objective function used in the default settings is already quite a good estimate for the full routing objective function. Second, the cells of the Standard Cells testbed do not allow many different legal placements with minimum width, due to their regular structure. The position of the points indicate how much additional runtime was spent in the global optimization mode. Small instances only use about twice as much runtime, whereas more complicated instances take up to factor 60 more runtime.

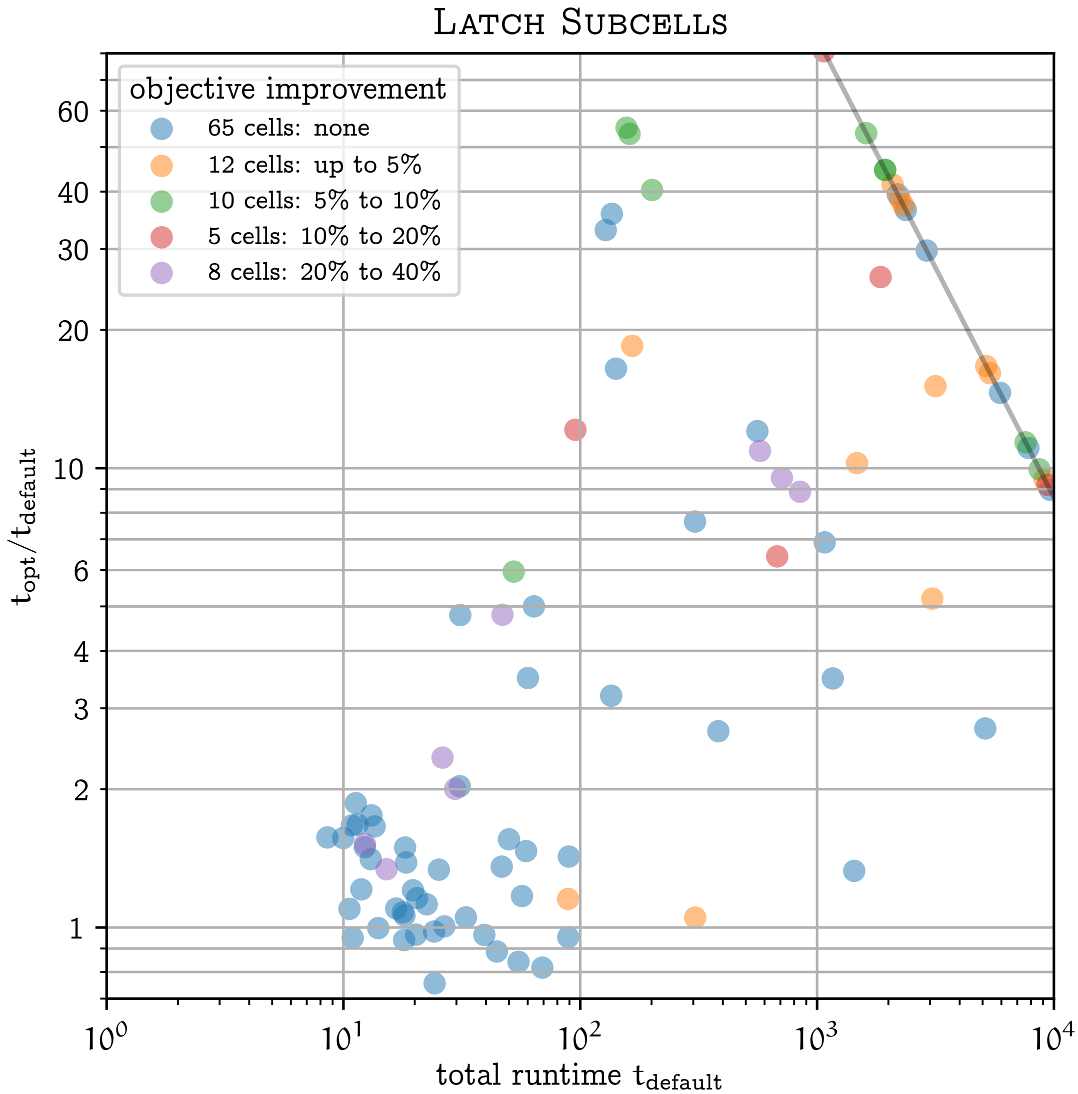


Figure 6.2: Results for globally optimum routings feature on Latch Subcells testbed. Similarly to the Standard Cells Figure 6.1, 26 instances did not finish in global optimum routing mode within 24h but some solution has been found for each of them. These instances lie on the gray line in the upper right of the plot. In comparison to the Standard Cells there are several things to note. First, the objective improvement for this testbed is larger. Some instances even improve by almost 40%. Notably, almost all of the instances which ran into the 24h timeout show improvements in the objective function, although further optimization had been possible.

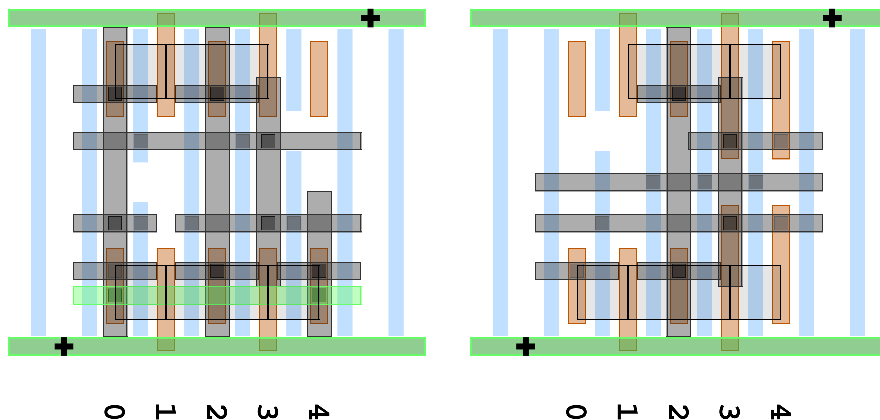


Figure 6.3: Instance from the LATCH SUBCELLS testbed which has 28% objective value improvement. This globally optimum solution on the right can be built without M2 which is preferred over the default solution with one M2 track. The default solution has been found after 25s, the globally optimum solution after 60s.

6.2 Folding

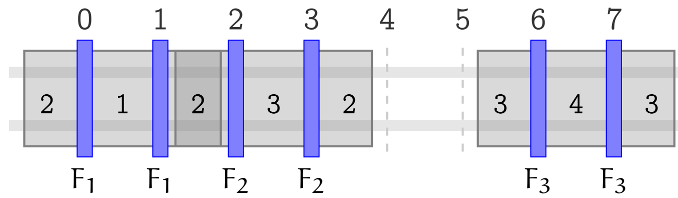
Folding is a technique which splits a FET into several smaller single finger FETs. Each of the smaller FETs has the same gate, source, and drain connections as the original FET. The only property which changes is the allowed size interval. This technique can be useful to find more compact placements.

After splitting a FET F, each of the resulting FETs $F^1, \dots, F^k$ has an allowed size interval containing only a single value[1]. A FET with size interval $[S_{\min}, S_{\max}]$ can be split into FETs $F^1, \dots, F^k$ with sizes $S^1, \dots, S^k$, if the sum matches the original FET interval, i.e.

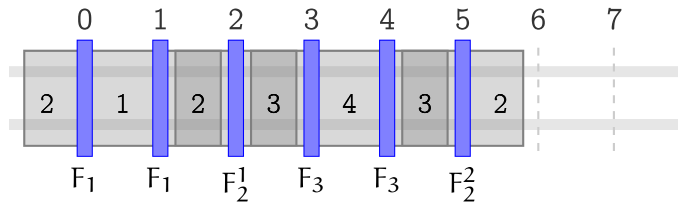
$$\sum_{i=1}^{k} S^i \in [S_{\min}, S_{\max}].$$

In the following we give an example of an instance which can be placed more densely using folding. The instance consists of three FETs F_1, F_2, and F_3 with $S_{\min} = S_{\max} = 4$ for each. The image allows a maximum FET height of 2 fins. The only possibility to place these FETs without folding is therefore with 2 fingers and height 2. The source/drain net indices of F_1, F_2, and F_3 are $1/2, 2/3$, and $3/4$, respectively. F_1 and F_2 can share contacts as well as F_2 and F_3 but not both simultaneously. Therefore a gap of 2 tracks is needed, in the example it is left between F_2 and F_3. Gates are labeled by FET names for clarity.

[1] in this section, a superscript denotes an index.



Using folding we can split F_2 into two FETs F_2^1, F_2^2 with a single finger of height 2 each. This allows all 4 resulting FETs to share contacts in a row without any gaps, resulting in a denser placement.



Our approach to solve instances using the folding technique does not modify the BONNCELL algorithm described in the previous chapters. Instead, it is built as an extension, meaning that the folding algorithm will generate instances with split FETs and let the core algorithm solve these instances. This has the advantage that the code of the core algorithm is not bloated up by folding logic. The instances generated by the folding algorithm will consist only of single finger FETs and many of these FETs will be identical. To avoid enumeration of identical placements by swapping identical FETs in the search tree, we add a rule which forces a certain ordering of the x positions of identical FETs. Thomä 2017 describes the folding algorithm in more detail.

Algorithm 4 shows the Folding Algorithm. Basically, it iterates over all cell widths and for each cell width iterates over all legal partitions of FETs into single finger FETs. The strength of the algorithm comes from two aspects. First, the order in which single finger partitions are iterated is chosen with care. Distributions which are heuristically more likely to yield a legal placement are tried first. Second, LOWERBOUNDWIDTH(d^p, d^n) prevents enumeration of single finger partitions which result in too large placements. LOWERBOUNDWIDTH is based on the methods presented in Section 5.5.

Folding allows to place many of the STANDARD CELLS with smaller width. Figure 6.4 shows the results. Relative runtimes for placement with folding compared to placement without folding range from slight speedups to factor 2400 slowdowns. Some instances cannot be solved with folding within a runtime limit of 24h whereas the default mode takes only up to 25min on the STANDARD CELLS.

Many of the successful folding solutions obey a very simple structure. They require only one finger to be split off one or two FET. The previously shown example also obeys this structure. A prototype implementation based on this heuristic has shown that solutions with the same quality compared to the full folding algorithm can be achieved in many cases with similar runtimes compared to the default mode.

Algorithm 4: FOLDING

input : FETs $\mathcal{F}$ to be placed
output: Routable folded placement P with minimum width

```
1  for W_cell := 1, 2, ... do
2      for F^p := F^p_min, ..., W_cell do                    // # fingers on P stack
3          foreach distribution d^p of F^p fingers to F_p do
4              for F^n = F^p_min, ..., W_cell do              // # fingers on N stack
5                  foreach distribution d^n of F^n fingers to F_n do
6                      if LowerBoundWidth(d^p, d^n) > W_cell then
7                          continue
8                      p ← PlacementOracle(d^p, d^n, W_cell)
9                      if p is not null then
10                         return p
```

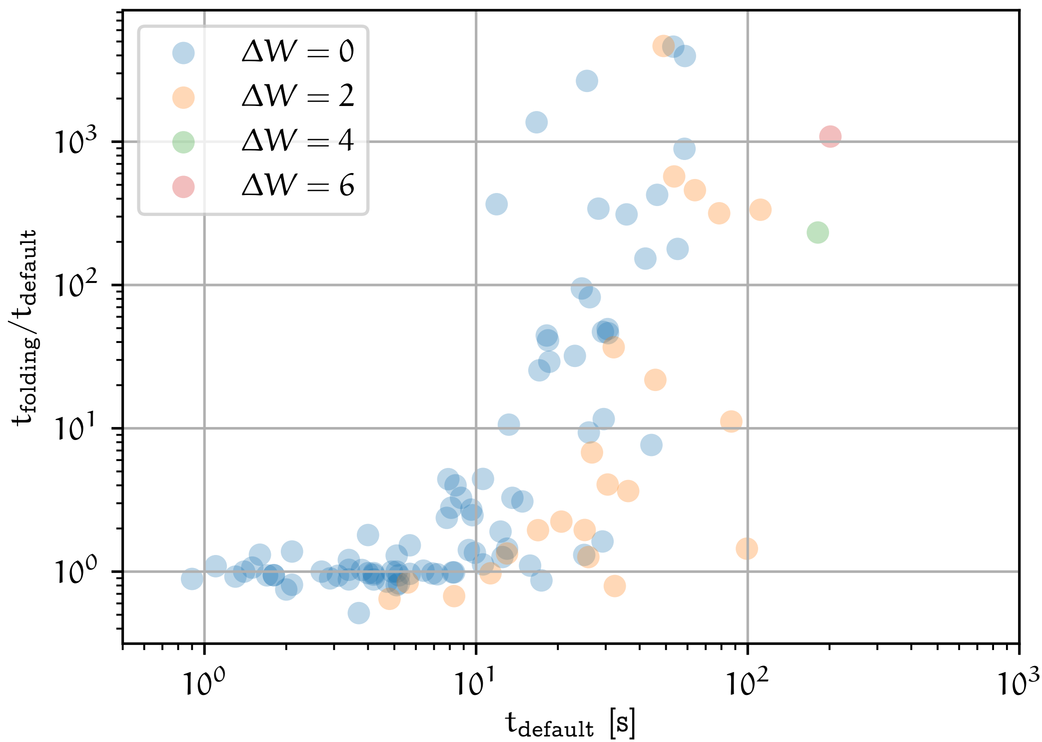


Figure 6.4: Results of Folding Algorithm on STANDARD CELLS testbed. 108 out of 126 instances finished within 4 days. One instance gets 6 tracks smaller ($\Delta W = 6$), another instance 4 tracks ($\Delta W = 4$), 22 instances improved by 2 tracks ($\Delta W = 2$), and 84 instances have already optimum width in default mode ($\Delta W = 0$). Note that cells can only have an even number of tracks as widths due to the cell image. Runtimes increase heavily especially for instances with larger runtime in default mode t_{default}.

Chapter 7

Big Cell Placement

Some cells are currently too large to be solved by the presented placement algorithm, even if all runtime improvements of Chapter 5 are applied. In the past, designers would then split the cell into several *subcells* which are solved independently by BONNCELL. These subcells needed to be connected to each other manually afterwards. The result was often unsatisfying large, as the choice of subcells determines the total cell width and is hard to estimate for designers. There were also difficulties to connect the subcells without using layers above the cell level. Furthermore, the entire approach was tedious and error prone, as it required a lot manual intervention.

By now, BONNCELL is able to automate these tasks and produce better results compared to the flow described above. It uses different strategies to divide large cells into manageable subcells, while optimizing the total cell width. It solves these subcells individually but guarantees routability of the entire cell. All connections between subcells are realized on cell layers, i.e. up to M2. This is all done fully automatically, without the need of user intervention. Connections between subcells are guaranteed by *pins*. A pin is a set of locations from which the net has to connect at least one. For example, a *north pin* is connected if the net connects any point on the north boundary of the cell. This guarantees that the neighboring cell on the next circuit row can connect to the net at some point. Pins can also be restricted to an exact position. This is useful if there already exists a wire in the neighboring cell which needs to be connected to. All placement algorithms in this chapter make use of the PLACEMENT ORACLE which is defined as follows.

Definition 7.1. Given

- a set of FETs $\mathcal{F}$,
- a placed and routed left neighbor cell, i.e. west pins with fixed tracks,
- north and east pins,
- south pins with fixed tracks, and
- the maximum cell width $w_{\max}$,

the PLACEMENT ORACLE either finds an optimum placement and routing of $\mathcal{F}$, s.t. the routing meets all pin requirements and has cell width $w \leqslant w_{\max}$, or proves that no such placement and routing exists.

This routing oracle is implemented by the core placement algorithm presented in the previous chapters. The big cell placement algorithms can use this oracle to guarantee routability of the entire cell. The entire cell is built circuit row by circuit row, starting from the bottom. For each row, several subcells are placed and routed from left to right, one after the other. Whenever a subcell is solved, all bits below and all subcells to its left are already placed and routed. This means that it can adapt its routing to the environment thereby guaranteeing routability of the entire cell.

Next, we describe how FETs are distributed to the circuit rows in Section 7.1. Then, we describe how individual circuit rows are solved by the DIVIDE PLACER (Section 7.2) and the LINEAR ARRANGEMENT PLACER (Section 7.3). Finally, we compare these placers in Section 7.4.

7.1 Multibit Cells

Very large cells are typically not implemented on one but several bits, i.e. neighboring circuit rows. To place a cell on several circuit rows, we first compute an assignments of FETs to the rows using a mixed integer programming approach. Assignments are evaluated by their number of bit crossing connections and an estimation for the total cell width. The rows are placed one after the other with each new placement respecting constraints due to already placed rows.

More specifically, given a set of FETs $\mathcal{F}$ with minimum width w_F, a number of bits B, coefficients for the objective function $c_W, c_C \in \mathbb{R}$, we solve the following MIP.

Variables

$x_{Fb} \in \{0,1\}$	$\forall F \in \mathcal{F}, b \in \{1,\dots,B\}$. 1 if FET F is placed on bit b
$l_N \in \{1,\dots,B\}$	$\forall N \in \mathcal{N}$. Lowest bit in which net N appears
$u_N \in \{1,\dots,B\}$	$\forall N \in \mathcal{N}$. Highest bit in which net N appears
$C \in \mathbb{N}$	total number of bit crossings
$W \in \mathbb{N}$	cell width

Objective

Minimize $\quad c_W W + c_C C$

Constraints

$$C = \sum_{N \in \mathcal{N}} u_N - l_N$$

$$W \geqslant \sum_{F \in \mathcal{F}_s} w_F x_{Fb} \qquad \forall b \in \{1,\dots,B\}, s \in \{P,N\}$$

$$x_{Fb} = 1 \Rightarrow l_N \leqslant b \leqslant u_N \qquad \forall F \in \mathcal{F}, \forall b \in \{1,\dots,B\}, \forall N \in \mathcal{N}(F),$$

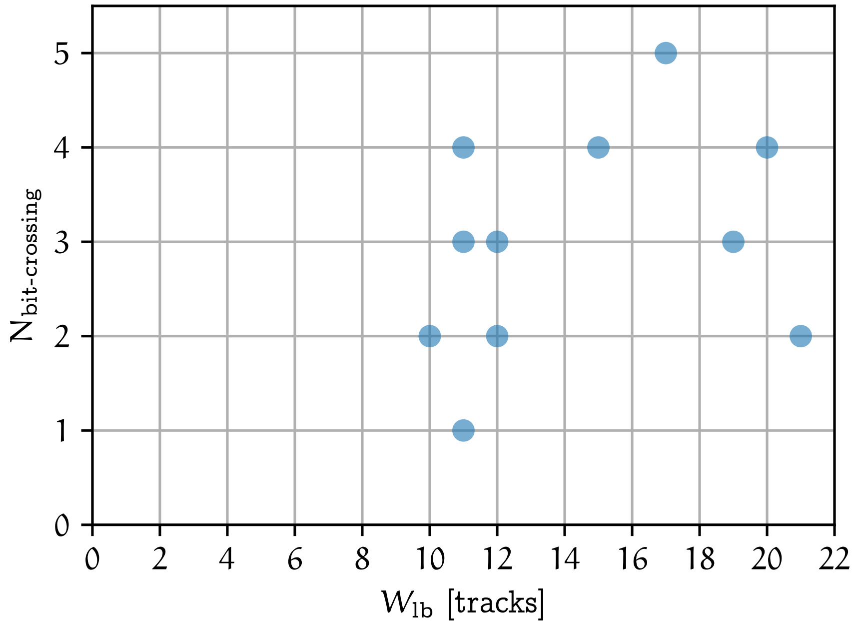


Figure 7.1: Estimated cell width W_{lb} and number of bit crossings $N_{\text{bit-crossing}}$ for the 11 double bit LATCH instances. Each point represents one instance. Most latches can be built with 2 or 3 bit crossings only which means that most of the 25 – 42 nets are connected within the bits. MIP runtime is very small for every instance, at most 0.13s.

where $\mathcal{F}_s$ denotes the FETs of type s and $\mathcal{N}(F)$ the set of nets connected to FET F.

To keep nets short, we penalize nets which cross bit boundaries. The MIP describes a solution which trades this off against the cell width. In practice we use the parameters $c_W = 10$ and $c_C = 1$ which usually means that from all solutions with minimum cell width, the solution with minimum number of bit crossings is chosen. The given constraints do not guarantee that a placement with cell width W exists, but W gives a lower bound on the cell width. The model assumes that all FETs can be built with minimum number of fingers and can all share. It is possible that improvements of this model allow better multibit placements. Figure 7.1 shows the estimated width and number of bit crossings for 11 double bit LATCH instances. Figures 7.2 and 7.3 give an example placement and routing of the double bit SDFFQ X3M latch.

7.2 Divide Placer

From now on we want to solve a single bit. It could be a single bit instance or a multi bit instance, where we have already distributed the FETs into separate bits. The larger the instance, the longer the PLACEMENT ORACLE takes. Our instance might be too large to be solved by a single call of the PLACEMENT ORACLE, so we are looking for an algorithm which always yields a solution, potentially at the cost of optimality. In this section we present the DIVIDE PLACER in two versions, top down and bottom up. The two algorithms are similar and shown in Algorithms 5 and 6.

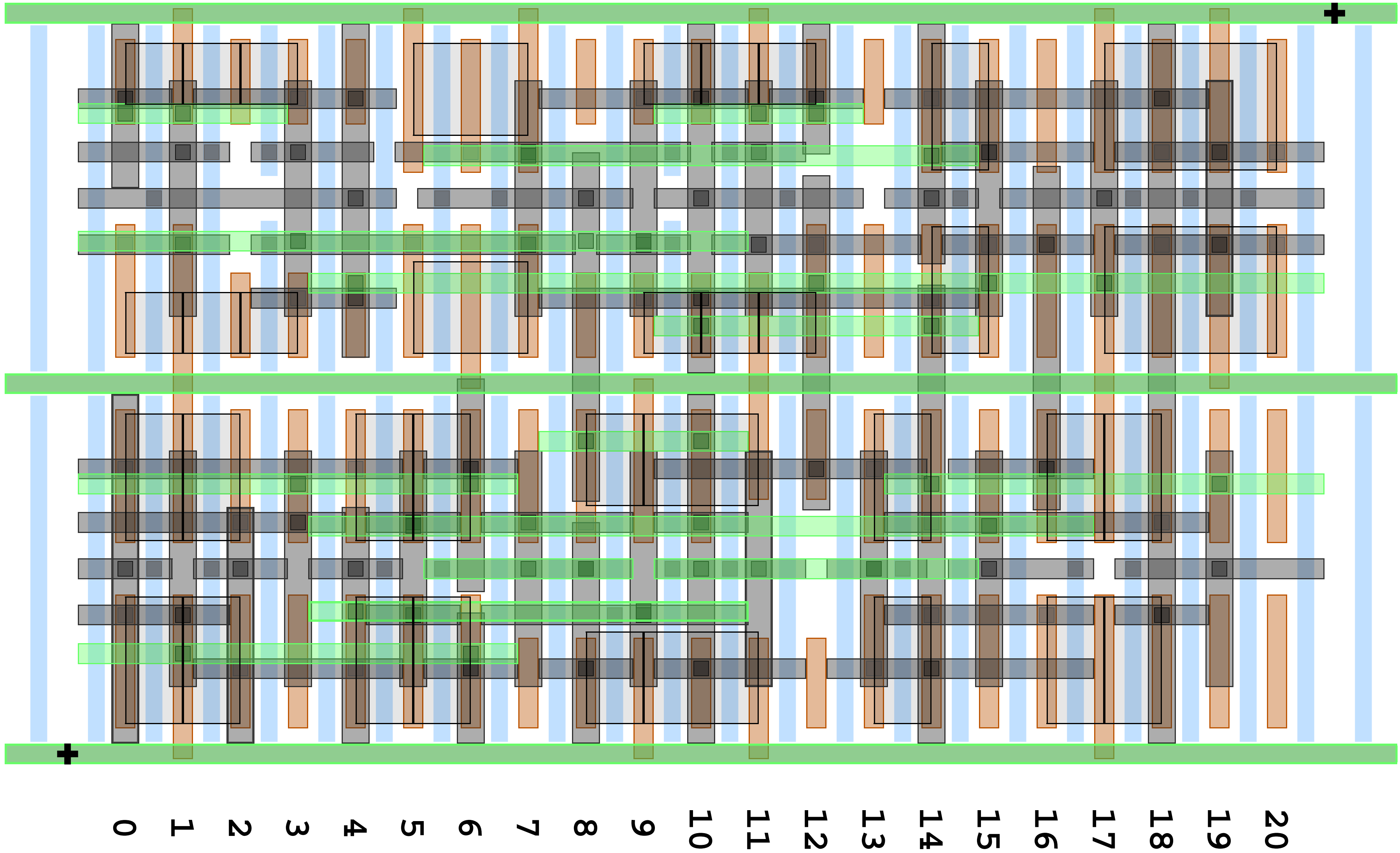


Figure 7.2: Double bit placement and routing of the SDFFQ X3M instance.

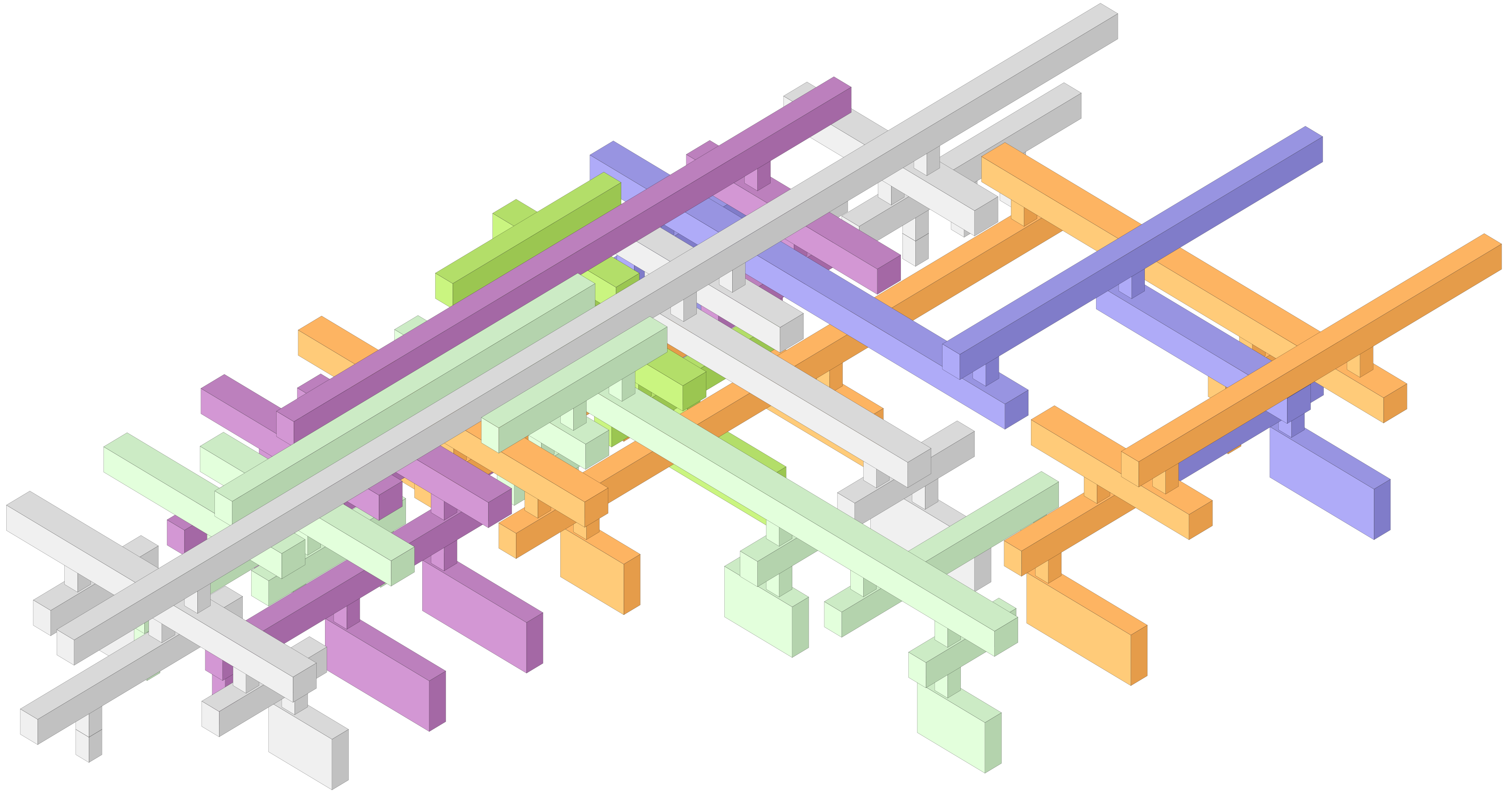


Figure 7.3: Isometric projection of SDFFQ X3M built as double bit instance (same routing as Figure 7.2). Only 7 out of 28 nets are shown to allow views of the lower layers. Each net is drawn with its own color.

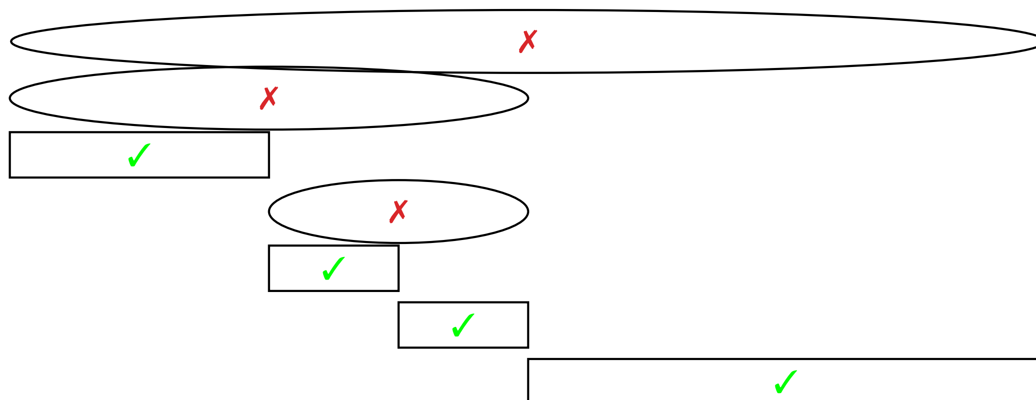


Figure 7.4: Visualization of Divide Placer in top down version. Each row from top to bottom corresponds to one Placement Oracle call in chronological ordering. Whenever a Placement Oracle call fails, the cell is divided into two subcells which are then tried, starting from the left. This makes sure that the cells are solved from left to right which guarantees routability of the entire cell. Final result consists of the 4 subcells marked with ✓.

For the top down version, we try to solve the entire cell with some given time limit. On failure, we split the cell into two subcells and recursively apply the algorithm on both of them. If we fail to find a solution for a subcell with only a single FET, the entire algorithm fails. Otherwise, we concatenate all subcell solutions which yields a solution of our original instance. Figure 7.4 gives an illustration of the algorithm in the top down version.

In the bottom up approach, we first split the cell into subcells and recursively apply the algorithm on these. This gives us a solution for the original cell built from two subcell solutions. Afterwards we try to find a better solution by solving the cell with a single Placement Oracle call. If this is successful, the result is preferable over the two subcell solutions. In practice we restrict the search space of the Placement Oracle to only look for solutions which are better than the already found concatenation of subcell solutions, as this can lead to considerable speedups.

Note that both versions of the Divide Placer allow the cells to be solved from left to right, i.e. when we solve a subcell, all subcells to the left are already solved. This property is important as it can be used to guarantee routability of the entire cell.

The Divide Placer makes use of the fact that cells with fewer FETs are, on average, solved faster than cells with more FETs. Optimum solutions might be lost however. First, one has to leave gaps between subcells to avoid shorts and guarantee manufacturability of the cell. Second, with each division into two subcells there is the chance to separate FETs which could be more densely packed in the same subcell. However, in some cases an instance can be divided into several subcells without sacrificing the total cell width. Figure 7.5 shows an example of such an instance.

So far, we have only described the basic skeleton of the Divide Placer. Three main questions remain to be answered: “how is routability guaranteed?”, “how to split

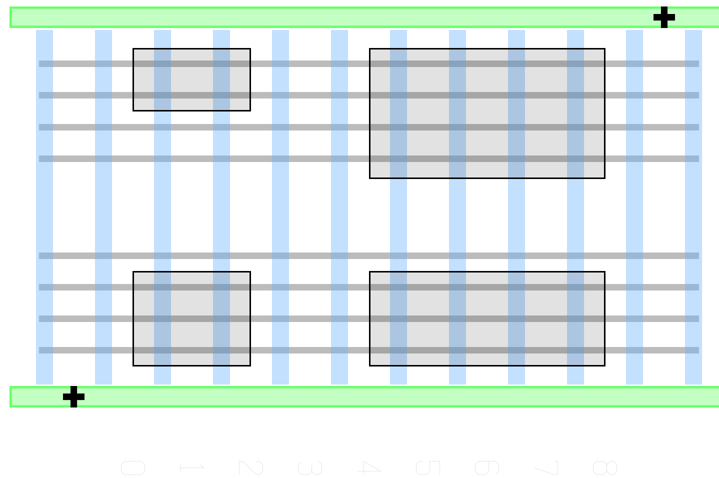


Figure 7.5: Example of an instance which can be divided into two subcells and still be placed with minimum width.

Algorithm 5: DIVIDE PLACER - top down

---

input : FETs $\mathcal{F}$.
output: Routed placement of $\mathcal{F}$ or timeout-failure.

1 result $\leftarrow$ PlacementOracle($\mathcal{F}$, $t_{\mathcal{F}}$)
2 **if** *result is successful* **then**
3   **return** result
4 **if** $|\mathcal{F}| = 1$ **then**
5   **return** timeout-failure   `// Single FET subcell failed`
6 Split $\mathcal{F}$ into $\mathcal{F}_L, \mathcal{F}_R$
7 **return** concat(DividePlacer($\mathcal{F}_L$), DividePlacer($\mathcal{F}_R$))

---

a cell into two subcells?", and "how much runtime is given to the PLACEMENT ORACLE for each subcell?". These questions will be answered in the next subsections.

7.2.1 Routability Guaranty

All actions required to guarantee routability are quite technical and we will only give the high level details. Routability is guaranteed by forcing connections to the subcell boundaries. If we know that a net N appears in this subcell and also in some subcell to the *right*, we force a connection on any track on the right boundary. If we know that a net N appears in this subcell and also in some subcell to the *left* we force a connection to the left boundary but this time to a specific track on a specific layer. This is because we know that the left neighbor has already a connection of N to its right boundary and we make sure to connect on the same track. There are some drawbacks with this techniques. The left subcell doesn't know about the routing of the right subcell and will choose any right boundary track. This choice might be very restrictive for the right subcell as it must adapt its placement to be able to connect properly. A more

Algorithm 6: DIVIDE PLACER - BOTTOM UP

```
   input  : FETs 𝓕.
   output: Routed placement of 𝓕 or timeout-failure.
1  if |𝓕| ⩾ 2 then
2  |  Split 𝓕 into 𝓕_L, 𝓕_R
3  |  P ← concat(DividePlacer(𝓕_L), DividePlacer(𝓕_R))
4  |  if P is not successful then
5  |  |  return timeout-failure
6  result ← PlacementOracle(𝓕, t_𝓕)
7  if result is successful then
8  |  return result                        // Guaranteed to be better than P
9  else
10 |  if |𝓕| = 1 then
11 |  |  return timeout-failure             // Single FET subcell failed
12 |  return P
```

sophisticated approach would find a trade off between both cells but is currently not implemented. The advantage of the current approach is that we only need to route single subcells. Once we have found a solution, there is no need to touch it anymore.

7.2.2 Subcell Splitting

Splitting FETs into subcells is done using a MIP approach similar to the bit assignment MIP used in Section 7.1. We minimize the cut value, i.e. the number of nets that need to connect both subcells and the total estimated width of both cells. The goal is to distribute FETs to two subcells, s.t. both of these subcells can be solved quicker than the original cell. In contrast to the bit assignment MIP, we have to add balance constraints to make sure that both resulting subcells are significantly smaller than the original cell. The entire model is given by

Variables

$x_{Fk} \in \{0, 1\}$	$\forall F \in \mathcal{F}, \forall k \in \{1, 2\}$. 1 if FET F is placed in subcell k
$l_N \in \{1, 2\}$	$\forall N \in \mathcal{N}$. Leftmost subcell in which net N appears
$u_N \in \{1, 2\}$	$\forall N \in \mathcal{N}$. Rightmost subcell in which net N appears
$C \in \mathbb{N}$	total number of bit cuts
$W_k \in \mathbb{N}$	Width of subcell $k \in \{1, 2\}$

Objective

$$\text{Minimize} \quad c_W(W_1 + W_2) + c_C C$$

Constraints

$$
\begin{aligned}
&C = \sum_{N \in \mathcal{N}} u_N - l_N \\
&W_k \geqslant \sum_{F \in \mathcal{F}_s} w_F x_{Fk} && \forall k \in \{1,2\}, \forall s \in \{P, N\} \\
&x_{Fk} = 1 \Rightarrow l_N \leqslant k \leqslant u_N && \forall F \in \mathcal{F}, \forall k \in \{1,2\}, \forall N \in \mathcal{N}(F) \\
&\sum_{F \in \mathcal{F}} x_{Fk} \geqslant r|\mathcal{F}| && \forall k \in \{1,2\},
\end{aligned}
$$

where r denotes the minimum relative number of FETs that should be in each subcell.

Similar to Section 7.1, the model only very roughly estimates the width of a subcell by summing up the minimum FET widths. This estimation is only accurate if all FETs in one subcell can share contacts. Especially for latch instances, this is often not the case. We tried to use an improved model, which respects the FET sharing rules and calculates a legal placement for each stack and subcell individually. This improved model has already been described in Section 5.5.2 to calculate a lower bound on the cell width. However, the improved model is deactivated by default, since its increased runtime outweighed the benefits from improved cell division. Irrespective of the exact model, we use the constants

$$r = \frac{1}{3}, \quad c_W = 10, \quad c_C = 1.$$

7.2.3 Runtime Distribution

The goal of the Divide Placer is to find the best possible placement within the given runtime limit. Therefore, we want to distribute the given runtime limit to Placement Oracle calls in the most efficient way. In the top down approach we start with the entire cell, try to solve it and on failure split the cell and recursively apply the procedure on the two subcells. Our approach assigns the same runtime for the oracle call for the entire cell as for the sum of the oracle calls for the two subcells. Between the two subcells the runtime is split evenly. Let t denote the runtime limit for the entire Divide Placer and t_0 the runtime limit for the initial cell containing all FETs. Then at each recursion depth the total amount of runtime is bounded by t_0. As the recursion depth is bounded by

$$d_{max} = \log_{\frac{1}{1-r}} |\mathcal{F}|,$$

where r is the splitting factor introduced in Section 7.2.2, the value

$$t_0 := \frac{t}{d_{max} + 1}$$

ensures that the total runtime of all Placement Oracle calls will not exceed the total runtime limit t. On a given depth level d, the runtime limit for a single subcell

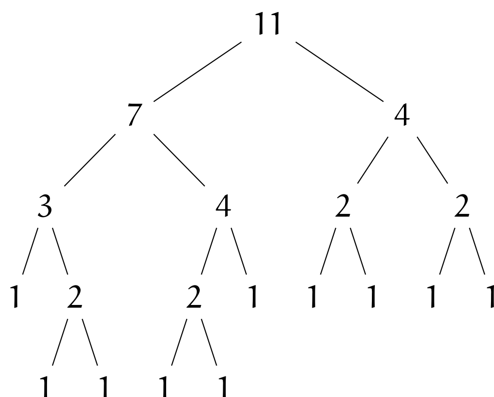


Figure 7.6: Full division tree for an instance with 11 FETs. Each node corresponds to a subcell and its label denotes the number of FETs inside the subcell. Splitting continues until every leaf subcell has only a single FET.

is given by $2^{-d}t_0$ as there are at most 2^d subcells on a given depth level. Note that on some levels there might be less subcells which need to be solved. In this case the reserved runtime for this depth level will not be entirely used.

With this approach it often happens that a large fraction of the total runtime limit remains unused after the DIVIDE PLACER finishes. This is because the runtime reserved for the levels deep down the division tree is never used because subcells at higher levels have already been solved. This gave rise to the bottom up version of the DIVIDE PLACER. In the bottom up approach we calculate the entire division tree as a first step. See Figure 7.6 for an illustration of an entire division tree for 11 FETs. The subcells of the tree are solved in post order tree traversal as shown in Figure 7.7. Initially, each of the FETs gets a runtime budget of $t/|\mathcal{F}|$. When a subcell is solved by the PLACEMENT ORACLE the remaining runtime budget of all FETs in this subcell is summed up and used as the runtime limit for this oracle call. This works as in future oracle calls these FETs will always be in the same subcell, s.t. the remaining runtime can be used for these calls.

Results of the two modes compared to lower bounds and the LINEAR ARRANGEMENT PLACER (Section 7.3) can be seen in Section 7.4.

7.3 Linear Arrangement Placer

The LINEAR ARRANGEMENT PLACER is a bottom up approach to solve large cells. Its main goal is to overcome the drawbacks of the DIVIDE PLACER and find smaller placements. The main issue of the DIVIDE PLACER is that if one subcell is chosen poorly and the PLACEMENT ORACLE fails to place it, the mistake cannot be undone later. In the top down version, this subcell will be split into several smaller subcells, thereby wasting space. In the bottom up version it can happen that even a small subcell is hard to solve and no parents of this subcell in the division tree will ever be tried.

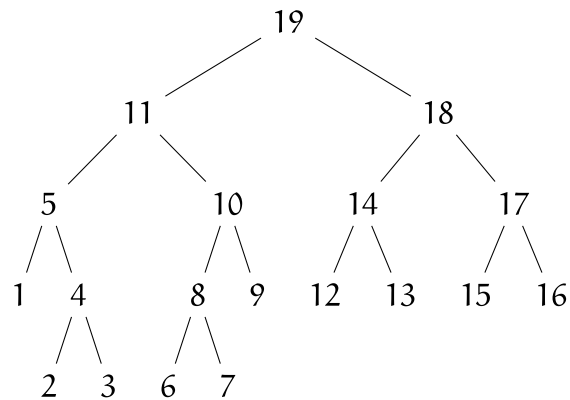


Figure 7.7: Postorder traversal of the tree shown in Figure 7.6. Numbers denote the order in which subcells are solved by the PLACEMENT ORACLE in the bottom up approach.

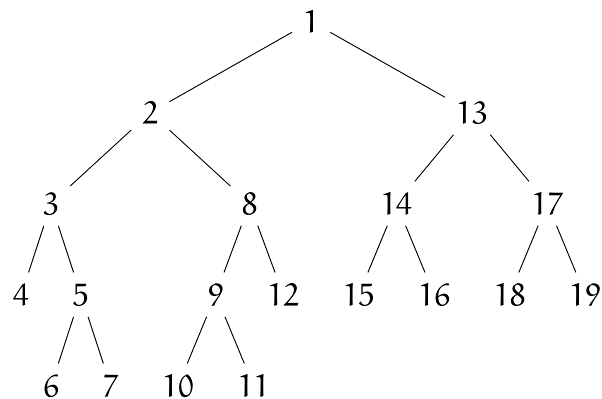


Figure 7.8: Preorder traversal of the tree shown in Figure 7.6. Numbers denote the order in which subcells are solved by the PLACEMENT ORACLE in the top down approach. Note that the children of a node are only solved when the PLACEMENT ORACLE did not find a solution for the node itself.

Subcells which are hard to solve for the PLACEMENT ORACLE are hard to detect before calling the oracle. Many technology dependent details decide whether a given instance is difficult or not. The strategy of the LINEAR ARRANGEMENT PLACER is to try many different subcells which are then combined to a placement of the entire cell. Most of the subcell solutions will not be used in the final result. This waste of runtime is compensated by the possibility to choose subcells which fit well together and discarding unfavorable subcell divisions.

In the DIVIDE PLACER the objective function of the division MIP had two parts: cut value and total cell width. The former is easy to measure without the PLACEMENT ORACLE, whereas the latter can only be estimated very roughly. The LINEAR ARRANGEMENT PLACER does not try to estimate the cell width but simply generates many subcells with small cut value. The cell width is optimized by choosing the right subcells, for which the exact width is known by calling the PLACEMENT ORACLE.

Subcells are created by distributing the FETs to unique positions $1, \dots, |\mathcal{F}|$. This is called a linear arrangement of the FETs. Each interval of positions $[a, b]$ then corresponds to a subcell containing the FETs from positions $a, a+1, \dots, b$. This approach makes it easy to control the cut value between subcells as they are determined by the linear arrangement. We compute a linear arrangement which minimizes the maximum cut k between any two positions. By doing so, we guarantee that a cut of no more than k exists between any two subcells.

In Section 7.3.1 we give the formal definition of the MIN CUT LINEAR ARRANGEMENT PROBLEM and present previous work. We also present the algorithm used in BONNCELL and analyze its performance on real world instances. Section 7.3.2 explains how the LINEAR ARRANGEMENT PLACER uses the subcells generated by the minimum cut linear arrangement to find a placement of the entire cell.

7.3.1 Min Cut Linear Arrangements

We begin by formally defining the MIN CUT LINEAR ARRANGEMENT PROBLEM. The notation used here is adapted and extended from Hamm 2018.

Definition 7.2. A *partial linear arrangement* of a hypergraph $H = (V, E)$ is a pair (X, ϕ), where $X \subseteq V$ is a subset of vertices and ϕ a bijection

$$\phi : X \leftrightarrow \{1, \dots, |X|\}.$$

Definition 7.3. The *cutwidth* of a partial linear arrangement (X, ϕ) *at position* $i \in \{1, \dots, |X|\}$ is defined as

$$\mathrm{cw}(X, \phi, i) := \left|\delta(\phi^{-1}(\{1, \dots, i\}))\right|$$

where $\delta(A)$ denotes the set of hyperedges adjacent to at least one vertex in A and $V \setminus A$.

The *cutwidth* of a partial linear arrangement (X, ϕ) is defined as

$$\mathrm{cw}(X, \phi) := \max_{i=1,\dots,|X|} \mathrm{cw}(X, \phi, i)$$

Definition 7.4. A partial linear arrangement (X, ϕ) can be extended to a partial linear arrangement (X', ϕ'), if

$$X \subseteq X',$$
$$\phi|_X = \phi'|_X.$$

Definition 7.5. A *linear arrangement* of a hypergraph $H = (V, E)$ is a partial linear arrangement (X, ϕ) with $X = V$. We denote the cutwidth of a linear arrangement ϕ by $cw(\phi)$. Similarly, the cutwidth of a linear arrangement at position i is denoted by $cw(\phi, i)$.

This gives us the following problem definition.

MIN CUT LINEAR ARRANGEMENT PROBLEM	
Instance:	Hypergraph $H = (V, E), k \in \mathbb{N}$.
Task:	Find a linear arrangement ϕ of H s.t. $cw(\phi) \leqslant k$, or decide that no such linear arrangement exists.

Previous Work

The classical definition of the MIN CUT LINEAR ARRANGEMENT PROBLEM operates on graphs instead of hypergraphs and is therefore a special case of our definition. This problem is known to be NP-complete, even if vertex degrees are bounded by 3 as shown by Makedon, Papadimitriou, and Sudborough 1985. This restriction is important for our application, as the vertices in instances appearing in BONNCELL correspond to FETs which are connected to at most 3 nets: gate, source, and drain. The instances appearing in BONNCELL usually have small cutwidth, which makes it interesting to look at the complexity parameterized in the maximum cutwidth k. Bevern et al. 2015 have shown that even for hypergraphs one can decide whether a minimum cut linear arrangement with cutwidth at most k exists or not in linear time, assuming that k is a constant. However, their algorithm is not constructive. Recently, Hamm 2018 gave a constructive algorithm with the same runtime guarantee. The algorithm used in BONNCELL is a linear program which does not have these runtime guarantees. Its main advantages are low implementation complexity and extensibility to incorporate additional constraints, e.g. balancedness of PFETs and NFETs. Göke 2015 designed and implemented an initial version of the algorithm but did not publish it.

Algorithm used in BonnCell

The following theorem yields the key idea for the algorithm used in BONNCELL.

Theorem 7.6. *Let $k \in \mathbb{N}$ and ϕ_1, ϕ_2 be partial linear arrangements of the same vertices $C \subseteq V$ with cutwidth at most k. Then ϕ_1 can be extended to a linear arrangement with cutwidth at most k if and only if the same holds for ϕ_2.*

Proof. Assume ϕ_1 can be extended to a linear arrangement with cutwidth at most k. We show that the same holds for ϕ_2. The other direction follows from symmetry. Let ψ_1 be an extension of ϕ_1 with cutwidth at most k. Define $\psi_2 : V \to \{1, \ldots, |V|\}$ such that

$$\psi_2(v) := \begin{cases} \phi_2(v) & v \in C \\ \psi_1(v) & v \notin C\,. \end{cases}$$

Then ψ_2 is a linear arrangement and an extension of ϕ_2 since $\psi_2|_C = \phi_2|_C$ and ψ_1 is an extension of ϕ_1. Furthermore, we have

$$\mathrm{cw}(\psi_2, i) = \mathrm{cw}(\phi_2, i) \leqslant k \text{ for } i \leqslant |C|$$

by definition of ϕ_2. Since

$$\psi_1^{-1}(\{1, \ldots, i\}) = \psi_2^{-1}(\{1, \ldots, i\}) \text{ for } i \geqslant |C|,$$

we also have by definition of $\mathrm{cw}(\psi, i)$ that

$$\mathrm{cw}(\psi_2, i) = \mathrm{cw}(\psi_1, i) \leqslant k \text{ for } i \geqslant |C|.$$

Therefore, ϕ_2 has cutwidth at most k. □

The idea of our algorithm is the following. For increasing i, we calculate all subsets of V with exactly i vertices, s.t. they can be arranged as a partial linear arrangement with cutwidth at most k. As we have just shown, it is sufficient to keep a partial linear arrangement of vertex set X as a representative for all partial linear arrangements of X. Algorithm 7 shows the algorithm in pseudo code.

Theorem 7.7. *Algorithm 7 solves the* Min Cut Linear Arrangement Problem *and can be implemented with runtime* $\mathcal{O}(nm2^n)$*, where* $n := |V|$*,* $m := |E|$*.*

Proof. We prove correctness first. Algorithm 7 has two parts, the first (Lines 1 to 8) computes partial linear arrangements with i elements for $i = 1, \ldots, \lceil \frac{n}{2} \rceil$. It keeps one representative for each subset of vertices X with $|X| = i$ for which a partial linear arrangement of X with cutwidth at most k exists. Computing partial linear arrangements up to half the instance is enough as two of them can be matched to a linear arrangement, as done in the second part (Lines 9 to 11). The critical part is the pruning step in Line 7. This is correct as has been proven in Theorem 7.6.

We show the runtime guarantee next. Before running the algorithm, we initialize a lookup table to be later able to check if $S \in \mathcal{C}_{|S|}$ in $\mathcal{O}(1)$. Since we need a binary entry for each of the $\mathcal{O}(2^n)$ subsets $S \subseteq V$ with $|S| \leqslant \lceil \frac{n}{2} \rceil$, this table needs $\mathcal{O}(2^n)$ time for initialization. Initially, only $\emptyset$ is contained in $\mathcal{C}_0$. We also create a lookup table to check if $|\delta(S)| \leqslant k$ for all $S \subseteq V$ with $|S| \leqslant \lceil \frac{n}{2} \rceil$. Again the table has $\mathcal{O}(2^n)$ binary entries. Each entry takes $\mathcal{O}(nm)$ to compute, resulting in a runtime of $\mathcal{O}(nm2^n)$.

Algorithm 7: MinCutLinearArrangement

input : Hypergraph $H = (V, E)$, $k \in \mathbb{N}$.
output: Linear arrangement ϕ of H s.t. $cw(\phi) \leqslant k$, or that no such linear arrangement exists.

```
1  C_0 ← {∅}                          // Initialize C_0 with the empty list
2  for i = 1, ..., ⌈n/2⌉ do
3      C_i ← ∅                        // C_i contains lists with exactly i elements
4      for S ∈ C_{i−1} do
5          for v ∈ V \ S do
6              S' ← S + {v}                          // append v to the list S'
7              if |δ(S')| ⩽ k and S' ∉ C_i then
8                  C_i ← C_i ∪ {S'}
9  for C ∈ C_⌈n/2⌉ do
10     if C^c ∈ C_⌊n/2⌋ then           // C^c denotes the complement of C
11         return C + REVERSE(C^c)
12 return no such linear arrangement exists
```

Lines 1 to 8 contain three for loops over sets with total cardinality $\mathcal{O}(n2^n)$. All operations inside the loops can be done in $\mathcal{O}(1)$ time using the lookup tables. Lines 9 to 11 contain one for loop iterating over $\mathcal{O}(2^n)$ elements. Computing the complement takes $\mathcal{O}(n)$ time. The total runtime is therefore dominated by lookup table creation, i.e. $\mathcal{O}(nm2^n)$. □

The number of partial linear arrangements considered by the algorithm can indeed be $\Omega(2^n)$. Consider the instance with n vertices and a single edge, connecting all vertices and $k = 1$. Every linear arrangement has cutwidth exactly 1 and each subset of vertices is the representative of a partial linear arrangement with cutwidth at most 1. Therefore, at least half of the subsets of V are considered which gives $\Omega(2^n)$. This example is somewhat degenerate as it uses a very large edge connecting all vertices. The bound on the maximum number of vertex subsets the algorithm considers can be improved to $\mathcal{O}(2^{k(\log n+s)})$, where s is the maximum size of an edge. However, real world instances with $k = 5$, $s = 12$, and $n \approx 30$ have been observed which can also be solved within seconds despite the poor worst case bound. It seems that the industry instances obey some yet undiscovered structure which bounds the number of solution candidates.

Our application in BonnCell requires some modifications of Algorithm 7. In Line 7 of Algorithm 7 the first partial linear arrangement with vertex subset S' is kept. In practice some linear arrangements are preferable to others. We are not only interested in the maximum cut value but, given two linear arrangements with equal maximum cut, we prefer the one which has lower cuts on other positions. When we

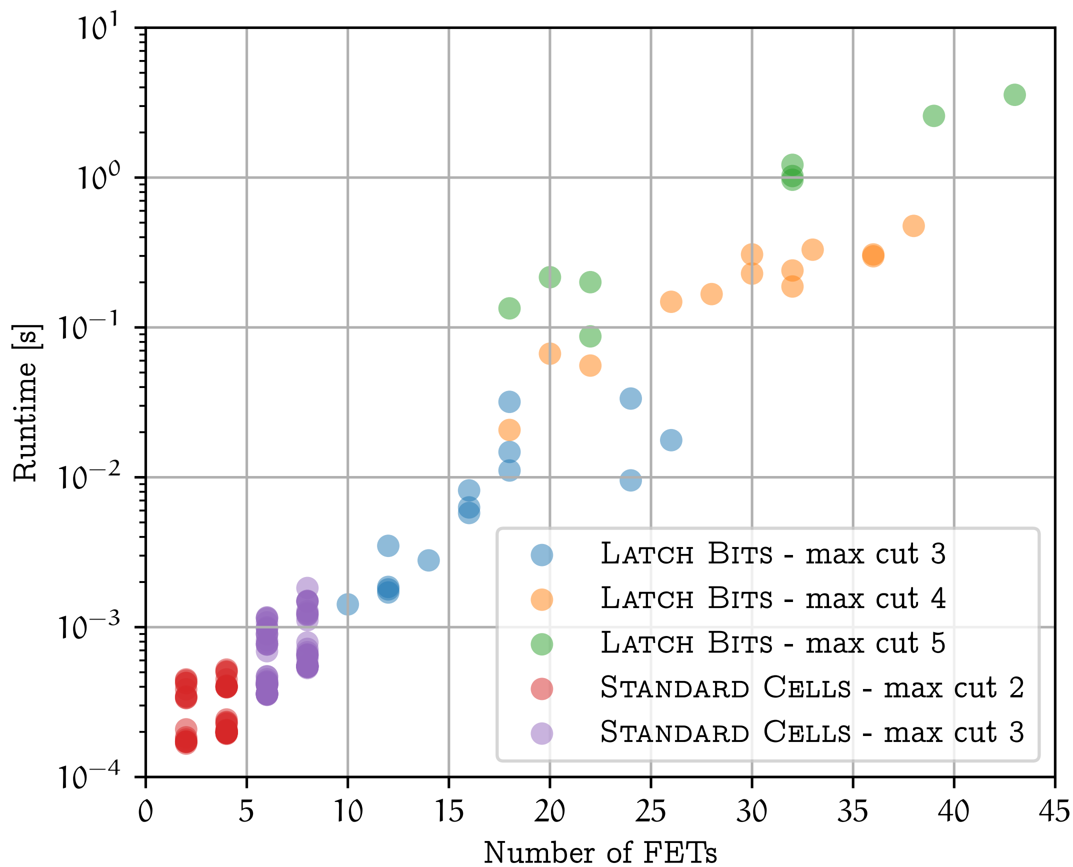


Figure 7.9: Runtime of MINCUTLINEARARRANGEMENT algorithm on STANDARD CELLS and LATCH BITS instances. All instances were solved within a few seconds. Larger cut values result in higher running time.

compare two partial linear arrangements ϕ, ϕ' on the same vertex set S, we keep the one which minimizes

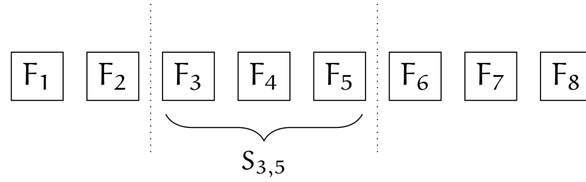
$$f(S, \phi) := \sum_{i=1}^{|S|} e^{cw(S,\phi,i)}.$$

The same objective function is also minimized when we select a complement in Lines 9 to 11. Note that this does not guarantee that $f(S, \phi)$ is globally minimized, as the local optimum chosen in Line 7 cannot necessarily be extended to a global optimum.

Figure 7.9 shows the runtime of our implementation on the STANDARD CELLS and LATCH BITS instances. All instances are solved within a few seconds and maximum cut values k are between 2 and 5 (including).

7.3.2 Algorithm of Linear Arrangement Placer

Assume we have already found a min cut linear arrangement with FETs in order $F_1, \ldots, F_n$.



The next step is to choose and solve subcells from this linear arrangement and choose placed subcells, s.t. the resulting placement for the entire cell is small and routable. This is done using the dynamic programming algorithm LinearArrangement-Placer (Algorithm 8).

Algorithm 8: LinearArrangementPlacer

input : FETs $F_1, \ldots, F_n$ in order of min cut linear arrangement
output: Routable placement P

```
1  w_t ← {0, if t = 0, else ∞}
2  for t := 1,...,n do
3      for s := t,...,1 do
4          R ← Solve(S_{s,t}, P_{s-1})          // Solve S_{s,t} with P_{s-1} as neighbor
5          if R = null then
6              continue
7          else if w(P_{s-1} ∪ R) < w_t then
8              P_t ← P_{s-1} ∪ R
9              w_t ← w(P_{s-1} ∪ R)
10 return P_n
```

The algorithm works as follows: w_t contains the width of the currently known smallest placement of $F_1, \ldots, F_t$ and P_t the corresponding placement. As described in the introduction of Chapter 7, the Placement Oracle needs to know the placement of the left neighbor subcell to guarantee routability. It is therefore important that P_{s-1} is not updated after subcell $S_{s,t}$ has been solved. Solving $S_{1,n}$ corresponds to solving the entire cell as one large subcell. We must therefore expect that some subcells will not be solvable within the user given time limit T and we need to distribute T to the Placement Oracle calls. Each iteration of the for loop starting in Line 2 gets at least T/n of the total runtime. The first iteration will start with a runtime limit of T/n. As only the subcell $S_{1,1}$ containing a single FET F_1 will be solved in this iteration, we expect that only a small fraction of the runtime will actually be used. We therefore distribute the remaining runtime equally to the remaining iterations. The same is done for all consecutive iterations. As the last Placement Oracle call solves the entire cell $S_{1,n}$, we can make sure that all unused runtime will be spent on this instance. This means if the algorithm finishes early, it has also solved the entire instance optimally, a nice property which the Divide Placer does not have. The remaining runtime within

an iteration of Line 2 is given entirely to each PLACEMENT ORACLE call. The idea is that the complexity of each call should increase as the number of FETs increases, s.t. this approach maximizes the number of solved instances within the given time limit. However, we have observed that in practice some instances get solved faster after more FETs have been added. Therefore, other strategies to distribute runtime might be more successful.

Runtime Improvements

For all PLACEMENT ORACLE calls with $s < t$ in Line 4 of Algorithm 8, we already know a placement solution for FETs $F_1, \dots, F_t$. We can limit the placement search space of the PLACEMENT ORACLE by an upper bound on the placement width which guarantees that the condition $w(P_{s-1} \cup R) < w_t$ of Line 7 holds for any given solution R. This also avoids PLACEMENT ORACLE calls which are intuitively undesirable: solving the subcell $S_{2,t}$ consisting of FETs $F_2, \dots, F_t$, i.e. only splitting off F_1 to the left, seems a bad choice. The first subcell $S_{1,1}$ will have no FETs on one stack and therefore in most cases waste space. Having solved $S_{1,1}$ and other subcells which can be concatenated to a placement of $F_1, \dots, F_k$ already, we get an upper bound on the width of $S_{2,k}$ which would improve the currently best solution found. In practice, this upper bound is often low, matching intuition, and proving that no sufficiently small solution of $S_{2,t}$ exists, takes only a very small amount of time. However, this does not work for the other direction. Subcells $S_{s,n-1}$ will be tried with high effort, although only a single FET F_n is missing. Here it would help to have $S_{n,n}$ already solved, so that an upper bound for the width of $S_{s,n-1}$ can be used just as previously. This does not work, as PLACEMENT ORACLE calls of $S_{n,n}$ require the placement of $S_{1,n-1}$, resulting in a circular dependency.

This technique helped to reduce runtime of the LINEAR ARRANGEMENT PLACER on LATCH BITS instances dramatically as shown in Figure 7.10. As explained above, in the normal use case of the LINEAR ARRANGEMENT PLACER the algorithm will run into a timeout. This means that saved runtime is invested in further PLACEMENT ORACLE calls to improve the quality of the final placement.

7.4 Results

Figure 7.11 shows detailed results of all big cell modes in comparison to lower bounds on the LATCH BITS testbed. For each of the 37 instances, the three big cell modes (DIVIDE PLACER with bottom up approach, DIVIDE PLACER with top down approach, and LINEAR ARRANGEMENT PLACER) and the default placer have been run. For most instances, the default placer does not return a valid placement but only a lower bound on the width of an area optimum placement. For each cell, w_{best} denotes the minimum placement width which has been found by the three big cell modes. For 33 out of 37 instances the LINEAR ARRANGEMENT PLACER solution had minimum width. For 6 instances the best solution found met the lower bound and is therefore optimal. The scatter plot shows one point for each instance and mode. The x coordinate is taken from

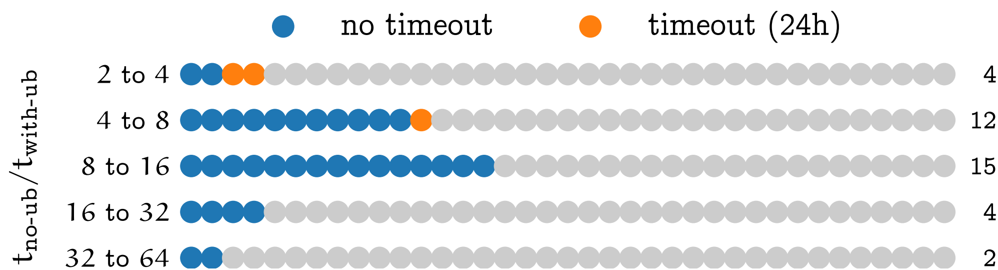


Figure 7.10: Histogram of runtime savings on LATCH BITS due to upper bound on maximum subcell width as described in Section 7.3.2. Runtime limit has been set to 24h and subcell width upper bounds have not been used. The time used by this mode is $t_{\text{no-up}}$. For all but 3 instances, the algorithm reached a timeout, i.e. $t_{\text{no-ub}} = 24\text{h}$. In this flow runtime that would have *not* been saved with subcell width upper bounds has been measured and is denoted by $t_{\text{width-ub}}$. Speedup is measured for each instance as $t_{\text{no-ub}}/t_{\text{with-ub}}$. The speedup of the entire testbed is 8.5. With active upper bounds the saved runtime will be used to try more subcells, thus improving the found placement instead of finishing early.

the width of the optimum solution, i.e. for a single instance the points for the 4 modes have the same x coordinate (cf. $w_{\text{best}} = 46$). Points with same x and y coordinates are stacked to show the number of points of each color. The y coordinate gives the relative placement width of a mode compared to the best solution for this cell. A green point with $y = 1.0$ denotes that the linear arrangement mode gave an optimum placement for this cell. It can be seen that the linear arrangement is dominant for various cell widths. The divide bottom up approach is slightly better compared to the divide top down approach, especially for larger cell widths. The gap between lower bound and best solution increases with the cell widths. It is unclear if this is due to inaccurate lower bounds or poor big cell solution quality. All cell widths are even numbers of PC tracks, as this is given by the cell image.

Figure 7.11: Results of big cell modes on LATCH BITS instances. The full description is given in Section 7.4.

Chapter 8

Comparison to Manual Layouts

8.1 Standard Cells

We were able to compare all 126 STANDARD CELLS against the library of manual layouts created by human experts. BONNCELL was able to solve all these cells *without* the big cell modes presented in Chapter 7 and therefore with minimum cell width. The average runtime needed per cell using a single thread was less than 2 minutes.

minimum area layout	# cells
BONNCELL	10
equal	110
human expert	6

In some cases the BONNCELL result improved the library results by more than 25%. Figure 8.1 shows an example of such a cell. For 6 cells the manual layout was better than the BONNCELL layout. BONNCELL was not able to find the solutions found by the experts because it classified them as illegal. In all these cells a technique was used which allows to leave some nets unconnected, i.e. the terminals of these nets do not have to be connected with wires. This is possible if, for a given placement, the terminals already have the same electric potential. Connecting these terminals is redundant since the wire would not transport any current. BONNCELL is currently not able to detect this kind of situation.

8.2 Latches

Latches are much larger and more complex cells. Manually finding good layouts for these cells is a challenging task, even for very experienced designers, taking days or weeks. We compared BONNCELL solutions on a testbed of 22 latches ranging in size between 10 and 72 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. In one case both BONNCELL and the designer found an area optimum solution without M2 usage.

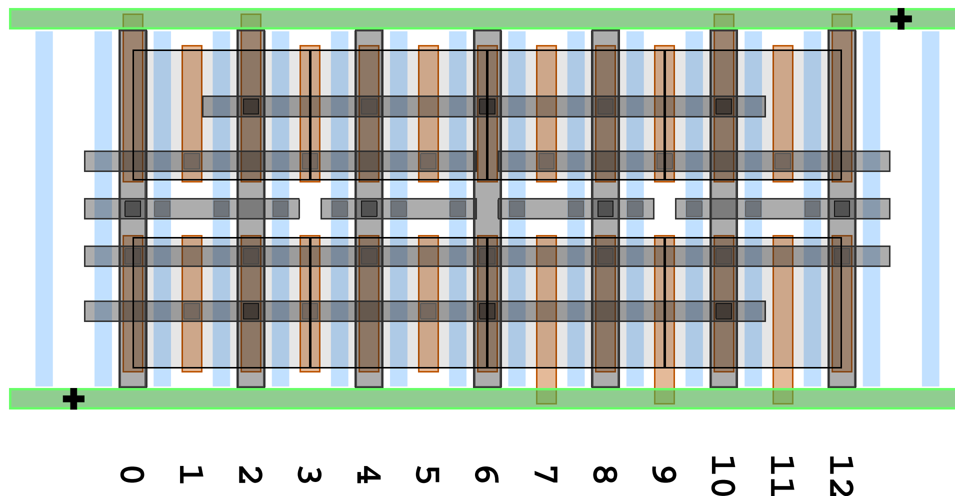


Figure 8.1: BonnCell layout of a OAI22 X4M instance which improves upon the manual layout by more than 25%.

latch	level	# FETs	# nets	runtime [h:mm:ss]	improved[1]
DFFFQ	X1M	38	27	2:45:56	yes
DFFQDICE	X1M	43	29	1:39:59	yes
DFFQ	X1M	28	21	1:12:31	yes
ELATN	X1M	12	11	21:11	yes
ELATS	X1M	10	11	17:28	yes
ELAT	X1M	12	11	20:19	yes
ELAT	X3M	12	11	54:44	yes
ELAT	X8M	12	11	55:38	yes
ESLATN	X1M	32	25	1:52:08	yes
ESLATS	X1M	26	25	1:10:18	yes
ESLAT	X1M	32	25	1:36:03	yes
ESLAT	X3M	32	25	1:10:49	yes
L1LATF	X1M	26	21	1:09:15	both optimum
N1LAT	X1M	38	27	2:12:42	yes
N1LAT	X3M	38	27	1:14:45	yes
SDFFQN	X1M	36	28	2:31:19	yes
SDFFQS	X1M	32	27	8:47:09	yes
SDFFQ	X1M	36	28	2:07:39	yes
SDFFQ	X3M	36	28	1:12:31	yes
SDFFSRPQ	X1M	44	34	1:22:14	yes
INVELAT	X1M	14	12	1:00:56	yes
INVELAT	X3M	14	12	10:54	yes

An example layout for a latch that BonnCell built using 5% less area than the designer's solution is shown in Figure 8.2. Using the approach described in Section 7.1, BonnCell can also build multi bit layouts of cells. Figure 7.2 on page 80 shows a layout of the same latch with two bits.

[1]Has BonnCell improved the manual layout by either area usage or number of used M2 tracks?

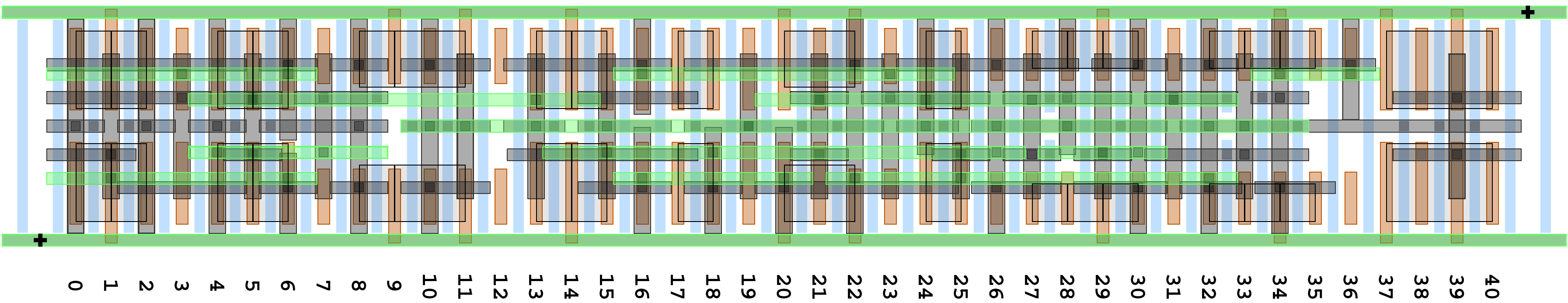


Figure 8.2: BonnCell's layout of the latch SDFFQ X3M that needs 5% less area than the best designer's solution while using the same number of M2 tracks.

8.3 PLCB

In this section we analyze the performance of BonnCell on the PLCB instance. The PLCB consists of 128 FETs and 89 nets and is built on 4 bits. Without the use of BonnCell, the size of the instance makes it impossible to be built by a single engineer within a reasonable amount of time. Therefore, the instance is split into several subinstances, which are then built separately by a team of engineers. These subinstances are not independent, as interfaces at the cell boundaries have to match. Initially it is not clear what the interfaces could look like, therefore several iterations of building cells and modifying them to have compatible wiring at the boundary have to be run. In total, it is a very time consuming task, taking weeks for an entire team.

The PLCB can also be solved by BonnCell's big cell modes. As for the latches, the Linear Arrangement Placer returned the best results.

BonnCell	72h	4×58 tracks
Team of human experts	several weeks	4×68 tracks

The final subcell placement of the PLCB is shown in Figure 8.3, the full placement and routing solution of the PLCB is shown in Figures 8.4 to 8.6.

The PLCB serves as a good alternative benchmark for the effectiveness of various speed up methods discussed in this thesis. Although it is only a single instance, over 1000 subcells have been solved within a single call of the Linear Arrangement Placer to find a good solution of the entire cell. Figures 8.7 and 8.8 show a comparison of the PLCB runs with different settings applied.

Figure 8.3: Final subcell placement of the PLCB. It can clearly be seen that the bit assignment is not yet optimum as there is a lot of unused space on the bottom right. However, this solution is 10 tracks smaller in width compared to the human expert solution which translates into almost 15% area reduction.

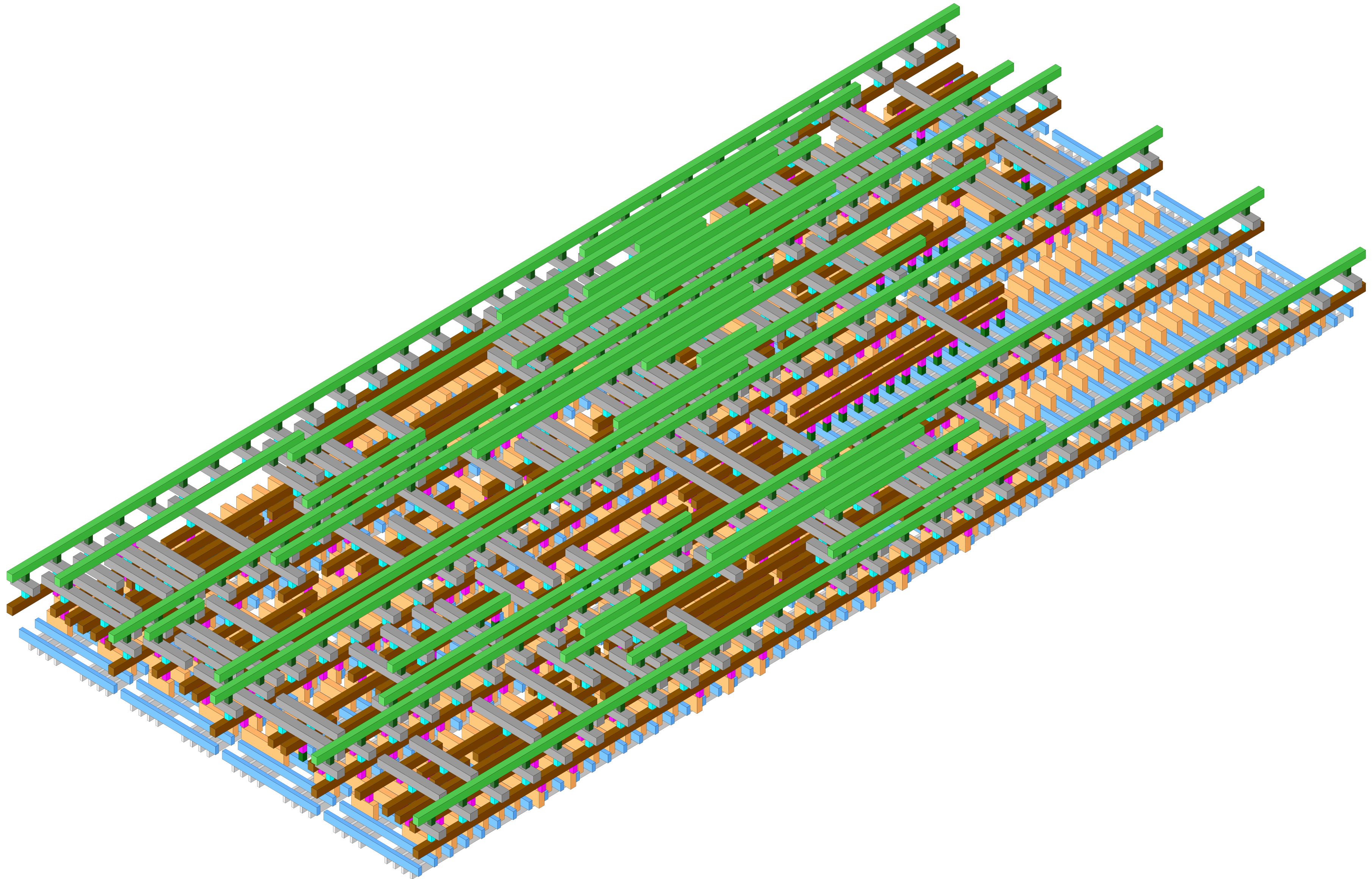


Figure 8.4: Routing of the PLCB placement shown in Figure 8.3 in isometric projection.

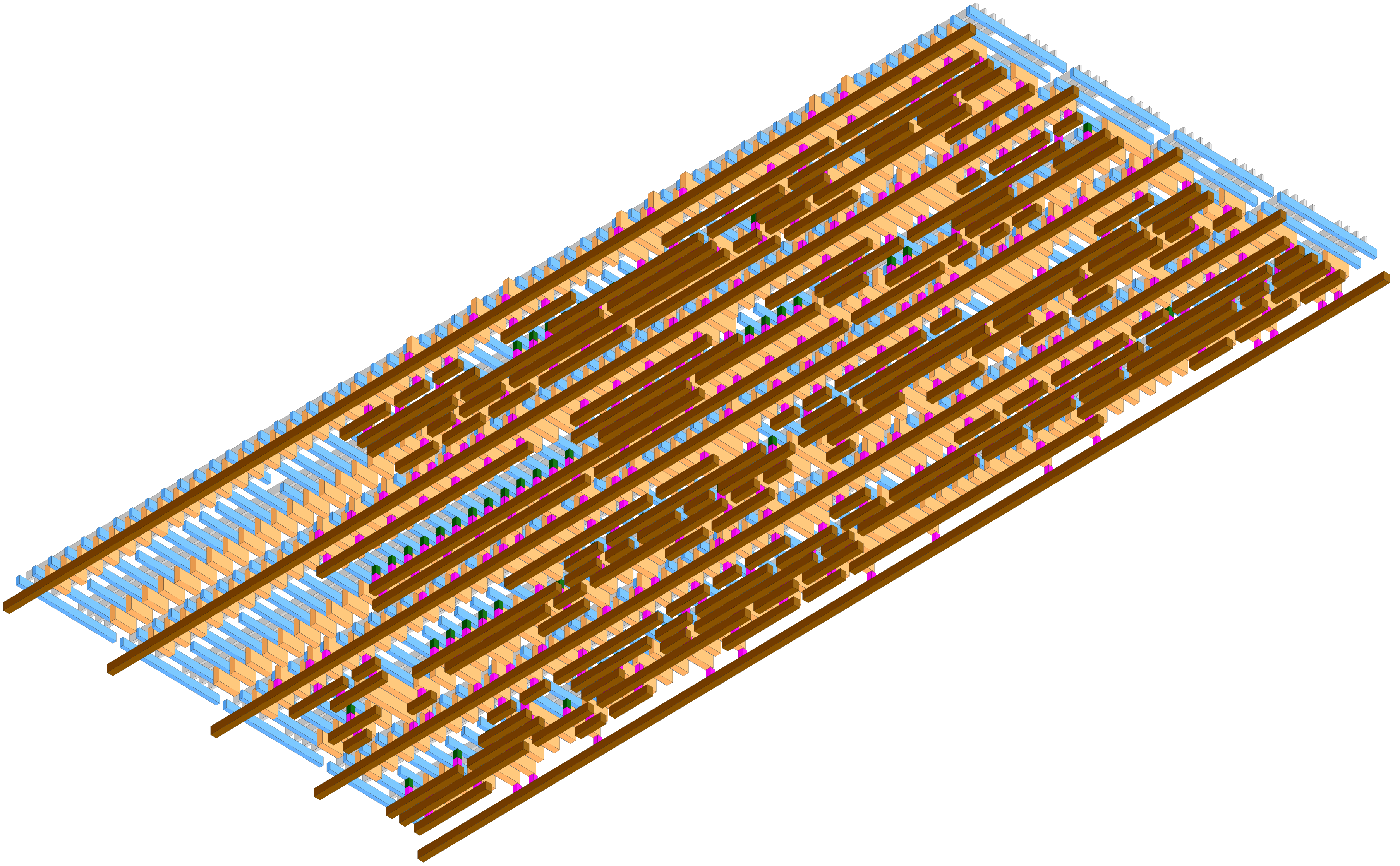


Figure 8.5: Same routing as of Figure 8.4 shown in isometric projection up to layer M0.

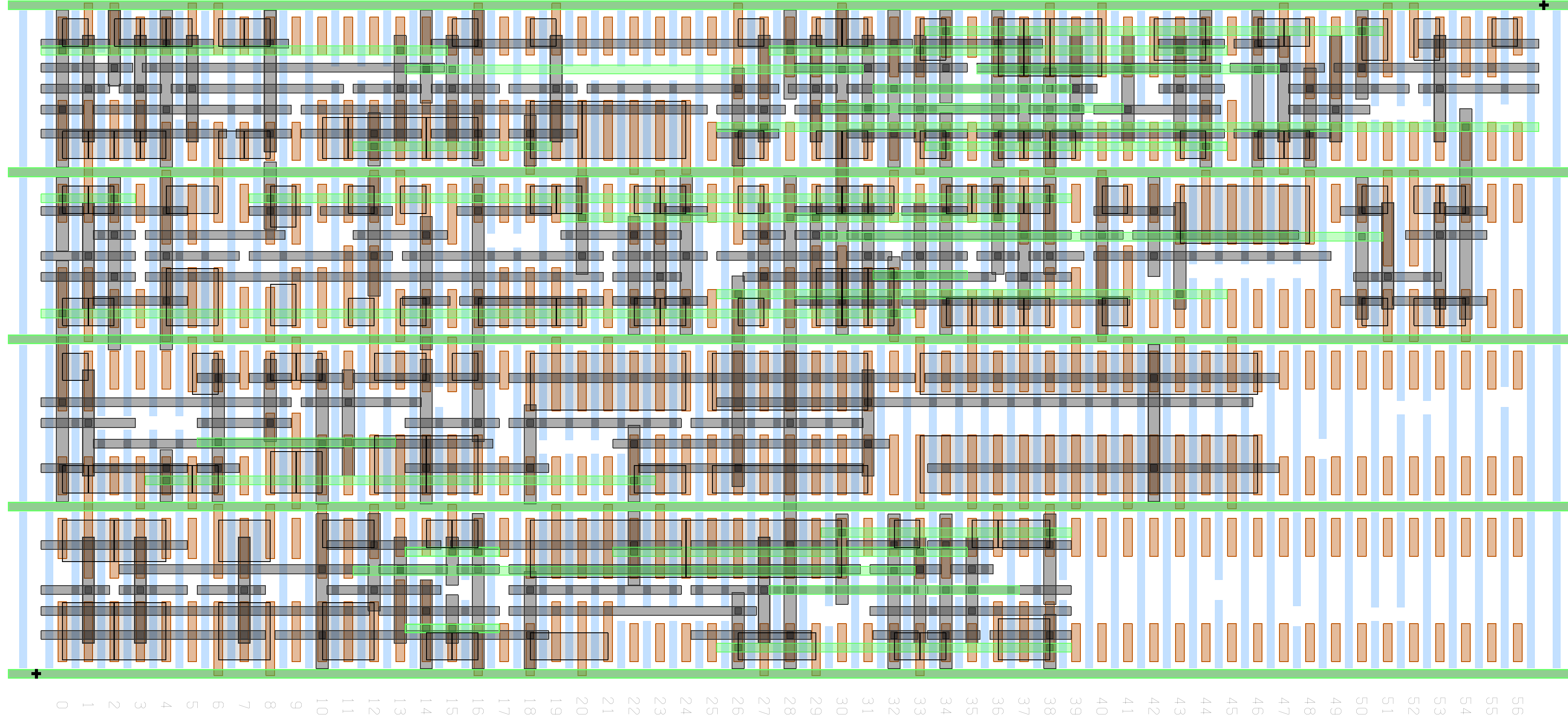


Figure 8.6: Same routing as of Figure 8.4 in z projection.

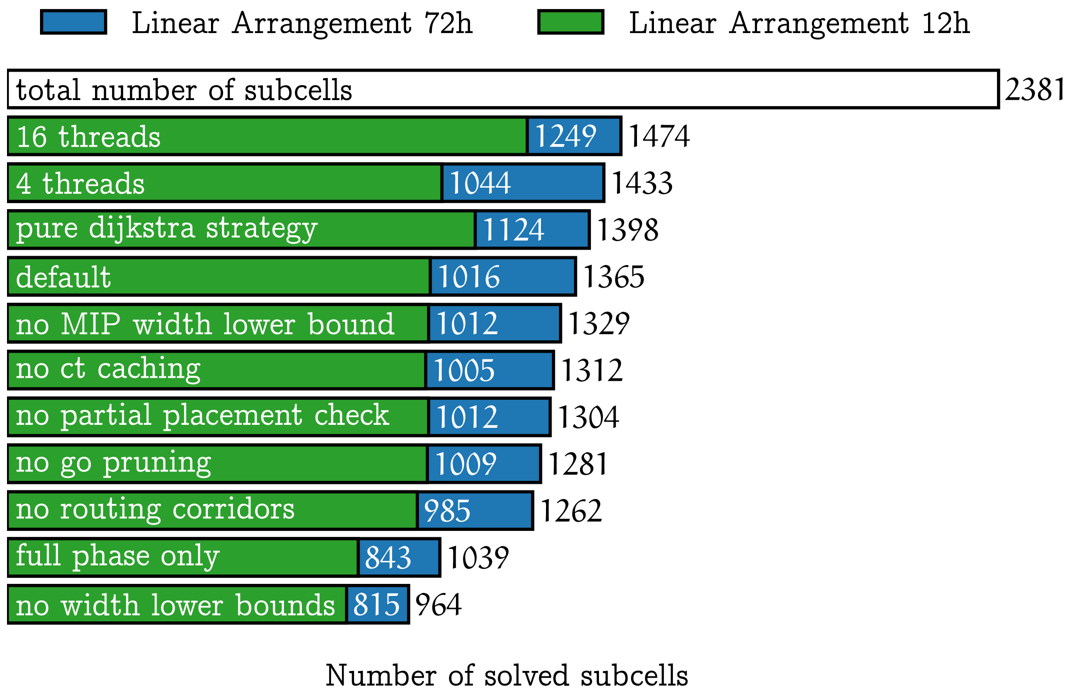


Figure 8.7: Comparison of number of solved subcells of the Linear Arrangement Placer on the PLCB. Each row shows two runs with runtime limits of 12h and 72h. Numbers of solved subcells with 72h runtime limit were always larger compared to 12h runtime limit, therefore the blue bar only shows the additional number of solved subcells. Effect of runtime improving features is tested by deactivating a single feature and activating all others. For example, "without FEOL oracle caching" has the same settings as default but does not use FEOL oracle caching. More solved subcells allow for better placement of the entire instance. However, a faster Placement Oracle does not guarantee that more subcells can be solved within a given time limit. As the bits are solved one after the other, a different routing solution for a lower bit has influence on the subcells solved in the upper bits. It can clearly be seen that without lower bounds on the placement width significantly fewer subcells can be solved. The pure Dijkstra enumeration strategy seems to be beneficial for solving the PLCB. A possible explanation is that most of the subcells solved by the Linear Arrangement Placer finish without timeout. In this case the Dijkstra enumeration is the fastest enumeration strategy as discussed in Section 5.7.

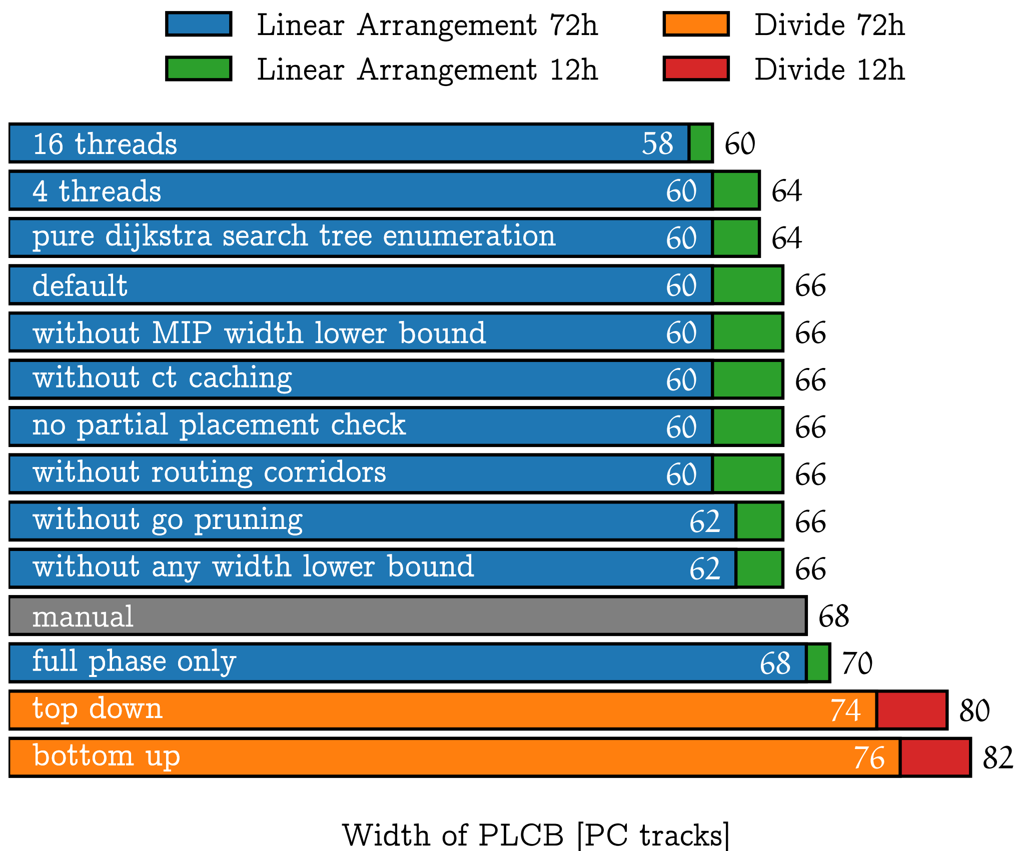


Figure 8.8: Width comparison of the PLCB with results from Divide Placer and Linear Arrangement Placer. Results for Linear Arrangement Placer are notably better than for the Divide Placer. Manual layout using 68 tracks has taken weeks for a team of human experts. The best layout with a width of 58 tracks is obtained by using the Linear Arrangement Placer with 16 threads and 72h runtime.

Chapter 9

Summary

Cell layout is a critical step in the design process of computer chips. A cell is a logic function or storage element implemented in CMOS technology by transistors connected with wires. As each cell is used many times on a chip, improvements of a single cell layout can have a large effect on the overall chip performance. In the past years increasing difficulty to manufacture small feature sizes has lead to growing complexity of design rules. Producing cell layouts which are compliant with design rules and at the same time optimized w.r.t. layout size has become a difficult task for human experts.

In this thesis we present BonnCell, a cell layout generator which is able to fully automatically produce design rule compliant layouts. It is able to guarantee area minimality of its layouts for small and medium sized cells. For large cells it uses a heuristic which produces layouts with a significant area reduction compared to those created manually.

The routing problem is based on the Vertex Disjoint Steiner Tree Packing Problem with a large number of additional design rules. In Chapter 4 we present the routing algorithm which is based on a mixed integer programming (MIP) formulation that guarantees compliance with all design rules. The algorithm can also handle instances in which only part of the transistors are placed to check whether this partial placement can be extended to a routable placement of all transistors.

Chapter 5 contains the transistor placement algorithm. Based on a branch and bound approach, it places transistors in turn and achieves efficiency by pruning parts of the search tree which do not contain optimum solutions. One major contribution of this thesis is that BonnCell only outputs routable placements. Simply checking the routability for each full placement in the search tree is too slow in practice, therefore several speedup strategies are applied.

Some cells are too large to be solved by a single call of the placement algorithm. In Chapter 7 we describe how these cells are split up into smaller subcells which are placed and routed individually and subsequently merged into a placement and routing of the original cell. Two approaches for dividing the original cell into subcells are presented, one based on estimating the subcell area and the other based on solving the Min Cut Linear Arrangement Problem.

BonnCell has enabled our cooperation partner IBM to drastically improve their cell design and layout process. In particular, a team of human experts needed several weeks to find a layout for their largest cell, consisting of 128 transistors. BonnCell processed this cell without manual intervention in 3 days and its layout uses 15% less area than the layout found by the human experts.

Appendix A

Testbeds

To evaluate the performance of our algorithms, different testbeds of cell instances are used. They originate from our industry partner and are real world 7nm instances. All evaluations have been run on an AMD EPYC 7601 at 2.2GHz.

A.1 Standard Cells

There are 126 instances in the STANDARD CELLS testbed which have 2 to 8 FETs and represent simple logic cells: INV, NAND, NOR, AOI, and OAI. AOI and OAI gates implement complex logical functions which in CMOS technology can be built more effectively compared to the sum of their parts. The cells implement the following functions.

cell type	# inputs	function
INV	1	$\neg A$
NAND	2	$\neg(A \wedge B)$
NOR	2	$\neg(A \vee B)$
AOI21	3	$\neg(A \vee (B \wedge C))$
AOI22	4	$\neg((A \wedge B) \vee (C \wedge D))$
OAI21	3	$\neg(A \wedge (B \vee C))$
OAI22	4	$\neg((A \vee B) \wedge (C \vee D))$

The testbed contains multiple variants of each cell type. This includes different power levels and different FET sizes. These realize different trade offs between power consumption, circuit speed (timing), and cell size. Table A.1 summarizes all instances of the STANDARD CELLS testbed. BONNCELL is fast enough to solve all STANDARD CELLS instances optimally within a few minutes.

A.2 Latches

Latches, or flip-flops, are much more complex and larger instances compared to the STANDARD CELLS. Their main function is to store information. In contrast to the standard logic gates, the output of a latch does not only depend on its input but also on its current state, i.e. the stored information. Latches exist in different variations

cell type	# cells	# FETs	# nets
AOI21	11	6	9
AOI22	11	8	11
INV	17	2	5
NAND2	16	4	7
NAND3	11	6	9
NAND4	5	8	11
NOR2	16	4	7
NOR3	8	6	9
NOR4	5	8	11
OAI21	10	6	9
OAI22	10	8	11

Table A.1: Standard cell instances. Each row shows number of instances, number of FETs, and number of nets for a given cell type.

and with different power levels. In total our testbed contains 26 different instances, cf. Table A.2.

All but 4 latch instances are too large to be solved by the BonnCell core routine. Therefore, the big cell modes (Chapter 7) are used to solve these instances. The big cell modes work by solving many subcells consisting of some of the FETs of the instance. Runtime improvements on these subcells correspond to improvements of the overall big cell mode performance. Therefore, we extract subcells created by the Linear Arrangement Placer (Section 7.3) and use them as the Latch Subcells testbed. These instances are all subcells being solved in a single bit Linear Arrangement Placer run with 19 hours of runtime limit. Subcells contain left and right pins for each net which needs to connect a neighboring subcell but do not set specific tracks for these pins to reduce the dependency on a specific routing solution. For each latch there are between 1 and 9 subcells. In total this process generated 100 subcell instances.

Of the latch instances, 11 are too large for a single circuit row. Those instances are built as double bit instances using two circuit rows. We use our multibit algorithm presented in Section 7.1 to split these cells into two single bit instances with M1 pins in the direction of the other bit. Again, the pins only force connections to the direction of the other bit instead of specific tracks in order to avoid that these instances depend on BonnCell routings. Together with the 15 original single bit latch instances, the 22 instances from the 11 double bit latches yield the Latch Bits testbed with 37 instances.

cell type	power level	# FETs	# nets	# bits	# subcells
DFFFQ	X1M	38	27	1	2
DFFFQ2	X1M	60	40	2	6
DFFFQDICE	X1M	72	42	2	6
DFFQ	X1M	28	21	1	2
DFFQDICE	X1M	43	29	1	3
ELAT	X1M	12	11	1	1
ELAT	X3M	12	11	1	2
ELAT	X8M	12	11	1	2
ELATN	X1M	12	11	1	1
ELATS	X1M	10	11	1	1
ESLAT	X1M	32	25	1	4
ESLAT	X3M	32	25	2	5
ESLAT	X8M	32	25	2	6
ESLATN	X1M	32	25	1	4
ESLATS	X1M	26	25	1	3
L1LATF	X1M	26	21	1	3
L2SFF	X1M	48	37	2	5
N1LAT	X1M	38	27	2	5
N1LAT	X3M	38	27	2	5
N1LAT	X8M	38	27	2	9
SDFFQ	X1M	36	28	1	3
SDFFQ	X3M	36	28	2	4
SDFFQDICE	X1M	64	42	2	8
SDFFQN	X1M	36	28	1	1
SDFFQS	X1M	32	27	1	4
SDFFSRPQ	X1M	44	34	2	5

Table A.2: There are several testbeds originating from the latch instances. The LATCH testbed contains 15 single and 11 double bit instances, one for each row of the table. The LATCH BITS testbed contains the original single bit latches and two separate single bit instances for each double bit latch. The DOUBLE BIT LATCH testbed contains the 11 double bit instances. The LATCH SUBCELLS testbed contains single bit subcells which have been created by the LINEAR ARRANGEMENT PLACER. The # bits and # subcells columns show how many instances each latch contributes to the respective testbed. In total there are 26 LATCH, 11 DOUBLE BIT LATCH, 37 LATCH BITS, and 100 LATCH SUBCELLS instances.

Bibliography

W. H. Arnold (2009). "Double-Patterning Lithography". In: *Journal of Micro / Nanolithography, MEMS, and MOEMS* 8.1 (cit. on p. 7).

R. Bar-Yehuda, J. A. Feldman, R. Y. Pinter, and S. Wimer (1989). "Depth-First-Search and Dynamic Programming Algorithms for Efficient CMOS Cell Generation". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8, pp. 737–743 (cit. on p. 6).

K. S. Berezowski (2001). "Transistor Chaining with Integrated Dynamic Folding for 1-D Leaf Cell Synthesis". In: *Euromicro Symposium on Digital Systems Design*, pp. 422–429 (cit. on p. 4).

R. van Bevern, R. G. Downey, M. R. Fellows, S. Gaspers, and F. A. Rosamond (2015). "Myhill–Nerode Methods for Hypergraphs". In: *Algorithmica* 73.4, pp. 696–729 (cit. on p. 89).

M. Burkhardt, J. Arnold, Z. Baum, S. Burns, J. Chang, J. Chen, J. Cho, V. Dai, Y. Deng, S. Halle, et al. (2009). "Overcoming the Challenges of 22-nm Node Patterning Through Litho-Design Co-Optimization". In: *Optical Microlithography XXII* (San Jose, California, USA). Vol. 7274 (cit. on p. 7).

J. Cortadella, J. Petit, S. Gomez, and F. Moll (2014). "A Boolean Rule-Based Approach for Manufacturability-Aware Cell Routing". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.3, pp. 409–422 (cit. on p. 6).

P. Cremer, S. Hougardy, J. Schneider, and J. Silvanus (2017). "Automatic Cell Layout in the 7nm Era". In: *Proceedings of the 2017 International Symposium on Physical Design*. ISPD (Portland, Oregon, USA) (cit. on p. 8).

Y. Du, Q. Ma, H. Song, J. Shiely, G. Luk-Pat, A. Miloslavsky, and M. D. Wong (2013). "Spacer-Is-Dielectric-Compliant Detailed Routing for Self-Aligned Double Patterning Lithography". In: *Proceedings of the 50nd Design Automation Conference*. DAC (Austin, Texas, USA), pp. 1–6 (cit. on p. 7).

S.-Y. Fang (2015). "Cut Mask Optimization with Wire Planning in Self-Aligned Multiple Patterning Full-Chip Routing". In: *Proceedings of the 20th Asia and South Pacific Design Automation Conference*. ASP-DAC (Chiba / Tokyo, Japan), pp. 396–401 (cit. on p. 7).

D.-S. Fu, Y.-Z. Chaung, Y.-H. Lin, and Y.-L. Li (2009). "Topology-Driven Cell Layout Migration with Collinear Constraints". In: *Proceedings of the International Con-*

ference on Computer Design: VLSI in Computers and Processors. ICCD (Lake Tahoe, California, USA), pp. 439–444 (cit. on p. 3).

J.-R. Gao and D. Z. Pan (2012). "Flexible Self-Aligned Double Patterning Aware Detailed Routing with Prescribed Layout Planning". In: *Proceedings of the 2012 International Symposium on Physical Design*. ISPD (Napa, California, USA), pp. 25–32 (cit. on p. 7).

M. X. Goemans and D. P. Williamson (1995). "A General Approximation Technique for Constrained Forest Problems". In: *SIAM Journal on Computing* 24.2, pp. 296–317 (cit. on p. 19).

A. Göke (2015). *First Version of Min Cut Linear Arrangement Algorithm (unpublished)* (cit. on p. 89).

I. Griva, S. G. Nash, and A. Sofer (2009). *Linear and Nonlinear Optimization* (cit. on p. 21).

M. Grötschel, A. Martin, and R. Weismantel (1997). "The Steiner Tree Packing Problem in VLSI Design". In: *Mathematical Programming* 78, pp. 265–281 (cit. on p. 19).

A. Gupta and J. P. Hayes (1998). "Optimal 2-D Cell Layout with Integrated Transistor Folding". In: *Proceedings of the 1998 International Conference on Computer-Aided Design*. ICCAD (San Jose, California, USA), pp. 128–135 (cit. on p. 5).

M. Guruswamy, R. L. Maziasz, D. Dulitz, S. Raman, V. Chiluvuri, A. Fernandez, and L. G. Jones (1997). "Cellerity: A Fully Automatic Layout Synthesis System for Standard Cell Libraries". In: *Proceedings of the 34th Design Automation Conference*. DAC (Anaheim, California, USA) (cit. on p. 6).

H. Haffner, J. Meiring, Z. Baum, and S. Halle (2007). "Paving the Way to a Full Chip Gate Level Double Patterning Application". In: *27th Annual BACUS Symposium on Photomask Technology* (Monterey, California, USA) (cit. on p. 7).

H. Haffner, J. Meiring, Z. Baum, S. Halle, and S. Mansfield (2008). "Solving the Gate ACLV & ADLV Challenges with Printing Assist Features". In: *Microlithography World* 17.2, pp. 7–11 (cit. on p. 7).

T. Hamm (2018). "Finding Linear Arrangements of Hypergraphs with Bounded Cutwidth in Linear Time". Master Thesis. Research Institute for Discrete Mathematics, University of Bonn (cit. on pp. 88, 89).

M. Harper, B. Weinstein, C. Simon, W. Morgan, V. Knight, N. Swanson-Hysell, M. Evans, M. Greco, and G. Zuidhof (2015). *python-ternary: Ternary Plots in Python*. Zenodo (cit. on p. 37).

D. D. Hill (1985). "Sc2: A hybrid automatic layout system". In: *Proceedings of the 1985 International Conference on Computer-Aided Design*. ICCAD (San Jose, California, USA) (cit. on p. 4).

N.-D. Hoàng and T. Koch (2012). "Steiner Tree Packing Revisited". In: *Mathematical Methods of Operations Research* 76.1, pp. 95–123 (cit. on p. 20).

S. Hougardy, T. Nieberg, and J. Schneider (2013). "BonnCell: Automatic Layout of Leaf Cells". In: *Proceedings of the 18th Asia and South Pacific Design Automation Conference*. ASP-DAC (Yokohama, Japan), pp. 453–460 (cit. on p. 8).

T. Iizuka (2006). "Optimal Layout Synthesis of Standard Cells in Large Scale Integration". PhD thesis. Department of Electronic Engineering, Graduate School of Engineering, The University of Tokyo, Tokyo, Japan (cit. on pp. 4–6).

K. Jo, S. Ahn, T. Kim, and K. Choi (2018). "Cohesive Techniques for Cell Layout Optimization Supporting 2D Metal-1 Routing Completion". In: *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*. ASP-DAC (Jeju Island, Korea), pp. 500–506 (cit. on p. 6).

I. Kang, D. Park, C. Han, and C.-K. Cheng (2018). "Fast and precise routability analysis with conditional design rules". In: *System Level Interconnect Prediction Workshop* (San Francisco, California, USA), p. 4 (cit. on p. 6).

R. Karmazin, C. T. O. Otero, and R. Manohar (2013). "Celltk: Automated Layout for Asynchronous Circuits with Nonstandard Cells". In: *19th International Symposium on Asynchronous Circuits and Systems*, pp. 58–66 (cit. on p. 6).

R. M. Karp (1972). "Reducibility Among Combinatorial Problems". In: *Complexity of Computer Computations*, pp. 85–103 (cit. on p. 49).

B. Klotz (2018). "Faster Leaf Cell Placement Algorithms". Master Thesis. Research Institute for Discrete Mathematics, University of Bonn (cit. on pp. 54, 58, 63).

C. Kodama, H. Ichikawa, K. Nakayama, T. Kotani, S. Nojima, S. Mimotogi, S. Miyamoto, and A. Takahashi (2013). "Self-Aligned Double and Quadruple Patterning Aware Grid Routing with Hotspots Control". In: *Proceedings of the 18th Asia and South Pacific Design Automation Conference*. ASP-DAC (Yokohama, Japan), pp. 267–272 (cit. on p. 7).

K. Lai, S. Burns, S. Halle, L. Zhuang, M. Colburn, S. Allen, C. Babcock, Z. Baum, M. Burkhardt, V. Dai, et al. (2008). "32 nm Logic Patterning Options with Immersion Lithography". In: *Optical Microlithography XXI* (San Jose, California, USA). Vol. 6924 (cit. on p. 7).

M. Lefebvre, D. Marple, and C. Sechen (1997). "The Future of Custom Cell Generation in Physical Synthesis". In: *Proceedings of the 34th Design Automation Conference*. DAC (Anaheim, California, USA), pp. 446–451 (cit. on p. 3).

L. Liebmann, A. Chu, and P. Gutwin (2015). "The Daunting Complexity of Scaling to 7nm without EUV: Pushing DTCO to the Extreme". In: *Design-Process-Technology Co-optimization for Manufacturability IX* (San Jose, California, USA). Vol. 9427 (cit. on p. 7).

B. J. Lin (2009). "Making Double Patterning Cost Single". In: *Journal of Micro / Nanolithography, MEMS, and MOEMS* 8.1 (cit. on p. 7).

F. S. Makedon, C. H. Papadimitriou, and I. H. Sudborough (1985). "Topological Bandwidth". In: *SIAM Journal on Algebraic Discrete Methods* 6.3, pp. 418–444 (cit. on p. 89).

R. L. Maziasz and J. P. Hayes (1991). “Exact Width and Height Minimization of CMOS cells”. In: *Proceedings of the 28th Design Automation Conference* (San Francisco, California, USA), pp. 487–493 (cit. on p. 4).

M. Mirsaeedi, J. A. Torres, and M. Anis (2011). “Self-Aligned Double-Patterning (SADP) Friendly Detailed Routing”. In: *Design for Manufacturability through Design-Process Integration V* (San Jose, California, USA). Vol. 7974 (cit. on p. 7).

R. Nair, A. Bruss, and J. Reif (1983). *Linear Time Algorithms for Optimal CMOS Layout* (cit. on p. 4).

L. S. Nyland and J. H. Reif (1996). “An Algebraic Technique for Generating Optimal CMOS Circuitry in Linear Time”. In: *Computers and Mathematics with Applications* 31.1, pp. 85–108 (cit. on p. 4).

D. Z. Pan (2009). “What is Double Patterning Lithography and Its Impact on Nanometer Design?” In: *ACM SIGDA Newsletter* 39.10 (cit. on p. 7).

D. Z. Pan, B. Yu, and J.-R. Gao (2013). “Design for Manufacturing with Emerging Nanolithography”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.10, pp. 1453–1472 (cit. on p. 7).

T. Polzin (2003). “Algorithms for the Steiner Problem in Networks”. PhD thesis. MPII Saarbrücken, pp. 1–126 (cit. on p. 20).

M. A. Riepe and K. A. Sakallah (2003). “Transistor Placement for Noncomplementary Digital VLSI Cell Synthesis”. In: *Transactions on Design Automation of Electronic Systems* 8.1, pp. 81–107 (cit. on p. 4).

N. Ryzhenko and S. Burns (2012). “Standard Cell Routing Via Boolean Satisfiability”. In: *Proceedings of the 49th Design Automation Conference*. DAC (San Francisco, California, USA), pp. 603–612 (cit. on p. 7).

C. Sarma, A. Gabor, S. Halle, H. Haffner, K. Herold, L. Tsou, H. Wang, and H. Zhuang (2008). “Double Exposure Double Etch for Dense SRAM: A Designer’s Dream”. In: *Optical Microlithography XXI* (San Jose, California, USA). Vol. 6924 (cit. on p. 7).

J. Schneider (2014). “Transistor-Level Layout of Integrated Circuits”. PhD Thesis. Research Institute for Discrete Mathematics, University of Bonn (cit. on pp. 3, 8).

S. Thomä (2017). “Algorithmen zum Platzieren ineinander gefalteter Transistoren”. Bachelor Thesis. Research Institute for Discrete Mathematics, University of Bonn (cit. on p. 75).

T. Uehara and W. M. van Cleemput (1979). “Optimal Layout of CMOS Functional Arrays”. In: *Proceedings of the 16th Design Automation Conference*. DAC (San Diego, California, USA), pp. 287–289 (cit. on p. 3).

— (1981). “Optimal Layout of CMOS Functional Arrays”. In: *Transactions on Computers* 30.5, pp. 305–312 (cit. on pp. 4, 5, 42).

R. Vicari (2018). “Simplex Based Graphs Yield Large Integrality Gaps for the Bidirected Cut Relaxation”. Master Thesis. Research Institute for Discrete Mathematics, University of Bonn (cit. on p. 20).

T. Weyd (2011). "Leaf Cell Layout". Bachelor Thesis. Research Institute for Discrete Mathematics, University of Bonn (cit. on p. 42).

R. T. Wong (1984). "A Dual Ascent Approach for Steiner Tree Problems on a Directed Graph". In: *Mathematical Programming* 28.3, pp. 271–287 (cit. on p. 20).

P.-H. Wu, M. P.-H. Lin, T.-C. Chen, T.-Y. Ho, Y.-C. Chen, S.-R. Siao, and S.-H. Lin (2013). "1-D Cell Generation with Printability Enhancement". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.3, pp. 419–432 (cit. on pp. 5, 6).

X. Xu, B. Cline, G. Yeric, B. Yu, and D. Z. Pan (2015). "Self-Aligned Double Patterning Aware Pin Access and Standard Cell Layout Co-Optimization". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.5, pp. 699–712 (cit. on p. 8).

A. M. Ziesemer and R. A. da Luz Reis (2014). "Simultaneous Two-Dimensional Cell Layout Compaction Using Milp with Astran". In: *IEEE Computer Society Annual Symposium on VLSI* (Tampa, Florida, USA), pp. 350–355 (cit. on p. 3).

A. Ziesemer and C. Lazzar (2007). "Transistor Level Automatic Layout Generator for Non-Complementary CMOS Cells". In: *IFIP International Conference on Very Large Scale Integration*, pp. 116–121 (cit. on p. 5).

A. Ziesemer, R. Reis, M. T. Moreira, M. E. Arendt, and N. L. Calazans (2014). "Automatic Layout Synthesis with ASTRAN Applied to Asynchronous Cells". In: *5th Latin American Symposium on Circuits and Systems (LASCAS)*, pp. 1–4 (cit. on p. 6).