Flow-based Partitioning and Fast Global Placement in Chip Design

Dissertation
zur Erlangung des Doktorgrades (Dr. rer. nat.)
der Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Markus Struzyna

aus

Tychy/Polen

Bonn, Juli 2010

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

Erstgutachter:	Prof. Dr. Jens Vygen
Zweitgutachter:	Prof. Dr. Dr. h.c. Bernhard Korte

Tag der Promotion: 30. September 2010
Erscheinungsjahr: 2010

Contents

Chapter 1

Introduction

One of the most challenging and fascinating applications of discrete mathematics is chip design. The development of modern processors and application specific integrated circuits *(ASICs)* is unimaginable without the use of advanced mathematics. Here, various hard problems have to be addressed on huge instances.

Modern computer chips consist of millions of modules (*circuits*) representing logical functions and memory elements, which have to be connected by *wires* in the later design process. A key task in chip design is *placement*, in which positions of the modules have to be determined in a given chip area. The placement has a direct impact on the total length of wires needed to interconnect them, and consequently on the performance of the chip — the signal transition times and the power consumption.

In the past, meeting timing constraints was not difficult and wiring was almost always achievable. But the technological advancement, competition and the growing complexity of the chips now make the design process a compound, interlaced, and time-consuming procedure. In this process, many interdependent optimization goals and tight constraints must be addressed. For these reasons, the requirements for placement have been subject to many modifications, as placement tools became an essential ingredient in integrated design flows at many stages and scenarios. Placement faces different object types and it is desirable to translate particular timing and wiring constraints into placement. At the same time, the increasing and often iterative use of placement makes the overall turnaround time highly sensitive to the runtime of the placement.

For all these reasons, powerful, flexible and fast placement algorithms are of particular interest more than ever.

In this thesis we address precisely these aspects of placement. As the placement problem is intractable in theory, we follow the classical approach and divide placement into the *global* phase, which is responsible for the computation of rough cell positions, and the *legalization* part. The major part of this work deals with global placement. We present several novel algorithms and generalized data structures, which are combined to a new version of the global placement part of BonnPlace (Brenner, Struzyna and Vygen [2008]). The new version is able to deal with generalized placement constraints, namely *movebounds*.

This thesis is organized as follows: in Chapter 2 we first introduce a formal description of the placement problem, and discuss briefly the most important optimization goals and classical constraints. After the summary of the major complexity results on the placement problem, we motivate global placement, the main subject of thesis.

Chapter 3 shows an overview of the major approaches to the global placement problem. We also present essential parts of the old version of BonnPlace, which serves as a starting point for our work.

The concept of movebounds is motivated and introduced in the beginning of Chapter 4. We use overlapping movebounds, which can be inclusive or exclusive. Our concept generalizes the specification of Si2 [2009] as both types can be interpreted as soft or hard constraints. We show that using a decomposition of the chip area into homogeneous (w.r.t. movebounds) *regions* can efficiently be used in a top-down partitioning scheme: given n circuits, m movebounds and r regions, we show that in $\mathcal{O}(n + m^2 r)$ time one can check whether a fractional partitioning with movebounds exists. We also prove that an optimal (almost integral) partitioning for some modular cost function can be found in $\mathcal{O}\big(r^2 n(\log(n) + r\log(r)\big)$ time. We discuss interactions between density and movebound constraints, and extend BonnPlace to handle hundreds of movebounds and millions of circuits efficiently.

In Chapter 5, we present a novel core routine, the flow-based partitioning algorithm for global placement. This new scheme is a combination of a global MinCostFlow network model for computing directions and a local, highly scalable, connectivity-aware sequence of partitioning steps for flow realization. Despite its global view, the number of nodes and edges in the new model is only linear in r and does not depend on the number of cells. Even on large instances, the MinCostFlow computation needs seconds. We use the optimal flow to obtain directions for cell movement. The movement is performed by a sequence of local quadratic netlength minimization and partitioning steps. We show that the realization can be parallelized efficiently maintaing repeatability. The new approach resolves several major problems of classical partitioning-based placement: it provides a global view, guarantees a feasible (fractional) partitioning for any initial solution, and reduces the risk of density violations. Our routine can be applied in incremental placements and allows density unaware optimization (timing, wirelength) during global placement. It is also a key step in the extension of the method by Brenner and Rohe [2002], to a combined movement, circuit inflation and density adjustment approach for congestion-driven placement, presented at the end of the chapter.

Chapter 6 deals with generalized data structures in placement. We show how given groups of modules (*clusters*) can efficiently be handled in global placement, using a hierarchical clustering and induced netlists. We extend the clique and star net models to handle clustered nets efficiently. We then discuss several implementation aspects of other components of the new BonnPlace, and pay particular attention to the parallelization of several major routines in Section 6.3.

In Chapter 7 we focus on *finding* groups of modules for placement (*clustering*). After a brief summary of the existing methods, we propose a novel approach which follows the idea of combining global connectivity information from a novel and fast random walk model and a bottom-up clustering scheme. We show how hitting times of random walks from large hypergraphs can be retrieved quickly and in a memory efficient way by a parallel algorithm.

Chapter 8 provides the experiments on a large testbed of recent real-world chips and benchmarks. We compare our tool to the old version of BonnPlace as well as to a modern industrial placer on the real-world chips. Without any algorithmic changes, the new implementation yields a two times faster version of BonnPlace, despite its generalized data structures. Significant improvements can then be obtained using the new flow-based partitioning.
Compared to the old BonnPlace the netlength improves by more than 8%, in several cases by more than 10%, in one case even 28%. Our new tool is more than 5.4 times faster than the previous version of BonnPlace. Results comparable to those produced by the old version of BonnPlace can even be obtained 8.7 times faster.
The new flow-based partitioning is shown to deal much more accurately with the new movebound constraints: comparing to the recursive approach, we obtain more than 10% shorter netlength on average for instances with inclusive movebounds. On instances with exclusive movebounds, the improvement is even 13.7% on average. Compared to the industrial tool we obtain similar results but are more than 5.5 times faster on instances without movebounds. With movebounds, the gap is more than 32% in favor of our approach, and we are to 9-20 times faster than the industrial tool.
The presented global placement tool performs well not only on real-world instances. On the recent benchmarks, we produce highly competitive results in terms of wirelength and obtain the currently best results on the latest benchmark set (Nam et al. [2006]).
Finally, we present the first results of our new clustering, which significantly outperforms *BestChoice* in terms of netlength in a reasonable runtime. Although our implementation is in a preliminary version, we achieve improvements of more than 6.6% in terms of netlength over *BestChoice*. Using the random walk clustering allows us to reduce the number of circuits by a factor of 10 and still obtain slightly shorter netlengths than the placements on unclustered netlists.

This work would not have been possible without the support of many people.

I would like to express my gratitude to my supervisors, Prof. Dr. Bernhard Korte and Prof. Dr. Jens Vygen, for their guidance, and the outstanding working conditions at the Research Institute for Discrete Mathematics at the University of Bonn. The institute under their lead became a distinctive place of research and development and the productive spirit was a huge motivation for my work.

I am very thankful to all members of the placement team: to Dr. Ulrich Brenner, Prof. Dr. Stefan Hougardy, Jan Schneider for the friendly, constructive atmosphere in the team and the many objective discussions over the years. Many thanks go to Levin Keller, Christoph Lauff, Alexander Renelt and Ulrike Suhl for the implementation effort they have spent in the new version of BonnPlace.
I am also thankful to Christoph Bartoschek, Jun.-Prof. Dr. Stephan Held, Christina Schulz, Dr. Christian Szegedy, Dr. Jürgen Werber for the cooperation in timing, to Karsten Muuss for answering many questions on data structures and methodology as well as to Dr. Dirk Müller for sharing his experience in congestion estimation. Special thanks go to Christoph and Dirk for their responses to my questions and the discussions on programming and parallelization.

I am grateful to Ulrich and Jan for backing me during the last months towards the end of completion of this thesis and the repeated proofreading. Stefan Hougardy's, Brian's, Dirk's and Uta's proofreading, remarks and valuable comments have also been a huge help.

My biggest thanks go to my family, my wonderful wife Uta and the lovely boys Patrick and Erik for their love, their permanent support and their infinite patience with me when leaving so early and coming home so late over the last months.

Chapter 2

Preliminaries

In this chapter we introduce the basic notation and conventions. We also provide a description of placement in Very Large Scale Integration (VLSI). We then briefly discuss optimization goals and constraints in placement and focus in particular on density. Finally, a relaxation of the placement problem is motivated and introduced, leading to the key problem considered in this thesis, *global placement*.

2.1 Basic Definitions

Within this work, we denote by $\mathbb{R}$ the set of real numbers, by $\mathbb{N}$ the set of natural numbers including 0 and by $\mathbb{Z}$ the set of integers. For the non-negative subset of reals we write $\mathbb{R}_+$ and $\mathbb{R}_{>0} := \mathbb{R}_+ \setminus \{0\}$.

When we consider the n-dimensional vector space $\mathbb{R}^n$, we interpret its elements as column vectors. For $a \in \mathbb{R}^n$ a^T is then the corresponding row vector. For a real matrix $A = (a_{ij})_{i,j=1}^n$, we write A^T for its transpose $A = (a_{ji})_{ij=1}^n$. For an n-dimensional vector b, let $\operatorname{diag}(b)$ be the diagonal matrix with entries $a_{ii} = b_i$, $i = 1, \ldots, n$. We write $\langle \cdot, \cdot \rangle$ for the standard scalar product $\langle \cdot, \cdot \rangle : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. Sometimes we identify for two vectors $a, b \in \mathbb{R}^n$ the 1×1 matrix $a^T b$ with $\langle a, b \rangle$. For a vector $a = (a_i)_{i=1}^n \in \mathbb{R}^n$, $|a| := ||a||_1 := \sum_{i=1}^n |a_i|$ is the L_1-norm, $||a|| := ||a||_2 := \sqrt{\langle a, a \rangle}$ the Euclidean (L_2-)norm and $||a||_\infty := \max_i |a_i|$ the maximum-norm of a.

For a finite set X let $\mathcal{P}(X)$ be the power set of X, and $|X|$ the cardinality of X.

An *undirected graph* G is a triple $G = (V(G), E(G), \phi_G)$, where $V(G)$ and $E(G)$ are finite sets of *vertices (nodes)* and *arcs (edges)* respectively and $\phi_G : E(G) \to \{\{v, w\} \subset V(G) | v \neq w\}$. A *directed graph (digraph)* is a triple $G = (V(G), E(G), \phi_G)$, with finite sets $V(G), E(G)$ and $\phi_G : E(G) \to (V(G) \times V(G)) \setminus \{(v, v) | v \in V(G)\}$, so we do exclude loops but allow parallel arcs. To abbreviate the notation we will sometimes define ϕ_G implicitly by specifying the arc set only. A graph without parallel arcs is called *simple*.

In an undirected graph G the set $\delta_G(v) := \{e \in E(G) | v \in \phi_G(e)\}$ denotes the set of *incident* edges of some vertex $v \in V(G)$. The cardinality $|\delta_G(v)|$ of this set is called the *degree* of v. In directed graphs we distinguish the sets $\delta_G^-(v) := \{e \in E(G) | \exists w \in V(G) \text{ with } (w, v) = \phi_G(e)\}$ and $\delta_G^+(v) := \{e \in E(G) | \exists w \in V(G) \text{ with } (v, w) = \phi_G(e)\}$, where the first set represents the edges entering some node $v \in V(G)$ and the second set contains edges leaving v. The numbers $|\delta_G^-(v)|, |\delta_G^+(v)|$ are called the *in-degree* and *out-degree* of $v \in V(G)$ respectively.

Let G be a digraph. A *path* P of length k in G is a graph with $V(P) = \{v_0, \dots, v_k\} \subset V(G)$ and $E(P) = \{e_1, \dots, e_k\} \subset E(G)$ such that $v_i \neq v_j$ for $0 \leq i < j \leq k$ and $\phi_G(e_i) = (v_{i-1}, v_i)$ for $i = 1, \dots, k$. A *cycle* C of length k is a graph with $V(C) = \{v_0, \dots, v_k\} \subset E(G)$ and $E(C) = \{e_0, \dots, e_k\}$ such that $v_i \neq v_j$ for $1 \leq i < j \leq k$ and $\phi_G(e_i) = (v_{i-1}, v_i)$ for $i = 1, \dots, k$ and $v_0 = v_k$.

Analogously, a *hypergraph* is a triple $\mathcal{H} = (V(H), E(H), \phi_H)$ where $\phi_H : E(H) \to \{X \subset V(H) : |X| \geq 2\}$.

A *network* is a quadruple (G, u, b, cost), where G is a directed graph, $u : E(G) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a *capacity function*, $\text{cost} : E(G) \to \mathbb{R}$ is a *cost function* and $b : V(G) \to \mathbb{R}$ is a map with $\sum_{v \in V(G)} b(v) = 0$. The b-values state the amount of *supply* if $b(v) > 0$, or *demand* if $b(v) < 0$. Supply nodes are called *sources* and the demand nodes *sinks*. We call a node $v \in V(G)$ with $b(v) = 0$ a *transit* node.

Given a network (G, u, b, cost), we call a function $f : E(G) \to \mathbb{R}_{\geq 0}$ a *flow* if f is dominated by the capacity function, i.e. $f(e) \leq u(e)$ for each $e \in E(G)$ and the *flow conservation rule* holds at any vertex: $b(v) = \sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e$. For a given network, the problem of finding a flow which minimizes the *flow costs* $\sum_{e \in E(G)} \text{cost}(e) f(e)$ is called a MinCostFlow Problem. We allow instances of a MinCostFlow problem to provide less supply than demand. This can be easily adapted to the definition above by extending a network (G, u, b, cost) to $(G', u', b', \text{cost}')$ where $V(G') := V(G) + s$, $E(G') := E(G) \cup \{(s, v) | v \in V(G)\}$, $b'(s) := -\sum_{v \in V(G)} b(v)$ and $b'(v) = b(v)\ \forall v \in V(G)$, $\text{cost}'(e) \equiv 0\ \forall e \in E(G') \setminus E(G)$ and $\text{cost}'(e) = \text{cost}(e)\ \forall e \in E(G)$.

Given an instance (G, u, b, cost) of the MinCostFlow problem and a flow f, the *flow-carrying subgraph* of G is the graph $G_f = (V(G), \{e \in E(G) | f(e) > 0\})$.

When considering geometric shapes, they will all consist of unions of rectangles. We fix the standard orthonormal basis of $\mathbb{R}^2$ and assume all rectangles in the plane to be axis-parallel w.r.t. this basis. For two subsets $A, B \subset \mathbb{R}^2$ we write $A \dot{\cap} B := \overbrace{(A \cap B)}^{\circ}$ for the interior of the intersection. We say A, B do not overlap if $A \dot{\cap} B = \emptyset$.

Finally, we write $\mathbb{P}$ and $\mathbb{NP}$ for the complexity classes of decision problems decidable in polynomial and non-deterministic-polynomial time respectively.

2.2 The Placement Problem

In this section we introduce the main problem of this work, namely the VLSI placement problem. We start with a couple of formal definitions.

Definition 2.2.1 A *netlist* is a quadruple $(\mathcal{C}, \mathcal{N}, P, \gamma)$ for which $\mathcal{C}$ and P are finite disjoint sets. The elements of $\mathcal{C}$ are *cells (placeable objects, circuits, gates)* and the elements of P are *pins (connection points, terminals)*. The map $\gamma : P \to \mathcal{C}\dot{\cup}\{\Box\}$ describes the pin-cell function, for which a pin p with $\gamma(p) = \Box$ is called a *preplaced pin (port)*. $\mathcal{N}$ is a partition of P (a family of disjoint, nonempty subsets whose union is P). The elements of $\mathcal{N}$ are called *nets (connections)*. We may assume $|N| \geq 2$ for each $N \in \mathcal{N}$.

We should mention that γ induces an equivalence relation on $\{p \in P | \gamma(p) \in \mathcal{C}\}$ where $p \cong q :\iff \gamma(p) = \gamma(q)$. Identifying each equivalence class $\{p|\gamma(p) = c\}$ with a cell allows us to interpret a netlist as a hypergraph $(\mathcal{C} \cup \{p \in P|\gamma(p) = \Box\}, \mathcal{N}/\gamma)$. Beyond the abstract connectivity information, we are in fact dealing with real physical objects. To this end we introduce:

Definition 2.2.2 Each cell $c \in \mathcal{C}$ provides two numbers $x_{\text{size}}(c), y_{\text{size}}(c)$ encoding its *rectangular shape* $A(c)$. For a given point in the plane $(x(c), y(c)) \in \mathbb{R}^2$, we write

$$A_{x,y}(c) := [x(c) - x_{\text{size}}(c)/2, x(c) + x_{\text{size}}(c)/2] \times [y(c) - y_{\text{size}}(c)/2, y(c) + y_{\text{size}}(c)/2]$$

for the *embedding* of the shape $A(c)$ at the location $(x(c), y(c))$ into the plane.

Definition 2.2.3 We write $x_{\text{size}}(\Box)$,$y_{\text{size}}(\Box)$ for the width and height of the *chip area*, and set $\mathcal{A} := [0, x_{\text{size}}(\Box)] \times [0, y_{\text{size}}(\Box)]$. In addition, we are given a finite set of *rectangular blockages* $\mathcal{B} = \bigcup_{i=0}^{k} B_i$, where for each blockage $B_i \subset \mathcal{A}$, $i = 0, \ldots, k$. We assume that no pair of blockages overlaps.

These are the major physical shapes involved in placement: the rectangular chip area $\mathcal{A}$, the shapes of each placeable object $c \in \mathcal{C}$, and finally a set of rectangular blockages. The last of these can model *preplaced* objects, non-rectangular chip areas, and forbidden areas provided by real physical objects or representing constraints. We are now able to define the main issue:

Definition 2.2.4 A *placement* is a pair of maps $x, y : \mathcal{C} \to \mathbb{R}_+$. A placement is called *legal* if the following conditions hold:

i. $A_{x,y}(c) \subset \mathcal{A}$ for all $c \in \mathcal{C}$

ii. $A_{x,y}(c)\dot{\cap}A_{x,y}(d) = \emptyset$ for all $c, d \in \mathcal{C},\ c \neq d$

ii. $A_{x,y}(c)\dot{\cap}B = \emptyset$ for all $c \in \mathcal{C},\ B \in \mathcal{B}$.

We sometimes write $(x, y)(c) := (x(c), y(c))$ and $(x, y) : \mathcal{C} \to \mathcal{A}$.

A placement is hence an embedding of the cells in the chip area. A placement is legal if the cells are entirely contained within the chip area and none of them intersects any other cell or blockage more than at the boundary.
We have stated what a legal placement is, but we have not said anything about its quality. To measure the quality of the placement, discussed later, we need the following:

Definition 2.2.5 The maps $x_{\text{offs}}, y_{\text{offs}} : P \to \mathbb{R}$ are called *offsets*, with $x_{\text{offs}}(p) := y_{\text{offs}}(p) = 0$ for any $p \in P$ with $\gamma(p) = \square$. For any $p \in P$ with $\gamma(p) = \square$, we are given in addition a *fix coordinate* $(\tilde{x}(p), \tilde{y}(p)) \in \mathcal{A}$. Given a placement $(x, y) : \mathcal{C} \to \mathcal{A}$, we identify the pin coordinates $(x(p), y(p))$ with:

$$(x(p), y(p)) := \begin{cases} (\tilde{x}(p), \tilde{y}(p)) & \text{if } \gamma(p) = \square, \\ (x(\gamma(p)) + x_{\text{offs}}(p), y(\gamma(p)) + y_{\text{offs}}(p)) & \text{else.} \end{cases}$$

Pin offsets describe the relative position of a pin p w.r.t. the cell, if $\gamma(p) \in \mathcal{C}$. If the pin does not belong to any placeable object, the offsets are defined to be 0. Then the given fixed position of the pin $(\tilde{x}(p), \tilde{y}(p))$ is taken as its location.

Definition 2.2.6 A *net model* is a function $\mathcal{L}$ evaluating a net for a given embedding of its pins: if we identify pins with points in the plane, the net model is:

$$\mathcal{L} : \{N \subset \mathbb{R}^2 | 2 \leq |N| < \infty\} \to \mathbb{R}_+.$$

The *net weights* are given by a function $\omega : \mathcal{N} \to \mathbb{R}_+$.

We are now able to formalize the:

General Placement Problem

Instance: A netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$, chip area $\mathcal{A}$, blockages $\mathcal{B}$, pin offsets $x_{\text{offs}}, y_{\text{offs}} : P \to \mathbb{R}$, preplaced pin positions $(\tilde{x}, \tilde{y}) : \{p \in P | \gamma(p) = \square\} \to \mathbb{R}^2$, cell sizes $x_{\text{size}}, y_{\text{size}} : \mathcal{C} \to \mathbb{R}_+$ and net weights $\omega : \mathcal{N} \to \mathbb{R}_+$.

Task: Find a legal placement $(x, y) : \mathcal{C} \to \mathcal{A}$ which minimizes $\sum_{N \in \mathcal{N}} \omega(N)\mathcal{L}(N)$.

For a netweight we will sometimes distinguish the x– and the y–part. This is possible because pins will be connected by series of axis parallel line segments. Evaluating the *weighted* netlength means to scale the horizontal coordinates of a net weighted with $\omega(N) = (\omega_x(N), \omega_y(N)) \in \mathbb{R}^2_+$ by $\omega_x(N)$ and the vertical ones by $\omega_y(N)$. Different weights during placement are needed to model different numbers of routing resources, distinct electrical properties of horizontal and vertical wires or for improved modeling of artificial connections when placing large blocks.

We have formulated an abstract optimization goal for the General Placement Problem. Now, we are going to specify which issues stand behind this formulation.

First of all, a placement of cells into the chip area does not make the chip work yet: the pins of a net have to be connected by physical electrical connections, the *wires*, during the *routing* phase. Routing is usually performed by Steiner trees typically using several planes. Parts of a net located on different planes are connected by *vias*. Placement has to make sure that the subsequent routing is not only feasible — an unroutable chip is of course useless — but also that the routing problem does not become too hard for the routing tool, to avoid runtime or quality problems (e.g. *detours, large number of vias*). The routing becomes increasingly harder in areas with high routing density (*congestion*). During routing, the router has to connect a net using some prescribed *wire type* of this net, usually providing the physical widths and minimum distance rules to other wires. For this reason, to lower the routing density, we have to shorten the connection lengths. We refer to Müller [2009] and Alpert, Mehta and Sapatnekar [2009] for a more detailed description of routing.

Beyond routing, modern chips have to satisfy tough *timing* and *power* requirements. Timing subsumes the requirements of electrical signals to appear within certain time slots at certain points to guarantee a proper computation of some logical function. Here, the latest signal arrival times do count. The signals can be slowed down almost everywhere by introducing additional cells (*buffers*). Signal transition time increases with the distance to be traversed: long distances result in longer resistances as well as capacities, and both contribute to the signal transmission time (*delay*). Even in chips which are not problematic in terms of timing, the capacity induced by the wirelength has a significant impact on the *power consumption*. Longer connections need more power as longer capacities have to be charged. The increasing use of chips in mobile devices makes this issue extremely important. A detailed description of timing optimization in VLSI design can be found in Held [2009].

Manufacturing costs cannot be neglected either. Apart from the costs resulting from development and the physical design process itself, the other issues which show up once the design process has terminated are significant. The chips are manufactured on *wafers*, on which more than hundred chips are packed. Smaller chip sizes allow to be packed more chips onto a single wafer and the production costs to be reduced. The other important aspect is the *yield*, i.e. the amount of chips which work without causing errors. The number of vias and the distance between wires play a key role here. Vias are fragile connections and a certain percentage of them often cannot be manufactured properly leading to unconnected parts of wires. If wires are packed too densely, this again increases the probability of some dust particle causing an electrical defect on the wire, making the chip useless (see Müller [2006] and references therein). Both effects come hand-in-hand with increasing wiring density.

All these reasons — routability, timing and manufacturing costs — lead to the objective function during placement, namely minimizing the wirelength. For the sake of modeling priorities among the nets, it is common to provide weights and to optimize the weighted netlength during the placement.

2.2.1 Net Models

Once we have decided to minimize the netlength as objective of placement, the question arises *as to how* the netlength should be measured in the General Placement Problem. Routing will be performed by *rectilinear Steiner trees* in most cases, so their aggregated weighted length comes into consideration as a possible net model. As modern routing tools are indeed able to connect most of the pins by a minimal Steiner tree (see Müller [2009]), it seems to be a good choice. Despite their theoretical complexity (the problem is $\mathbb{NP}$-hard, see Garey, Graham and Johnson [1977]), Steiner trees can be computed quickly for most nets (Warme, Winter and Zachariasen [2000]). However, another major problem occurs in stability. The placement, and in the consequence also the pin embedding in the plane, have a severe impact on the Steiner tree topology. A slightly different embedding of the pins can yield a completely different Steiner tree. It is therefore hard to minimize such an instable function. Instead, it is very common to consider the *bounding box (half perimeter wirelength)* as an optimization goal.

$$BB(N) := \max\{x(p)|p \in N\} - \min\{x(p)|p \in N\} + \max\{y(p)|p \in N\} - \min\{y(p)|p \in N\}. \tag{2.1}$$

The reasons for this are diverse. The major argument comes from the fact that this model turns out to be a good approximation to the Steiner tree model, even beyond 2- and 3- terminal nets, for which it is exact. The second reason is the stability: the model is continuous, which makes the optimization easier. Finally, computing $BB(N)$ has linear complexity.

This is perhaps the most important and most widely spread objective for VLSI placement, also considered in the international benchmarks Nam et al. [2005], Nam et al. [2006]. The General Placement Problem with $BB(N)$ as net model is often called the Standard Placement Problem.

If one chooses the net model topology *before* the placement, and asks for the best possible representation of the Steiner tree by a net model, Brenner and Vygen [2001] show that *Clique:*

$$CL(N) := \frac{1}{|N|-1} \sum_{p,q \in N} \big(|x(p) - x(q)| + |y(p) - y(q)|\big) \tag{2.2}$$

is the model of choice, which can be computed in $\mathcal{O}(|N|\log(|N|))$ time by sorting.

When timing comes into play, the dependence between signal delay and netlength is no longer a linear one. The popular simplified *RC-delay* (see Elmore [1948], Rao [1995]) imposes a mixed quadratic-linear dependence for simple line segments. For larger nets, this mixed quadratic-linear delay depends furthermore on the Steiner tree topology and the problem of computing delay optimal placements becomes non-convex. For this reason it is hardly solvable in reasonable time, except for special cases, as examined by Bock [2010]. Nevertheless, we should keep in mind non-linear net models. We return to them later in Chapter 3.

2.2.2 Placement and Density

Netlength, however, cannot be considered as the only objective for placement beyond legality. Placement usually cannot pack cells with arbitrary density for several reasons. Paradoxically, several issues motivating the shortest possible wirelength mentioned above prevent us from packing the placeable objects too closely. During placement, we do not have a precise idea about the realization of the net with a wire by later routing. Too many pins in an area may lead to unroutable instances. Timing, considered indirectly during placement, may require additional buffers or another cell implementation providing different electrical properties (*gate sizing*) to strengthen the electrical signals necessary to traverse larger distances. In regions with dense routing, *coupling* may slow down signal transition times further.

In addition, the fact that placement is used in different design stages requires the placement tool to reserve some free space for later cell insertions, either for the purpose of timing optimization on the present netlist as mentioned, or for later netlist manipulations due to methodology in the design process like *clock tree construction*, (for a detailed description see Maßberg [2009]) or netlist manipulations by logic changes for timing purpose (see e.g. Ossenberg-Engels [2010], Xiang et al. [2010]) as well as late netlist changes and engineering changing orders, *(ECOs)*.

Density constraints impose conditions limiting the area occupied by cells in some area. Given a placement, a point on the chip is occupied by a discrete number of cells, and even a legal placement has a local occupancy of 0% or 100% (of course except sets of measure zero the boundaries where up four cells can meet). It is hence natural not to focus on particular points but to consider density and occupancy in terms of measure theory as "almost everywhere" conditions. Density constraints are thus imposed by some measurable function with values between 0 and 100% on the chip area.

Now the question arises as to how to detect if a placement is respecting the density constraints. Here, several canonical requirements immediately show up: the density violation should be a translation- and rotation- invariant measure. Rotating the chip or shifting the placement and the density constraints by a constant should yield the same values. Another natural postulate is monotonicity: if for a given placement the density constraints are tightened, the resulting density measure should increase. The same should happen if more cells are added to a design, while density constraints are kept the same. Furthermore, the measure should apply to non-legal placements too. In particular, we would like to measure, for a given placement, how far it is from being legal, i.e. what is the legalization effort for this placement. Let us formalize the basic issues first.

Definition 2.2.7 For a given placement $x', y' : \mathcal{C} \to \mathbb{R}$ we define the area consumption using a characteristic function for each cell:

$$\chi_c(x,y) := \begin{cases} 1 \text{ iff } (x,y) \in A_{(x',y')}(c) \\ 0 \text{ else.} \end{cases} \tag{2.3}$$

The *occupancy* function is then

$$u(x,y) := \sum_{c\in\mathcal{C}} \chi_c(x,y). \tag{2.4}$$

So, the occupancy is a step function with integral values which provides the number of cells covering a point in the plane. Density constraints should control the local occupancy for a point in the plane. Moreover, it is reasonable to require that the overall occupancy can met density requirements within some bounded area. Let us assume that this can be done within the compact rectangle $\mathcal{A} \subset \mathbb{R}^2$.

Definition 2.2.8 A *density constraint* function is a measurable map $\mathcal{D} : \mathbb{R}^2 \to [0,1]$. For a given set of cells $\mathcal{C}$ with some placement in the chip area $\mathcal{A}$ and the induced occupancy function u we require:

$$\int_{\mathcal{A}} \mathcal{D}(x,y)dxdy \geq \int_{\mathcal{A}} u(x,y)dxdy \tag{2.5}$$

In other words, beyond an elementary measurability, we require that no more than one cell occupies a point in the plane and that we are able to place the cells in $\mathcal{A}$ without causing density violations. We may assume $\mathcal{D}(x,y) = 0$ for $(x,y) \notin \mathcal{A}$ and for our purpose it is also sufficient to assume that $\mathcal{D}$ is an elementary step function.
One option to measure density violation satisfying which satisfies the monotonicity, translation and rotation invariance would be to treat cells as arbitrary granular particles, and to measure the *movement* which is required to satisfy the density conditions.

This is a classical continuous mass transportation problem, of which the earliest formulations date from the late 18th century by Monge [1781]. A slightly different form was also considered by Kantorovich [1960] and since then by many others. For details and a historical overview see Villani [2009] and the references therein.
For balanced cases, where $\int_{\mathcal{A}} u = \int_{\mathcal{A}} \mathcal{D}$, the formulation by Gangbo and McCann [1995] asks for the of optimal transport, i.e. for a map $\mathcal{T} : \mathbb{R}^2 \to \mathbb{R}^2$ which minimizes the functional:

$$\varpi_{u,\mathcal{D}}(\mathcal{S}) := \int_{\mathcal{A}} ||(x,y) - \mathcal{S}(x,y)||_2 u(x,y)dxdy, \tag{2.6}$$

over all maps with the mass-preserving property, i.e. for all measurable set $X \subset \mathcal{A}$: $\int_X \mathcal{D} = \int_{\mathcal{S}^{-1}(X)} u$. Gangbo and McCann [1995] show the existence and uniqueness of $\mathcal{T}$ for the balanced case, even for the more general, strictly convex cost functions. The density violation can then be defined as the costs of optimal transport $\varpi_{u,\mathcal{D}} := \inf_{\mathcal{S}} \varpi_{u,\mathcal{D}}(\mathcal{S})$. To treat the unbalanced case, one can introduce *dummy cells* of total volume $\int_{\mathcal{A}} \mathcal{D} - \int_{\mathcal{A}} u$, for which the movement costs are constantly zero to any point in the plane. However, one has to accept the loss of uniqueness of the transport map.
It remains an open question, how exact values of $\varpi_{u,\mathcal{D}}$ can be computed, even for the balanced case, but at least an approximation scheme using a sequence of discrete transportation problems can be used to obtain bounds for $\varpi_{u,\mathcal{D}}$, and applies to both, the balanced and unbalanced instances.

In practice, however, it is often sufficient to restrict the density evaluation to average values on much coarser grids, as we will see in Chapter 3.

Finally to unify the notation, analogously to (2.3) we set for a blockage $B \in \mathcal{B}$:

$$\chi_B(x,y) := \begin{cases} 1 \text{ iff } (x,y) \in B \\ 0 \text{ else,} \end{cases} \tag{2.7}$$

and $\chi_{\mathcal{B}}(x,y) := \sum_{B\in\mathcal{B}} \chi_B(x,y)$.

2.2.3 Other Constraints

The General Placement Problem simplifies the placement problem in several ways. We have identified pin locations with points in the plane, but in reality, pins have shapes. The pin areas are usually small enough to be modeled as points using the center of the bounding box of its shapes without causing significant errors.
Second, in practice the cell locations have to obey grid alignments and cannot be chosen arbitrarily. That means we are given a set of grids (*cell tables*) and an assignment for each cell to some grid. The grids appear in two different flavors. Most placeable objects, *the standard cells*, are indeed rectangular and even share a common y_{size}. The grid they have to snap into is very fine in comparison to the chip area and even to most of the cell widths. This grid comes from the power supply structure of the cells, yielding *cell rows.*
Usually the placement does not only compute positions, but also *orientations* of the cells.

Definition 2.2.9 An *orientation* is a map

$$\varrho : \mathcal{C} \rightarrow \{\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}, \begin{pmatrix}1 & 0\\0 & -1\end{pmatrix}, \begin{pmatrix}-1 & 0\\0 & 1\end{pmatrix}, \begin{pmatrix}-1 & 0\\0 & -1\end{pmatrix}, \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}, \begin{pmatrix}0 & 1\\-1 & 0\end{pmatrix}, \begin{pmatrix}0 & -1\\1 & 0\end{pmatrix}, \begin{pmatrix}0 & -1\\-1 & 0\end{pmatrix}\} \tag{2.8}$$

For a cell c with assigned orientation $\varrho(c)$, the pin offsets $(x_{\text{offs}}(p), y_{\text{offs}}(p))$ transform to $\varrho(c)(x_{\text{offs}}(p), y_{\text{offs}}(p))$ for each $p \in P$ with $\gamma(p) = c$.

For a given technology, usually either the first four orientations are allowed if cell rows are parallel to the x-axis, or the last four for cell columns. In addition, the orientations within a row are then restricted to be either the first and the third, or the second and the fourth to support the power structure. Depending on the technology, the roles of x and y can swap. This does not impose any restrictions, as any placement can be then done on a 90° rotated instance and the result can be rotated back again.
Beyond the standard cells, whose number may be in the millions, there are usually no more than a few thousand, larger cells— the *macros*— which might be assigned to coarser grids. Macros do sometimes come with non-rectangular shapes and some of them are allowed to

be mirrored on the $x-$ or y-axis or both. Macro placement rules may also involve binary relations between macros, forcing them to be placed with certain maximal or minimal distances. A recent topic consists of the placement of cell groups with very tight relative maximal distance bounds, which basically allow one relative ordering within the group with fixed relative offsets. These cell groups, the *hard clusters*, are also interpreted as macros. However, the issues involved in finding a legal macro placement do not play a key role in this thesis, but one should keep them at the back of one's mind.

Finally, another type of constraints, called *movebounds*, may force cells to be placed in certain areas of the chip. We refer to Chapter 4 for a detailed discussion.

2.3 Complexity Issues in Placement

From the computational complexity point of view, placement contains several hard subproblems. It is easy to see that deciding if an instance of the General Placement Problem exists is hard: it contains the well known Partition problem (Karp [1972]). Even if one allows several relaxations in which, e.g. the chip area is scaled to be α times larger than the sum of cell areas, the problem is $\mathbb{NP}$-complete (see Schneider [2009]).

In practice, finding a legal placement for standard cells is usually easy, even greedy methods would work in most cases. The widths and the heights of most objects are tiny in comparison to the placement area and packing does not state a real matter here. But even if one restricts the General Placement Problem instance to consist of cells being squares with unit height and width and finding a legal placement really becomes a trivial problem, the optimization goal makes the problem $\mathbb{NP}$-hard.

Already the very special case, in which $\mathcal{A}$ is a single row and each net has two pins, contains the classical $\mathbb{NP}$-hard Linear Arrangement Problem (see Garey and Johnson [1979]). Queyranne [1986] proved that no constant factor approximation algorithm for the Quadratic Assignment Problem exists, unless $\mathbb{P} = \mathbb{NP}$.

There is hardly any good news concerning approximation algorithms for placement, even in special cases. Again, restricting the placement to unit squares and nets consisting of two pins admits an $\mathcal{O}(\sqrt{\log(|C|)})$-approximation algorithm, as claimed by Charikar, Makarychev and Makarychev [2007]. Grid alignment constraints also suffice for $\mathbb{NP}$-completeness, even if an optimal placement is known (see Schneider [2009]). For a given placement, the choice of optimal orientation states another $\mathbb{NP}$-hard task, as proved by Cheng, Yao and Hu [1991].

All statements on complexity above hold for netlists with $|N| = 2$ for each net N. In practice, larger nets are frequent. Additionally in these cases the $\mathbb{NP}$-hardness of the Steiner-Tree problem comes into play. Vygen [2008] showed that it is indeed hard to place a single cell optimally when it has three pins. Beyond simple netlength evaluation, routability constraints often do impose another class of hard problems, as a limited number of space is available for interconnections. This involves e.g. Balanced-MinCut (see Garey and Johnson [1979]) and Edge-Disjoint-Paths (see e.g. Korte and Vygen [2008]).

2.4 Global Placement

In the previous section we have seen that placement contains multiple $\mathbb{NP}$-hard problems, even under several simplifications. This fact limits the hope for an efficient algorithm properly addressing netlength minimization, grid constraints and packing at the same time. In addition, in modern chips the number of nets, cells and pins can be in the *millions* which tightens the efficiency issue once more.
Placement is used in several stages of the design process, often iteratively during timing optimization (see e.g. Held [2009] for an integrated timing-driven placement approach). A legal placement is often not necessary for early optimization, good estimates suffice at this stage. Legal placement is mandatory for routing, concluding the design automation process. Since the chip data is frequently updated and changed, several design cycles are often needed. The placement runtime has a large impact on the overall *turn-around-time*. Fast placement algorithms are thus indispensable.
For all these reasons, it is common to relax the placement problem. For standard cells, the most popular strategy is to distinguish the *global placement* and *legalization*. Other issues are the *macro (large block) placement*, in which large blocks are placed and *Local (Detailed) Placement*, in which manipulations of legal placements are performed.

During global placement, one drops the hard disjointness constraint, but tries to reduce overlaps among cells and between cells and blockages while focusing on netlength minimization. The cell size is tiny compared to the placeable area, so packing issues do not arise in principle and are left for the legalization and macro placement. Legalization is then responsible for addressing grid constraints, correct orientation, and disjointness. It is difficult to specify a proper output of global placement since the meaning varies and is often adapted to the needs of the particular legalization tool. Indisputable is the requirement that global placement should reduce overlaps and density violation. The following legalization step should be possible by applying local operations only. Although an example of how this could be measured in theory is given in Section 2.2.2, in practical approaches the notion of density and its violation varies too. Some placement tools combine netlength and rough measures of density violation into one objective, others focus on netlength minimization while trying to maintain density bounds by construction. These reasons make different global placements sometimes hard to compare and leave room for interpretation if the netlength of a placement is shorter but the density is more violated or vice versa. An easy way to incorporate netlength and density violation was proposed by Nam et al. [2006] during a placement contest for *legal* placements. We will return to this issue when we discuss different placement approaches in Chapter 3.

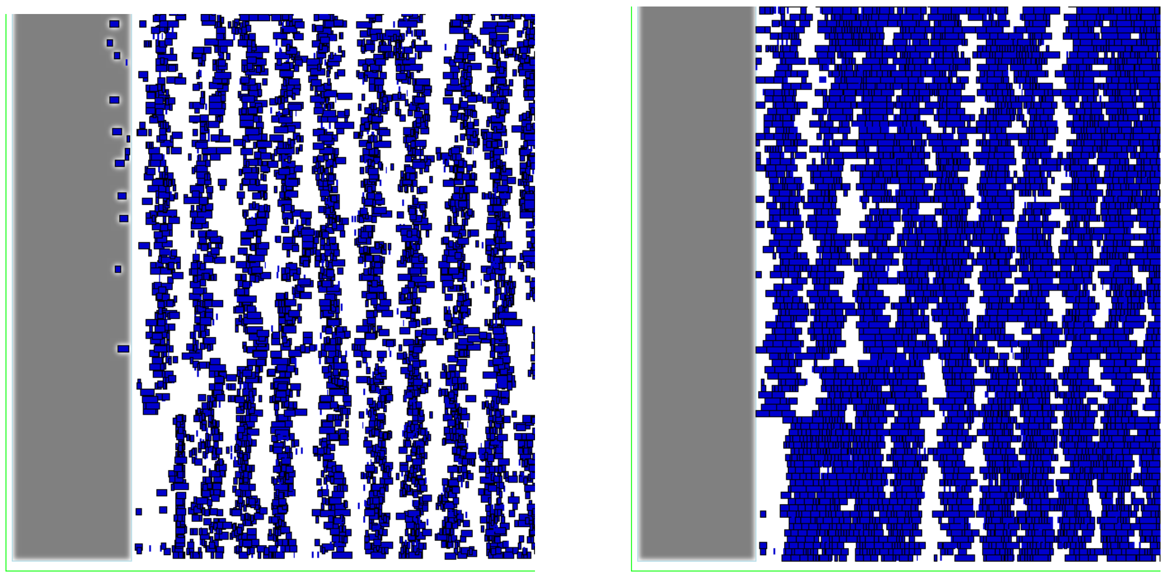


Figure 2.1: Fragment of a chip: global placement outcome (left) and legalized placement (right)

Chapter 3

Global Placement Algorithms

In the previous chapter we described the VLSI placement problem and motivated its relaxation, global placement. There are many different approaches for global placement. In this chapter we present and summarize the properties of global placement algorithms. For a more detailed description of several techniques in global placement we refer to Chang, Jiang and Chen [2009]. We start with older approaches, then focus on analytical placement with its netlength minimization and overlap reduction techniques and finally we commit ourselves to the details of a state-of-the-art analytic and partitioning based placement tool, BonnPlace.

3.1 Non-analytic Approaches

Several early approaches to the placement problem were modeled in the context of Simulated Annealing, in which an arbitrary objective function can be optimized, given an oracle which provides some solution. During optimization, one accepts improvements and a some inferior intermediate solutions. In order to guarantee convergence, the probability that some inferior solution is accepted is becoming smaller and smaller and the overall algorithm converges against a local optimum. An example of such an algorithm is TimberWolf (Sechen and Sangiovanni-Vincentelli [1986]). For others we refer to the book of Wong, Leong and Liu [1988] and the references therein. The Simulated Annealing placers, although theoretically able to find an optimal placement under some assumptions, are usually too slow to obtain good placements. Except for some applications in floor-planning, this technique can be considered obsolete for large scale designs.

Another important class of VLSI placement algorithms consists of so-called *MinCut* placers. In this approach, used for example in placement tools like CAPO (Roy et al. [2005]) and the early version of NTUPlace (Chen et al. [2005]), the netlist is considered as a hypergraph and partitioned together with the chip area. The resulting parts are then assigned to parts of the chip and this process is repeated recursively, until the windows are small enough. Despite the fact that the balanced MinCut problem is another $\mathbb{NP}$-hard task (Garey and

Johnson [1979]) and that the fast heuristics proposed by Fiduccia and Mattheyses [1982] and Ababei et al. [2002] do not provide any quality guarantee, the placements produced by these tools are often reasonable. Unfortunately, another significant problem occurs. The MinCut placers are unstable: slightly different input often leads to completely different placements. This makes the industrial use of them difficult in an iterative process. After all, the MinCut techniques play an important role in routability and congestion driven placement as well as in grouping cells, the *clustering*, even if they do not seem to be suitable as core placement routines.
Beyond classical MinCut placers, we mention Dragon (Wang, Yang and Sarrafzadeh [2000]), FengShui (Khatkhate, et al. [2004] as well as Agnihotri, Ono and Madden [2005]) are hybrid approaches combining MinCut and Simulated Annealing algorithms.
MinCut approaches do not address netlength minimization directly, which can lead to inferior placements when focusing on cuts only. Simulated Annealing accepts any kind of objective, in particular netlength, but cannot be applied with reasonable results and runtime, except for tiny instances. In order to optimize netlength on large scale, another strategy is needed. This motivates another large class of placement tools, namely the *analytic* ones.

3.2 Analytic Netlength Minimization

Netlength minimization as an objective has been discussed in Section 2.2. Then, in Section 2.2.1, we introduced the linear net models which are considered as optimization goals for placement. Linear net models, although reflecting most of the later routing distances, lead to various problems in placement. The first reason is the optimization issue as piecewise linear functions are not differentiable. Despite the fact that placement minimizing linear bounding-box netlength can be considered as the dual of a MinCostFlow problem and computed exactly, it is difficult to compute global optima in acceptable time if the number of variables is in the millions.
The second reason comes from the fact that placements optimizing linear netlength (without any disjointness constraints) are not unique in general. Those computed by such a MinCostFlow approach tend to be crowded in some particular corner solution and do show significant overlaps. It is then not evident how to resolve them without introducing too much arbitrariness. Such an effect does not show up for strictly convex net models.
There are different methods for coping with the drawbacks above, but all use a non-linear approach to approximate the linear netlength and are unified in the analytic placement concept.

3.2.1 Non-linear Non-quadratic Net Models

Due to the difficulty of direct minimization of linear objective functions, several authors propose to approximate the linear bounding box model with smooth functions. The placement tools mPL6 (Chan et al. [2006], Cong and Luo [2008]), Vaastu (Agnihotri and Madden [2007]), NTUPlace3 (Chen et al. [2008]) as well as APlace (Kahng and Wang [2006]) use the so-called log-sum-exp model:

$$LSE_x(N) := \alpha \log\big(\sum_{p\in N} \exp(x(p)/\alpha)\big) + \alpha \log\big(\sum_{p\in N} \exp(-x(p)/\alpha)\big) \tag{3.1}$$

and $LSE(N) := LSE_x(N) + LSE_y(N)$. It is easy to see that $LSE(N)$ converges against $BB(N)$ when α tends to 0.
Other approaches propose to approximate the bounding box model with L_p norms:

$$LP_x(N) := \sum_{p,q\in N} \big((x(p) - x(q))^p + \alpha\big)^{1/p}, \tag{3.2}$$

and $LP(N) := LP_x(N) + LP_y(P)$. In the context of VLSI placement, this was first proposed by Alpert et al. [1997] for $p = 2$. Again, $LP(N) \overset{1/\alpha, p\to\infty}{\to} BB(N)$. When comparing (3.1) to the LP model, the first leads to better results, as reported by Kahng and Wang [2006].
Li and Koh [2007] propose to approximate the bounding box model by recursively applying the $CHKS$ function for approximation of the two-variable max function:

$$CHKS(z_1, z_2) = \frac{\sqrt{(z_1 - z_2)^2 + \alpha^2} + z_1 + z_2}{2} \tag{3.3}$$

with a smoothing parameter α. Now, for a vector $z = (z_1, \dots, z_n)^T \in \mathbb{R}^n$ define recursively $f_{i,i}(z) := z_i$ for $i = 1, \dots, n$, $f_{i,i+1}(z) := CHKS(z_i, z_{i+1})$, $i = 1, \dots, n$ and finally $f_{i,j}(z) := CHKS(f_{i,k}(z), f_{k+i,j}(z))$ for $1 \leq i < j \leq n$ with $1 < j - i$ and $k = \lfloor \frac{i+j}{2} \rfloor$.
This leads to an approximation of $\max(z)$ with $f_{1,n}(z)$ and, using the fact that $\max(-z) = -\min(z)$, $BB(N)$ can be computed this way. The comparison of the $CHKS$ model to (3.1) shows comparable results in both wirelength and runtime, as reported by Li and Koh [2007].
Non-linear and non-quadratic net models are in general more difficult to compute than quadratic ones. Iterative methods for computation are usually too slow to be performed on the entire netlist. To cope with this issue, almost all tools using the LSE model make use of the *multilevel approach*, in which cells are grouped and a cell group is considered as one placeable object. One proceeds from coarse to fine cell groups and from coarse to fine grids.

3.2.2 Quadratic Net Models

Quadratic placement is probably one of the oldest approaches, already considered by Hall [1970], even before Intel built its first 4004 microprocessor and design automation was not an issue yet. One of the first quadratic placement algorithms in VLSI design was presented by Tsay, Kuh and Hsu [1988], see also Brenner and Vygen [2009] for more historical references.
There are different quadratic net models. We start with the quadratic clique and the equivalent star model:

Definition 3.2.1

$$CL_2(N) := \frac{1}{|N| - 1} \sum_{p,q\in N} \big((x(p) - x(q))^2 + (y(p) - y(q))^2\big), \tag{3.4}$$

$$ST_2(N) := \inf_{(x',y')\in\mathbb{R}^2} \sum_{p\in N} \big((x(p)-x')^2 + (y(p)-y')^2\big). \tag{3.5}$$

It should be noted that for the quadratic star the infimum is indeed attained with (x', y') satisfying $|N|x' = \sum_{p\in N} x(p)$ and $|N|y' = \sum_{p\in N} y(p)$. It is well known that $CL_2(N) = \frac{2|N|}{|N|-1} ST_2(N)$, so in particular both can be computed in linear time. Both net models are used by several placement tools, e.g. BonnPlace (Vygen [1997] and Brenner, Struzyna and Vygen [2008]), FastPlace (Viswanathan and Chu [2005], Viswanathan, Pan and Chu [2007]), hATP (Nam et al. [2006]), RQL (Viswanathan et al. [2007]), Q-Place (Lua et al. [2007]), DPlace (Luo and Pan [2008]), FDP (Vorwerk, Kennings and Vannelli [2004]), mFAR (Hu, Zeng and Marek-Sadowska [2005]) and UPlace (Yao et al. [2005]), using partially different weighting heuristics. BonnPlace, for example, discounts larger nets with more than 4 terminals with an additional factor of $2/\sqrt{|N|}$ as suggested by Brenner [2000] due to empirical results.

For an improved approximation of the linear net model by the quadratic one, Gordian-L (Sigl, Doll and Johannes [1991]) use the following iterative net model: in iteration $k = 2, \ldots$ one optimizes essentially:

$$GordianL_x^{(k)}(N) := \frac{1}{|N|-1} \sum_{p,q\in N} \frac{\big(x_k(p) - x_k(q)\big)^2}{|x_{k-1}(p) - x_{k-1}(q)|} \tag{3.6}$$

where the initial distances come from the quadratic minimum and $GordianL^{(k)}(N) := GordianL_x^{(k)}(N) + GordianL_y^{(k)}(N)$. Indeed, it can be proved that $\lim_{k\to\infty} GordianL^{(k)}(N) = CL(N)$ (see Struzyna [2004]). Recall that linear clique was the model of choice for a-priori topologies, so in the limit one can obtain the best achievable result. But after all, one cannot expect more than linear convergence of this method, so in general this approximation method will be too slow.
Another recent proposal was made by Spindler, Schlichtmann and Johannes [2008] in Kraftwerk2: The authors suggest to compute a quadratically optimal placement, sort the pins in each net w.r.t. horizontal and vertical coordinates separately. Then, given this relative order, they cancel the pure internal connections. The resulting horizontal part of the net model can then be written as:

$$B2B(N) := \sum_{p,q\in N} \omega_{x,pq}(x(p) - x(q))^2 \tag{3.7}$$

where $\omega_{x,pq}$ is defined as:

$$\omega_{x,pq} := \begin{cases} \frac{2}{(|N|-1)\left(x'(p)-x'(q)\right)} & \text{iff } p \text{ or } q \text{ is not an internal pin (w.r.t. sorting in } x\text{-coord.)} \\ 0 \text{ else,} & \end{cases} \tag{3.8}$$

and $x'(p)$ is the initial location of the pin p directly before minimizing (3.7).
The quadratic net models are popular for several reasons. The simplicity of computation is obvious: minimizing the sum of quadratic netlengths means to find a minimum of the quadratic program (*QP)*:

$$\min_{x,y}\bigl(x^T Ax - x^T b + y^T A'y - y^T b'\bigr) \tag{3.9}$$

for appropriate constant matrices A, A' and vectors b, b'. Without any side constraints one has even $A = A'$. For strictly convex problems this is nothing other than solving the linear systems

$$Ax = b \text{ and } A'y = b'. \tag{3.10}$$

So these systems can be solved independently for the x– and y–coordinate by separability of the quadratic net models.
The matrices A, A' are diagonally dominant, positive semidefinite and symmetric. We focus on the horizontal part. Assume that each net has already been replaced by a clique or star and the weights have been adjusted. Then we have nets with exactly 2 pins only and if we treat star nodes as cells we can write:

$$A = (a_{c,d})_{c,d=1}^n = \begin{cases} \sum_{\{p,q\}\in\mathcal{N}:\gamma(p)=c,\gamma(q)\neq d} \frac{\omega(N)}{|N|-1} & \text{if } c = d, \\ \sum_{\{p,q\}\in\mathcal{N}:\gamma(p)=c,\gamma(q)=d} -\frac{\omega(N)}{|N|-1} & \text{if } c \neq d. \end{cases} \tag{3.11}$$

The dimension of the matrix n is the sum of the number of cells and the number of nets modeled by the star model. As on typical VLSI netlists, the number of nets is in the same order as the number of cells, the matrix is sparse. The right-hand side vector $b = (b_c)_{c=1}^n$ basically reflects the weighted offsets differences between the corresponding pins.
Connections $\{p, q\} \in \mathcal{N}$ to some preplaced pin q (i.e. $\gamma(q) = \square$) contribute to the diagonal entry of $\gamma(p)$ only. If in each connected component there is a connection to a preplaced pin, then the matrix A is positive definite, and there is a unique solution.
Sparsity, symmetry and the fact that A is positive definite make the linear system solvable by fast iterative methods, e.g. the already classical preconditioned conjugate gradient method with Incomplete Cholesky Decomposition as preconditioning technique (see for example the textbook by Stoer and Bulirsch [2002] for details).
A different reason for quadratic netlength minimization is the impact on timing. As initially mentioned, the signal delay is a mixed linear-quadratic function in wirelength. Although the delay is not a pure quadratic function, long nets are subject to later buffering. The buffering step has a linearization effect on its own, and quadratic netlength minimization places the buffer chains optimally – in equidistant steps.
Stability is another relevant issue: slightly modified netlists (e.g. through slightly different weights) lead to similar placements as shown by Vygen [2007]. Such small weight deviations often occur due to the non-deterministic behavior of timing engines, which consequently compute slightly different arrival times and weights. Stability is extremely important for industrial use. During the design process, placement is used several times on a single netlist, and even the netlist is subject to modifications. It is mandatory that local changes have a local impact on the result and do not change the overall placement.

3.3 Density and Overlap Reduction

The minimization of netlength without any side constraints usually leads to highly overlapping placements. There are various strategies that incorporate overlap reduction and compliance with the density targets. We now present the most important ones, but first we start by considering the density measures in *global placement*.
Almost all global placement algorithms use a uniform grid to discretize the capacity and density functions in order to efficiently compute these values. The only exception is the warping approach proposed by Xiu and Rutenbar [2007], who do not address density constraints at all and focus on legal placements only.

Such a coarse approach provides several benefits. On the one hand, for efficiency and numerical purposes it is usually necessary to stop overlap reduction before all overlaps are removed and continue with legalization at this stage. On the other hand, if one needs some space to be left for routability or future cell insertion reasons, it is often enough to provide the space in a certain neighborhood of particular cells. The precise location is not that important.

Thus, to unify the notation we provide the discrete version of capacity, target density and free area w.r.t.some grid Γ in the plane:

Definition 3.3.1 Given $\mathcal{A}$ and a grid $\Gamma = \Gamma_{(\alpha_x,\alpha_y)} := \{(i\alpha_x, j\alpha_y) | i, j \in \mathbb{Z}\}$, $\alpha_x, \alpha_y \in \mathbb{R}_{>0}$, a *window (bin)* w in Γ is a non-empty half-open rectangle of the form

$$w := \mathcal{A} \cap [i\alpha_x, (i+1)\alpha_x) \times [j\alpha_y, (j+1)\alpha_y) \text{ for some } i, j \in \mathbb{Z}.$$

For two integers $s, t \in \mathbb{N}$, an $s \times t$*-window* W in Γ is a rectangle $W = \mathcal{A} \cap [i\alpha_x, (i+s)\alpha_x) \times [j\alpha_y, (j+t)\alpha_y)$ for some $i, j \in \mathbb{Z}$ with $W \subset \mathcal{A}$. For a collection $\{w_1, \ldots, w_k\}$ of windows, a *coarse window covering* $w_1, \cdots, w_k$ is some $s \times t$-window W with $W \supseteq w_i$ for $i = 1, \ldots, k$.
Given two grids Γ and Γ', we say Γ' is a *refinement* of Γ if $\Gamma \subset \Gamma'$.
For a (fine or coarse) window w we denote by $\text{area}(w)$ the product of w's edge lengths. For a set of disjoint blockages $\mathcal{B}$, the *blocked area* in w is the amount of blocked space in w: $\text{blocked\_area}(w) := \sum_{B\in\mathcal{B}} \text{area}(B \cap w)$ and the *free area* in w is $\text{free\_area} := \text{area}(w) - \text{blocked\_area}(w)$. A window w with $\text{free\_area}(w) = 0$ is called *blocked* and *unblocked* otherwise.
The *capacity* in a window is then

$$\text{capa}(w) := \int_w (1 - \chi_{\mathcal{B}}(x, y))\mathcal{D}(x, y)dxdy \tag{3.12}$$

and finally the *target density* in an unblocked window w is defined as:

$$\mathcal{D}(w) := \frac{\text{capa}(w)}{\text{free\_area}(w)}. \tag{3.13}$$

For a blocked window w we set $\mathcal{D}(w) := -\infty$.

The definitions of capa,$\mathcal{D}$ and free_area extend to finite sets of rectangles in $\mathcal{A}$ in a natural way. From now on, we assume each grid to be uniform and for a fixed grid, introducing density constraints means to require that the total amount of cells *in a window* w divided by the free area in a window does not exceed $\mathcal{D}(w)$. We have to stress that the meaning of cells *in* a window varies. While some tools do consider a cell c to be in a window w if $A_{(x,y)}(c) \cap w \neq \emptyset$, others evaluate a point-wise assignment, where a cell can only be assigned to exactly one window. We now discuss the different approaches in detail.

3.3.1 Penalty Methods

When we discretize the density constraints and consider average values, then netlength minimization with density constraints can be modeled as a constrained optimization problem. Density is again evaluated window-wise and if we recall the occupancy definition from (2.4) the problem is then:

$$\min \sum_{N\in\mathcal{N}} \omega(N)\mathcal{L}(N) \tag{3.14}$$

$$\text{s.t.} \int_w u(x',y')dx'dy' \leq \text{capa}(w) \quad \text{for each window } w. \tag{3.15}$$

The condition (3.15) can explicitly be expressed as a function in cell coordinates, as $u(x',y') = u_{(x,y)}(x',y')$ for a placement $x, y : \mathcal{C} \to \mathbb{R}$:

$$\Xi_w(x,y) := \int_w u(x',y')dx'dy' \leq \text{capa}(w) \quad \text{for each window } w, \tag{3.16}$$

but this function is not differentiable. Again, to cope with this fact, smoothing is applied. There are different methods for obtaining a smoothed version $\tilde{\Xi}_w$ of Ξ_w: Chan et al. [2006] propose to solve the differential equation $\Delta\tilde{u}(x',y') - \alpha\tilde{u}(x',y') = -u(x',y')$ with smoothing parameter α, assuming constant $\tilde{u}$ at the boundary. Then the occupancy function u is replaced by $\tilde{u}$ and $\tilde{\Xi}_w(x,y) := \int_w \tilde{u}(x',y')dx'dy'$. Aplace (Kahng and Wang [2006]) and NTUPlace3 (Chen et al. [2008]) use quadratic splines for direct approximation of Ξ_w.
The smoothed $\tilde{\Xi}$ is not a simple quadratic function. It is not astonishing that this method is applied in the context of the non-quadratic *LSE* net model: the common formulation of Aplace, NTUPlace3, and mPL6 is then to iteratively solve the unconstrained problem

$$\min \sum_{N\in\mathcal{N}} LSE(N) + \lambda \sum_w (\tilde{\Xi}_w - \text{capa}(w))^2. \tag{3.17}$$

3.3.2 Artificial Nets and Anchors

Merging density constraints into a quadratic placement framework should not disturb the positive properties of the unconstrained quadratic approach. One possibility to achieve placements with lower density violations is to “pull out” some cells out of crowded areas towards empty ones using artificial nets with artificial pins (anchors). There are different methods for computing anchor pin locations and weights for these artificial nets.

Force-Directed Approaches

The initial idea of force directed placement is due to Eisenmann and Johannes [1998], who proposed interpreting placement in the context of electrostatic particle distribution. The density then corresponds to the electrostatic charge density. The cell usage at some point can be considered as an electric potential which is exposed to forces in an electric field. Assuming the existence of such a conservative, non-constant potential Φ leads to Poisson's equation in which the derivative of the force $f = \nabla\Phi$ is proportional to the difference between usage and capacity:

$$\triangle\Phi(x, y) = \nabla f = -\alpha\big(u(x, y) - \text{capa}(x, y)\big) \tag{3.18}$$

with some constant $\alpha \in \mathbb{R}_+$. The solution of this differential equation cannot only be guaranteed but is indeed unique and turns out to be the function

$$f(x, y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\big(u(x', y') - \text{capa}(x', y')\big)\frac{(x, y)^T - (x', y')^T}{||(x, y) - (x', y')^T||_2}dx'dy'. \tag{3.19}$$

Now, let us return to placement and restrict ourselves to horizontal coordinates. Then, for a given placement x, the task is to find a new placement which moves towards the less dense areas according to f. In terms of quadratic placement, this essentially means starting with a quadratically optimal placement $x^{(0)}$ satisfying $Ax^{(0)} = b$, then computing forces according to (3.19) and manipulating the right hand side to $Ax = b + c$ where the difference Δc of c is proportional to the computed force. The process is iterated until there are no significant density violations.

The initial approach of Eisenmann and Johannes has been subject to several improvements through the years, addressing both the runtime and the placement quality. Modern force-directed placers use many different methods which do not fit into the electrostatic model anymore. Instead of manipulating only the right hand side of the linear system, additional anchors and artificial nets are defined. Given a solution $x^{(0)}$ of $Ax = b$, new anchor locations x' are determined by the current cell locations $x^{(0)}$ and the forces: $x' = x^{(0)} + f$. The weights of the connections are given by ω, which is usually controlled by a parameter and decreasing in the course of the placement. This leads to the linear system:

$$Ax - b + \text{diag}(\omega)(x - x') = 0 \tag{3.20}$$

which is the same as

$$\big(A + \text{diag}(\omega)\big)x = b + \text{diag}(\omega)x' \tag{3.21}$$

and can indeed be seen as adding preplaced pins and nets to the system. The solution minimizes the quadratic netlength and the length of artificial nets between cells and the anchors determined by the forces. This idea was the nucleus of a vast number of different placement tools: FDP (Vorwerk, Kennings and Vannelli [2004]), mFAR (Hu, Zeng and Marek-Sadowska [2005]), FastPlace (Viswanathan and Chu [2005], Viswanathan, Pan and

Chu [2007]), RQL (Viswanathan et al. [2007]) and Kraftwerk2 (Spindler, Schlichtmann and Johannes [2008]) follow this idea. The force computation starts again from the new solution obtained by (3.21) and can be applied iteratively. This corresponds to the discretization of a continuous repulsion process looking for cell distribution equilibria.
Kraftwerk2, instead of an accumulation of repulsion forces as suggested by the initial model, uses *hold forces* preventing a solution of (3.21) from collapsing back to $x^{(0)}$, the force computation as proposed by Spindler, Schlichtmann and Johannes [2008] is not a cumulative process any more.
The runtime issue is addressed in FastPlace: Viswanathan and Chu [2005] accelerate the convergence process by the so-called *cell shifting*, in which cells are iteratively assigned to windows in such a way that the windows' capacity bounds are not violated. This is done in an alternating way, row- and column-wise. After the cell shifting anchor points on the chip boundary and artificial nets are defined to prevent collapsing to the initial solution. This is iterated on finer grids.

The authors of FDP (Vorwerk, Kennings and Vannelli [2004]) and RQL (Viswanathan et al. [2007]) control the forces computed by (3.19) in two different ways. FDP combines the repulsive force with an additional artificial net and an anchor connected to the cell's desired location (in terms of bounding box netlength) in order to minimize linear netlength. Following Vorwerk, Kennings and Vannelli [2004], this strategy helps to cope with one general problem in force directed approaches: the cell order is determined by the initial solution even if relative order changes would serve the minimization of linear wirelength. Such nets make order changes possible and directly help to minimize linear wirelength.
The authors of RQL (Viswanathan et al. [2007]) choose a brute-force heuristic and reset a certain amount of entries in the force vector f which exceed a certain threshold. They observed that the force amplitude is extremely high for a small amount of cells and leads to large movement and inferior results.
By this method, the overlap elimination is performed in a slightly softer manner, where the cells whose repulsive force have been canceled are moved indirectly as they follow their neighbors. The latter strategy helps to cope with another problematic issue in force directed placement, namely the fact that these approaches tend to distribute the cells uniformly, even if a high target density is allowed and several areas of the chip do not have to be used.

Another way is chosen by the authors of Uplace (Yao et al. [2005]), who, instead of evaluating conditions of type (3.15), observe that discrete cosine transformation can be used to express density balance differences in terms of frequencies. After discretization, the quadratic differences between (discrete) frequencies can be evaluated and in this formulation low frequencies correspond to globally unbalanced densities, while the high frequencies correspond to local deviations only. So going for equilibrated density balance means to look for placements yielding high frequencies, which translates to minimizing the sum of quadratic differences between the entries in the (discrete) frequency matrix. Then forces are added which correspond to the linearized approximation of the quadratic frequency differences.

Diffusion

The diffusion approach in VLSI placement was initially proposed by Ren et al. [2005] for the concept of incremental legalization. The idea is based on the diffusion process in which cells are treated as gas particles and tend to move from high density areas to parts of the chip with low density. The diffusion process describes the concentration (density) of particles over time and behaves conformally to Fick's laws with diffusion parameter α,

$$\frac{\partial d_{x,y}(t)}{\partial t} = \alpha \nabla^2 d_{x,y}(t), \tag{3.22}$$

in which the density difference over time at some point in the plane is proportional to the derivative of the slope of d in this point. The diffusion process comes to an end at the chip's boundary, thus $\nabla d_{x,y}(t) = 0$. The solution of this equation can be seen as assigning velocities to the cells which then move with high speed from areas with significant slope differences and slowly in areas with almost uniform density. The authors of DPlace, Luo and Pan [2008], apply this idea to global placement. The iterative velocity computation and direct cell shifting via (3.22) is problematic for efficiency reasons and in particular in the presence of blockages. The authors of DPlace hence use the solution of (3.22) to compute artificial nets and anchors and continue as in force directed methods.

Cell–window Matching

A combination of discrete optimization and artificial nets and anchors is proposed in Vaastu (Agnihotri and Madden [2007]). The authors solve a bipartite matching problem between clustered cell groups and windows. In order to obtain a matching formulation, cells are clustered to similar cluster sizes which on their side correspond to window capacities.
As assignment costs, the approximated wirelength gain is chosen. To this end, for each clustered cell group C and each window w the wirelength based on the present location of C is evaluated first. Then, the movement of C to w's center is simulated (all other cells remain at their position) and the wirelength after this simulated step is evaluated. The difference is used for assignment costs between C and w. A matching is then computed using the SUCCESSIVESHORTESTPATH method, restricting the solution space to flows whose assignment radius is bounded by some constant for efficiency reasons.
Then, the computed MINCOSTFLOW solution is not used for cell assignment directly, but transformed to artificial nets between cells and the destination windows, computed by the MINCOSTFLOW. The entire process is applied iteratively, increasing the attractors' weights and decreasing the clusters' sizes until each cluster contains a small number of cells. To check the possible infeasibility due to radius constraints, MAXFLOW algorithms are used for quick feasibility checks and the search radius is increased wherever necessary.

3.3.3 Partitioning

Partitioning follows in principle a very simple idea: instead of indirectly penalizing density violations or iteratively reducing overlaps by anchors and artificial nets, make sure that

by construction there are not more cells than capacity in a window by a strict assignment of cells to windows. There are different techniques of partitioning and unlike the MinCut approaches, where recursion can be done without any side effects, in analytic placement one has to make sure that after the partitioning the following optimization steps do not collapse to the initial solution.
Partitioning-based analytic placement algorithms interchangeably apply netlength minimization and partitioning, proceeding in a sequence of grids $\mathcal{A} = \Gamma_0, \ldots, \Gamma_k$. In the early partitioning-based quadratic placers such as PROUD (Tsay, Kuh and Hsu [1988]) and GORDIAN (Kleinhans et al. [1991]) iterative bisection was used to recursively partition cells from a window into two subwindows. Beyond MinCut methods for bisection, one can also minimize the deviation costs from the present placement to a solution respecting the partitioning. The latter can easily be computed in linear time using a median search and is motivated by the idea that a placement which minimizes quadratic netlength should be disturbed as little as possible to enforce legality.
Then, in order to prevent the solution from collapsing back to the original one, Tsay, Kuh and Hsu [1988] proposed local projection methods, in which, when computing the quadratically optimal placement in column i, the cells in neighboring windows were temporarily projected on the i-th column's boundary. This guarantees that, while minimizing netlength in column i, the other cells are considered to be fixed, and none of the cells in the i-th column leaves it. Kleinhans et al. [1991] in their work instead proposed to impose linear constraints, restricting the center of gravity of cells in a window to the window's center. Quadratic netlength minimization with linear constraints can efficiently be computed if the number of constraints is small and each cell is considered in at most one such constraint per dimension (i.e. one per horizontal and one per vertical part). Using variable elimination, unconstrained solution methods can be applied. However, both approaches have significant drawbacks – the temporary projection optimizes quadratic netlength locally which is not the real objective. The center of gravity assignments can hardly deal with uneven density targets.
The partitioning approach proposed by Vygen [1997] introduced two major aspects. The first one was a linear-time algorithm, *Quadrisecton*, for simultaneous cell assignment to *four* windows minimizing the L_1 movement. The second was the *terminal propagation*: a net traversing grid coordinates *(cut-lines)* is split up into two nets with artificial pins at cut-lines resulting from reciprocal projection. Assume we have a net $N = \{p, q\}$ and partitioning has decided that

$$x(p) \leq x_\Gamma \leq x'_\Gamma \leq x(q), \tag{3.23}$$

where x_Γ, x'_Γ are some (possibly equal) horizontal grid coordinates (imposing vertical cut-lines). Then, N is replaced by N_p and N_q where N_p connects p to a new artificial pin at x_Γ and N_q connects q to another new artificial pin at x'_Γ. This preserves (3.23) in the following quadratic optimization step. Quadrisection with terminal propagation has been the core routine of BonnPlace for years and was also implemented in hATP Nam et al. [2006].
Brenner [2005] proposed in his dissertation the *Multisection* algorithm, in which an assignment of n cells to a constant number of windows can be computed in $\mathcal{O}(n \log(n))$ time

minimizing arbitrary cost functions. In fact, Multisection does not rely on any geometric structures unlike Quadrisection, so rectangular windows are no longer necessary and the handling of movebounds is possible. Of course, unless $\mathbb{P} \neq \mathbb{NP}$ all these methods, Bi–, Quadri– and Multisection, can only compute an optimal fractional assignment. Given m target windows, the solution can be transformed to an integral one with the same costs for all but $m-1$ cells. We now present the major routines of BonnPlace in detail.

3.4 BonnPlace

In this section we present the major routines of BonnPlace. This placement tool also fits in the concept of partitioning-based quadratic placement algorithms, in which starting from a quadratically optimal placement, the cells are partitioned from coarser to finer grids, $\mathcal{A} = \Gamma_0, \ldots, \Gamma_k$. The initial grid is imposed by the chip area, the vertical step size of the finest grid corresponds to cell row steps (i.e. to the height of standard cells). The horizontal one depends on the widths of standard cells and is chosen in such a way that the horizontal grid step of Γ_k about three times wider than most standard cells.

BonnPlace then proceeds in *levels*: in level $i = 1, \ldots, k$ the cells are assigned to windows induced by Γ_{i-1}. Then `Partitioning`$(\mathcal{C}, \Gamma_{i-1}, \Gamma_i)$ computes an assignment of all cells $\mathcal{C}$ to the next finer grid Γ_i, followed by quadratic netlength minimization `QP`(Γ_i), in which terminal propagation on Γ_i's boundaries for $i = 1, \ldots, k$ is applied.

For the purpose of quadratic netlength minimization, BonnPlace uses a hybrid clique/star net model. Nets are in general modeled by a clique, in particular when they are split by terminal propagation. Fixed pins at grid's boundaries and split nets do not increase the number of entries in the matrix, thus the corresponding off-diagonal entry vanishes and is replaced by incremented diagonal entries. For large net parts within a window, a clique is replaced by an equivalent star. With this idea, not only the horizontal and the vertical linear programs can be solved independently: given a grid Γ_i, $i = 1, \ldots, k$ the quadratic program for the cells in each row and each column of the Γ_i can be computed independently from any other row and column, in particular parallel algorithms apply. For details see Struzyna [2004] and Brenner and Struzyna [2005].

Now we turn to partitioning, which is a movement minimization approach in BonnPlace:

Partitioning Problem
Instance: A set of cells $\mathcal{C}$, a set $\mathcal{W}$ of windows, $\text{dist} : \mathcal{C} \times \mathcal{W} \to \mathbb{R}_+$,
Task: Find an assignment $\rho : \mathcal{C} \to \mathcal{W}$, such that the capacities are satisfied:

$$\sum_{c \in \mathcal{C}:\rho(w)} \text{size}(c) \leq \text{capa}(w) \ \forall w \in \mathcal{W} \tag{3.24}$$

and

$$\sum_{c \in \mathcal{C}} \text{dist}(c, \rho(c)) \tag{3.25}$$

is minimized.

For each cell $c \in \mathcal{C}$ its (one-dimensional) area $\text{size}(c) \in \mathbb{R}$ is considered.

Brenner [2005]'s Multisection algorithm can compute an optimal solution to the Partitioning Problem in $\mathcal{O}\big(nm^2(\log(n) + m\log(m))\big)$-time, in which at most $m-1$ cells are assigned fractionally. As the total area of the fractionally assigned cells is usually very small w.r.t. the window capacities, the rounding does not cause significant errors (see also Vygen [2005]). The runime bound, however, suggests that the number of windows involved in one partitioning step should not be too big.

Remark 3.4.1 (Uniform Density Adjustment) As the total capacity of target windows may be insufficient (e.g. for rounding reasons from earlier partitioning steps), the capacity in these windows is adjusted *uniformly.* Let x denote the total capacity of $\mathcal{W}$. Then the capacity of each unblocked window in $\mathcal{W}$ is scaled by a factor of $1+\max\{0, \big(\sum_{c\in\mathcal{C}_W} \text{size}(c) - x\big)/x\}$.. Blocked windows, i.e. windows with $x = 0$ are not considered for density adjustment.

The movement based approach for partitioning in combination with terminal propagation causes another problem: cells assigned to some window sometimes have all their neighbors on one side, the right one say. Then the quadratic netlength minimization places them all at one coordinate. Movement based partitioning starting from such a placement would cause arbitrary decisions. To this end, linear constraints on the cells in the window are imposed: a `ConstraintQP`(Γ_i) is computed, prescribing the center-of-gravity *(COG)* of the cells. As COG there are various heuristics: one can choose the center of the window as proposed in Kleinhans et al. [1991], but to avoid unnecessary movement in instances which do not require uniform distribution, one should not move the new COG too far away. In BonnPlace, if the cells are crowded too closely in one corner, the new COG is essentially moved towards the center-of-gravity of the free area in the window in such a way that the rectangle centered at the new COG with edge lengths corresponding to twice the distance between the new COG and the crowded corner scaled with average cell density in the window can accommodate all its cells (without paying attention to particular blockages).

3.4.1 Recursive Partitioning

In the recursive partitioning of level $i = 1, \ldots, k$ one computes $\texttt{ConstraintQP}(\Gamma_{i-1})$, prescribing new COGs to cells. Then for each window W induced by Γ_{i-1}, a MULTISECTION instance is created with all cells in $\mathcal{C}_W$ and the window set $\mathcal{W} := \{w \text{ window induced by } \Gamma_i | w \subset W\}$ (i.e. to the subwindows of W induced by Γ_i).

3.4.2 Repartitioning

Cell assignments via MULTISECTION do not consider connectivity explicitly, a revision of such decisions is hence sometimes desirable. For this purpose BONNPLACE applies the *repartitioning* scheme. Given a partition of cells in Γ_i, repartitioning considers coarse $s \times t$ windows consisting of s subsequent horizontal and t subsequent vertical windows induced by Γ_i. Usually $s, t \in \{2, 3\}$.
For each such coarse window W, $\texttt{LocalQP}(\mathcal{C}_W, W)$ and $\texttt{LocalConstraintQP}(\mathcal{C}_W, W)$ are computed, in which all cells outside of W are considered as fixed objects. Then a MULTISECTION step is applied with all $s \times t$-windows forming W as targets and again a `LocalQP` is computed with terminal propagation to preserve the latest partitioning decision. If the resulting bounding box netlength has improved, the new partitioning is accepted, otherwise one rejects it and restores the original solution.
One loop of repartitioning that visits all $s \times t$-windows is called an *iteration*. Another iteration is started if the previous one improved the netlength by a certain percentage. In case of recursive partitioning another iteration is started when this gain exceeds 1%, and the number of iterations is bounded by 5 for runtime reasons.

3.4.3 Iterative Partitioning

Iterative partitioning in level $i = 1, \ldots, k$ assigns cells to Γ_i according to the cells' locations, without paying attention to capacity bounds. Then one considers coarse windows W which consist of 2×2-windows induced by Γ_i as in repartitioning. A global heap is created, storing all such coarse windows W with the lexicographic key (a, b, c) consisting of three values sorted by increasing key values. Assume $W = \bigcup_{i=1}^{4} w_i$, then W's key consists of:

$a :=$ $\max\{1, \sum_{C \in \mathcal{C}_W} \text{size}(c)/\text{capa}(W)\}$ (the average overload in W, but at least 1),

$b :=$ $-\max\{\sum_{C \in \mathcal{C}_{w_i}} \text{size}(c)/\text{capa}(w_i) | i = 1, \ldots, 4\}$ (the negative maximum overload in the fine windows of W)

$c :=$ distance between $\text{COG}(\mathcal{C}_w)$ and the center of W where w is the window attaining the maximum overload among $w_1, \ldots, w_4$.

Once the heap is filled, the algorithm picks the coarse window at the top of the heap, which has some violation in its finer windows but provides capacity to reduce these and the cells of the violated subwindow are close to the center. In this window W a repartitioning is computed and the solution is accepted if the maximum overload among the finer windows in W decreased. The keys are updated and one repeats these steps until no overloads persist.

3.4.4 Macro Placement and Legalization

Subsequent grid refinement leads to situations in which large cells become relatively big w.r.t. the window they are assigned to. These large cells cannot be moved within a small window. To this end, `MacroPlacement` fixes the large cells during the global placement loop in a legal position as soon as the cell's edge lengths exceed a certain percentage of the window's height and width. For each window BonnPlace tries to place large cells in its window in a greedy manner first. If some cells cannot be placed in their windows, another try is started to place them legally in the chip area, dropping the window constraints. The shapes of the fixed cells become blockages.
When the finest grid is reached, cells are assigned to windows which correspond to parts of cell rows and all non-standard cells are fixed. Finally, the standard-cell legalization as proposed by Brenner and Vygen [2004] is called.

3.4.5 Congestion-Driven Placement

In congestion-driven placement, the placement tool tries to avoid routability problems during and after the placement. In BonnPlace the idea of Brenner and Rohe [2002] finds application. To this end, the simple partitioning for levels 3-7 is replaced by a usual iterative partitioning, then a quick and rough routing is computed and in congested areas the cells are *inflated*, i.e. their area demand is artificially increased (but not their physical shapes). This step makes a partitioning possibly infeasible, so another iterative partitioning is used to reduce density violations.

3.4.6 Parallelization

In BonnPlace several subroutines are able to run in parallel mode. A simple implementation allows the use of up to 4 processors. The parallelization is based on the shared-memory model using the pthread-library (see Buttlar, Farell and Nichols [1996]). Three time consuming routines are parallelized in BonnPlace: the global QP, repartitioning and iterative partitioning. The global QP works on a queue model, where each grid row/column corresponds to one job and is put into the queue. Then a set of parallel workers picks a job, computes the quadratic program, saves the locations to the cells and continues as long as the queue is not empty. Repartitioning and iterative partitioning are subject to static geometric partitioning: the chip area is divided into four quadrants, say $0, 1, 2, 3$ and first the diagonally opposed quadrants $0, 2$ are processed in parallel, then synchronized, followed by parallel processing of the diagonally opposed quadrants $1, 3$. We will present an improved implementation in Chapter 6.

3.4.7 Overall Algorithm

The overall structure of BonnPlace is summarized in Algorithm 1. The grid refinement $\Gamma_i \rightarrow \Gamma_{i+1}$, $i = 1, \dots, k$ usually doubles the number of cut lines in either direction, which yields 4 subwindows in Γ_{i+1} for each window induced by Γ_i. However, as the vertical cuts impose a much finer structure than the horizontal ones, for the latest levels each window is split by a 2×3 grid. Depending on the chip size, the number of levels is usually between 7 and 11. As `Partitioning` either the recursive partitioning is applied or iterative partitioning. In the first case, repartitioning is used with up to 5 iterations, in the second with up to two.

Algorithm 1: BonnPlace

```
1  Input(A placement instance)
2  Output(A placement)
3  AdjustCapa ($\mathcal{C}, \mathcal{A}$)
4  Let $\Gamma_0, \dots, \Gamma_k$ be the subsequent grid refinements.
5  QP ($\Gamma_0$)
6  for $i = 1, \dots, k$ do
7      MacroPlacement ($\Gamma_{i-1}$)
8      Partitioning ($\mathcal{C}, \Gamma_{i-1}, \Gamma_i$)
9      QP ($\Gamma_i$)
10     Repartitioning
11 Legalization
```

Chapter 4

Movebounds

In the past, controlling placement basically meant manipulating the netlist or introducing blockages or density constraints. The latter applied to *all* placeable objects at the same time. Although a rough concept of position constraints for *a subset* of cells, the *movebounds*, has appeared in different chip design methodologies and applications (Bednar et al. [2002], Osler [2004], Hu et al. [2004], Xiong et al. [2006]) and is even part of the OpenAccess Si2 [2009] standard, there has not been any systematic work published on this until now. In this chapter we are first going to briefly motivate such constraints, specify the concept of movebounds and discuss their properties. Later, we will focus on the interaction between movebounds and density constraints in placement and present a partitioning based placement algorithm which respects movebounds.

4.1 Motivation

The task of VLSI placement is to find positions and orientations for cells in the chip area in such a way that the cells do not overlap. Density as well as blockage constraints are considered and the final result respects timing and routability constraints. Beyond explicit netlist manipulations, the different possibilities of influencing the global placement with *a-priori* constraints traditionally consist of blockages and density constraints.
Another issue, namely different grid specifications, plays a special role here, as there is usually one grid for the entire design, reflecting the power-supply of the standard-cell library and possibly a small number of slightly coarser sub-grids for subsets of cells (gatearrays, special latches) reflecting particular routability issues. These grids are typically uniformly distributed all over the entire chip area and their granularity is usually too fine to be considered as a global placement constraint. The grid constraints enter into play either during legalization and detailed placement phases for small objects or during the MacroPlacement subroutine for a small amount of large cells.
So, from global placement point of view there have been only blockage and density constraints which could be used for driving the placement. Now the question arises as to how blockages and density constraints can be applied to a *subset* of cells: the issue of *movebounds* emerges.

Applications of such constraints in VLSI-Placement are diverse. First of all, the designer is able to influence the placement outcome towards a particular solution, which might be necessary for timing or routability issues as reported e.g. by Salz [2009], Osler [2004], or to control clock domains Nam and Villarubia [2009].
At the same time, placement is often used in very early design stages, where parts of the netlist will be subject to major logical and physical reviews. If the designer is aware of such future changes, but they are not yet part of the instance, it is frequently desirable to reserve space and restrict the solution space. This strategy allows the available netlist to be optimized early without pretending as if this was already the final data.
Beyond the opportunities for designers in taking control of the placement manually, there are different major applications of movebounds without manual interaction: the movebounds are indeed method of choice, wherever placement of certain cells is bounded by electrical constraints (e.g. *IO-cells* with maximum resistance bounds impose such requirements).
Third, the increasing number of hierarchical architectures opens a new vast field for movebound applications: Hu et al. [2004] use movebounds for the simultaneous placement of different power-levels, the *voltage islands* on a single design. Bednar et al. [2002] discuss the trade-offs between flat and hierarchical placement and use movebounds as a methodology in between. The movebounds allow the detailed internal view of hierarchical subunits (*SoCs, RLMs*), but maintain the overall hierarchical process at the same time. This concept is also known as *glass box* or *transparent placement.*

4.2 Movebounds and their Properties

We want to distinguish between two types of movebounds: the first type should simply enforce the associated cells to be placed within the defined area without having any influence on non-associated. The second type should in addition state a blockage to the non-associated cells. Movebound constraints are considered either as hard constraints which have to be satisfied or as soft ones. The latter can be relaxed by the placement tool. We formalize the definition first:

Definition 4.2.1 (Movebounds) A movebound M is a triple $(A(M), \eta(M), \xi(M))$ where $A(M)$ is a set of axis-parallel, non-empty rectangles, $\eta(M) \in \{\text{hard}, \text{soft}\}$ states whether the movebound is a hard or soft constraint and finally $\xi(M) \in \{\text{incl}, \text{excl}\}$ encodes whether the movebound area is accessible for non-associated cells (inclusive) or states a blockage for other cells (exclusive).

This definition extends and specifies the concept of a movebound (called *cluster*) in the OpenAccess standard Si2 [2009], in which only the combinations inclusive hard, exclusive (here exclusive means hard for the associated cells and soft for the remaining) and suggested (which basically correspond to inclusive soft movebounds in our case) are provided. Unlike the definition in Si2 [2009], we distinguish exclusive movebounds to be a hard condition for the associated cells and blocked area for the others. The exclusive soft movebounds are considered to be soft for the associated cells and soft for the non-associated ones. An exclusive movebound in terms of Si2 [2009] can be modeled by a combination of an inclusive hard movebound for the member cells and an exclusive soft movebound with the same area without cells. Although any exclusive hard movebound condition with area A can be modeled as an inclusive hard movebound for the non-associated cells with complementary area A^C, the OpenAccess standard does not cover the case of exclusive hard movebounds in combination with soft movebounds.

To abbreviate the notation, for a given set of movebounds $\mathcal{M}$ we will write $\mathcal{M}^{\text{excl}}$ for the subset of exclusive, $\mathcal{M}^{\text{incl}}$ inclusive, $\mathcal{M}^{\text{hard}}$ hard and $\mathcal{M}^{\text{soft}}$ soft movebounds and $\mathcal{M}^{\text{exclhard}}$ ($\mathcal{M}^{\text{exclsoft}}\dots$) for exclusive hard and so on.

Now, let us omit soft movebounds for the time being; we will focus on them later. Hard movebounds are treated as hard constraints during the placement, which requires a restriction of the legality notion. Formally:

Definition 4.2.2 (Legal placement with movebounds) Given a set of cells $\mathcal{C}$, a set of hard movebounds $\mathcal{M}$ and a cell-movebound assignment $\mu : \mathcal{C} \to \mathcal{M}$, we call a placement $(x, y) : \mathcal{C} \to \mathbb{R}^2$ *legal w.r.t. movebounds* if:

$$A_{(x,y)}(c) \subset \bigcup A(\mu(c)) \qquad \forall c \in \mathcal{C} \tag{4.1}$$

and

$$\forall c \in \mathcal{C}, \forall M \in \mathcal{M}^{\text{exclhard}} \backslash \{\mu(c)\} \Rightarrow A_{(x,y)}(c) \dot{\cap} \bigcup A(M) = \emptyset. \tag{4.2}$$

When considering hard movebounds, it is indeed sufficient if each cell is assigned to *exactly one* movebound. It should be noted that for a given set of movebounds $\mathcal{M}$, cells which are not assigned to any movebound can be assigned to a new hard movebound with area $\mathcal{A} \setminus \bigcup_{M \in \mathcal{M}^{\text{exclhard}}} A(M)$.
Cell – movebound assignments, in which cells are potentially assigned to several hard movebounds can be replaced by assignments in which a cell is mapped to one movebound with an area corresponding to the intersection of the original movebound areas. If a cell, say $c \in \mathcal{C}$, is assigned to some soft and some hard movebounds, such a replacement will be trivial, if the non-soft movebound area of c does not intersect any hard movebound area of c. Here, the hard movebounds dominate and the soft movebound conditions have to be dropped. It is not so straightforward in cases where soft and hard movebound areas of some cell intersect. All soft movebound areas which do not intersect the hard movebounds can again be dropped, but it is not clear how to model soft movebound violations within the hard movebound area. In practice, however, it is not a real issue as such nested movebound conditions between soft and hard movebounds can also be modeled by artificial nets. We will thus restrict ourselves to cases in which a cell is assigned to precisely *one* movebound. We also write $\mathcal{C}_M := \{c \in \mathcal{C} | \mu(c) = M\}$. We can also expect for each exclusive hard movebound, say M, $A(M) \dot{\cap} A(N) = \emptyset$, for all $N \in \mathcal{M}, N \neq M$: the area $A(M)$ is not

accessible for any other movebounds' cells anyway and such situations can be scanned directly at the input. With the latest requirement we obtain directly:

Corollary 4.2.3 Let $\mathcal{M}$ be a set of movebounds and $\mu : \mathcal{C} \to \mathcal{M}$ a cell-movebound function. If each exclusive hard movebound does not intersect any other movebound's area, then a legal placement of $\mathcal{C}$ is legal with respect to movebounds if and only if:

$$\forall c \in \mathcal{C} : A_{(x,y)}(c) \subset \bigcup A(\mu(c)). \tag{4.3}$$

$\square$

4.3 Partitioning-Based Placement with Movebounds

Rectanglular movebounds are linear constraints and could be respected by placement algorithms which minimize bounding-box netlengths. In particular, in combination with a branch-and-bound approach, even an optimal placement respecting such movebounds could be found in theory by an exponential-time algorithm.
Such an approach however is not even applicable to instances with few tens of placeable objects, not to mention modern ASICs with millions of cells. In addition, it is mandatory to respect densities for practical purposes. Both reasons demonstrate that we need a different strategy here. We also drop the convexity requirement to the movebound areas and allow arbitrary shapes consisting of unions of axis-parallel rectangles.

Placement with Movebounds

Instance: A placement instance $(\mathcal{C}, \mathcal{N}, P, \gamma, \mathcal{A}, \mathcal{B}, \mathcal{M}, \mu, \mathcal{D})$ with movebounds.
Task: Find a legal placement which is legal w.r.t. movebounds with minimum bounding-box netlength of $\mathcal{N}$.

Of course the placement problem reduces to the problem with hard movebounds by specifying $\mu(c) = \mathcal{A}$ for each cell $c \in \mathcal{C}$. The placement with movebounds is thus at least as hard as the placement without them. Again, we will relax the problem for global placement purpose similarly to the approach without movebounds and treat each cell as shapeless object. We can ask for the necessary conditions for a feasible solution. As movebound areas can overlap, the cells of each movebound have to share the common parts. It is hence not enough to ask if all cells' sizes fit into the chip area, but we have to check the capacity condition for each *subset of movebounds*:

Remark 4.3.1 Fix a placement instance with movebounds consisting of $\mathcal{C}, \mathcal{M}, \mu$ as above. Then if the instance is feasible, we have $\forall \mathcal{M}' \subset \mathcal{M}$:

$$\sum_{c \in \mathcal{C} : \mu(c) \in \mathcal{M}'} \text{size}(c) \le \text{capa}\Big(\bigcup_{M \in \mathcal{M}'} A(M) \Big). \tag{4.4}$$

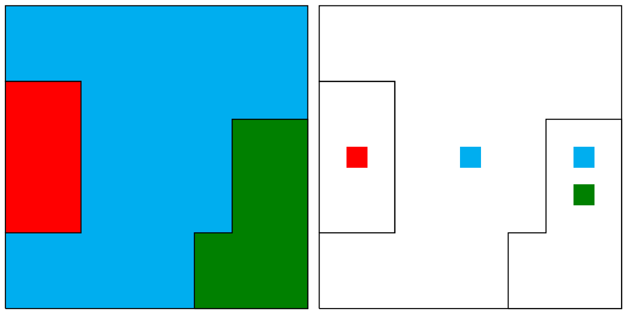


Figure 4.1: Three movebounds: an exclusive hard N, and two inclusive M,L which overlap (left). The resulting three maximal regions (right).

As it seems very inconvenient to evaluate the area and its capacity for subsets, we introduce the concept of:

Definition 4.3.2 (Regions) Given a set of movebounds $\mathcal{M}$, a region r is a set of axis-parallel rectangles with the following properties:

i. The interior of each pairwise intersection of rectangles of r is empty

ii. $\forall M \in \mathcal{M} : r \dot{\cap} A(M) \neq \emptyset \Rightarrow r \subseteq A(M)$.

We say a movebound M *covers* a region r whenever $r \subseteq A(M)$.

From now on we will consider partitions of the chip area into disjoint (in terms of $\dot{\cap}$) regions.

The regions reflect a decomposition of the chip into homogeneous areas in a certain way. For each region r, each point in r (possibly except for the boundary) lies in the same movebounds. Even if we ask for the minimal number of regions which are needed to partition a set of movebounds $\mathcal{M}$, we can obtain $2^{|\mathcal{M}|}$ of them in the worst case as any movebound's area can intersect any other. Better bounds for the number of regions can be obtained by considering the number of rectangles: if the movebound constraints consist of l rectangles, there exist a decomposition into regions using $\mathcal{O}(l^2)$ rectangles, which is polynomial in l. In practice, one usually has for a set $\mathcal{R}$ of regions $|\mathcal{R}| = \mathcal{O}(|\mathcal{M}|)$. It is thus reasonable not to try all intersections when computing regions. To this end one can exploit the property that, if the area of a movebound subset, say $\mathcal{M}'$, does not intersect any other movebound's area, say that of $N \in \mathcal{M} \setminus \mathcal{M}'$, then the intersection of the areas of $\mathcal{M}'$ and any other subset $\mathcal{M}''$ containing N will also be empty. Thus, the computational effort of computing the regions can indeed be limited to the number of intersections and geometrical complement operations and the number of intersections of movebounds.

The regions now allow a very nice combinatorial formulation of the necessary feasibility condition:

Lemma 4.3.3 Consider $\mathcal{C}, \mathcal{M}, \mu$ as above and let $\mathcal{R}$ be a decomposition of the areas of $\mathcal{M}$ into regions. Define the graph

$$G = \big(\{s,t\}\dot\cup\mathcal{C}\dot\cup\mathcal{R}, \{(s,c)|c \in \mathcal{C}\} \cup \{(c,r)|r \subset A(\mu(c))\} \cup \{(r,t)|r \in \mathcal{R}\}\big)$$

and the MaxFlow-Instance (G, u, s, t) with $u((s,c)) = \text{size}(c) \forall c \in \mathcal{C}$, $u((r,t)) = \text{capa}(r)$ for all $r \in \mathcal{R}$ and $u \equiv \infty$ elsewhere. Then the maximum flow in (G, u, s, t) has the value $\text{value}(f) = \sum_{c\in\mathcal{C}} \text{size}(c)$ if and only if (4.4) holds.

Proof: Assume (4.4) holds and the maximum flow value is $\text{value}(f) < \sum_{c\in\mathcal{C}} \text{size}(c)$. By the Max-Flow-Min-Cut Theorem, there is an $s-t$ cut of total capacity value(f). Let $\mathcal{C}'$ denote the set of cell nodes and $\mathcal{R}'$ the region nodes separated by the cut from t. Let $\mathcal{M}' := \{\mu(c)|c \in \mathcal{C}'\}$. We claim that $\mathcal{M}'$ is a set of movebounds violating (4.4). First observe, that for M in $\mathcal{M}'$ all regions covered by M are in $\mathcal{R}'$: otherwise there were an edge with infinite capacity in the cut. Thus,

$$\begin{aligned}\sum_{c\in\mathcal{C}'}\text{size}(c) + \sum_{c\in\mathcal{C}\setminus\mathcal{C}'}\text{size}(c) > \text{value}(f) &\geq \sum_{r'\in\mathcal{R}'} u((r',t)) + \sum_{c\in\mathcal{C}\setminus\mathcal{C}'}\text{size}(c)\\ &= \text{capa}\big(\bigcup_{M\in\mathcal{M}'} A(M)\big) + \sum_{c\in\mathcal{C}\setminus\mathcal{C}'}\text{size}(c),\end{aligned} \tag{4.5}$$

so the condition is violated for $\mathcal{M}'$. To see the converse, assume $\text{value}(f) = \sum_{c\in\mathcal{C}} \text{size}(c)$. For any $\mathcal{M}' \in \mathcal{M}$ we have:

$$\begin{aligned}&\text{capa}\big(\bigcup_{M\in\mathcal{M}'} A(M)\big) = \sum_{r \text{ covered by } M\in\mathcal{M}'} \text{capa}(r) \geq \sum_{e=(r,t)\in E(G): r \text{ covered by} M\in\mathcal{M}'} f_e\\ = &\sum_{e=(c,r)\in E(G): r \text{ covered by } M\in\mathcal{M}'} f_e \geq \sum_{c\in\mathcal{C}:\mu(c)\in\mathcal{M}'} \text{size}(c).\end{aligned} \tag{4.6}$$

□

For complexity purposes it is sufficient here to contract all cell nodes which belong to the same movebound and compute the MaxFlow on the contracted graph:

Corollary 4.3.4 Consider $\mathcal{C}, \mathcal{M}, \mu, \mathcal{R}$ as above. Then the condition 4.3.1 can be checked in $\mathcal{O}(|\mathcal{C}| + |\mathcal{M}|^2|\mathcal{R}|)$ time.

Proof: Scanning of $\mathcal{C}$ is required to construct the nodes $\mathcal{C}/\mathcal{M}$. We have $|\mathcal{M}| \leq |\mathcal{R}|$ and obtain the bound by the algorithm of Gusfield, Martel, and Fernández-Baca [1987] for bipartite graphs. □

4.3.1 Partitioning with Movebounds

Recall the structure of top-down iterative partitioning in global placement's recursive partitioning (Section 3.4.1). At any stage we are given a window W with a set of cells $\mathcal{C}_W$ assigned to W with $\sum_{c\in\mathcal{C}_w} \text{size}(c) \leq \text{capa}(W)$. Now, given a refinement $\{w_1, \ldots, w_k\}$ of W, we ask for a partitioning $\rho : \mathcal{C}_w \to \{w_1, \ldots, w_k\}$ respecting the capacities in each of the subwindows $w_i, i = 1, \ldots, k$. When introducing movebounds, we have to make sure that each subwindow does not only contain at most a certain amount of cell area, but *the right amount*. Similarly to Remark 4.3.1 we define for a set of windows $\mathcal{W}$:

Definition 4.3.5 Fix a set $\mathcal{C}$ of cells, a set $\mathcal{M}$ of movebounds, a cell-movebound function $\mu : \mathcal{C} \to \mathcal{M}$ and a set of windows $\mathcal{W}$. A partitioning $\rho : \mathcal{C} \to \mathcal{W}$ with movebounds is called legal if $\forall w \in \mathcal{W}, \forall \mathcal{M}' \subseteq \mathcal{M}$:

$$\sum_{c \in \mathcal{C}: \rho(c) = w \text{ and } \mu(c) \in \mathcal{M}'} \text{size}(c) \leq \text{capa}\big(w \cap \bigcup_{M \in \mathcal{M}'} A(M)\big). \tag{4.7}$$

Basically, this can be achieved using a decomposition of *each* window $w \in \mathcal{W}$ into a set of regions $\mathcal{R}_w$ and solving a Multisection problem. The density constraints in global placement (cf. Section 3.3) now translate to capacities for regions. By decomposition, we have $\sum_{r \in \mathcal{R}_w} \text{capa}(r) = \text{capa}(w)$ for each window $w \in \mathcal{W}$, i.e. the total capacity per window remains the same when using regions.

Algorithm 2: PartitioningWithMovebounds1$(\mathcal{C}, \mathcal{W}, \mathcal{M}, \mu)$

```
1  for $w \in \mathcal{W}$ do
2      $\mathcal{R}_w$ :=ComputeRegions $(w, \mathcal{M})$
3  for $c \in \mathcal{C}$ do
4      for $r \in \bigcup_{w \in \mathcal{W}} \mathcal{R}_w$ do
5          if $\mu(c)$ covers $r$ then
6              cost$((c, r))$ := dist$(c, r)$;
7          else
8              cost$((c, r))$ := $\infty$;
9  $\rho'$:=Multisection $(\mathcal{C}, \bigcup_{w \in \mathcal{W}} \mathcal{R}_w, \text{cost})$
10 for $c \in \mathcal{C}$ do
11     $\rho(c) := w$ iff $\rho'(c) \in \mathcal{R}_w$
12 return $\rho$.
```

Lemma 4.3.6 Assume the partitioning instance $(\mathcal{C}, \mathcal{M}, \mu, \mathcal{W})$ with movebounds is feasible. Then Algorithm 2 computes a legal fractional partitioning with minimum total assignment costs in time $\mathcal{O}\big(k^2|\mathcal{C}|(\log(\mathcal{C}) + k\log(k))\big)$ with $k = \sum_{w \in \mathcal{W}} \mathcal{R}_w$.

Proof: As the instance is feasible, there exists an assignment with finite cost which by correctness of Multisection will be found. For each window $w \in \mathcal{W}$ the MaxFlow-condition of Lemma 4.3.3 induced by $\mathcal{C}_w := \{c \in \mathcal{C} | \rho(c) = w\}$ and $\mathcal{R}_w$ implies the legality of the partitioning ρ. The runtime follows directly from the worst-case number of regions and MULTISECTION's runtime (Brenner [2005]). $\square$

Lemma 4.3.6 reveals the problematic issue here: the complexity of such a partitioning depends exceedingly on the number of regions in the instance, which in the worst case can be of the order $\mathcal{O}(|\mathcal{W}|2^{|\mathcal{M}|})$ in theory. Fortunately, in VLSI instances the number of movebounds is very small compared to the number of cells, usually up to a few tens and even on chips with flattened hierarchy the total amount of movebounds did not exceed 205 in the largest cases. On the other hand, usually only few of them, if any, intersect in practical cases, so we observe a small number of regions in a window.

For the special case of overlap-free movebounds, the runtime of Algorithm 2 is thus polynomial in $|\mathcal{M}|$, namely $\mathcal{O}\Big((|\mathcal{M}||\mathcal{W}|)^2 \cdot |\mathcal{C}| \cdot \big(\log(\mathcal{C}) + \big(|\mathcal{M}||\mathcal{W}|\big)\log(|\mathcal{M}||\mathcal{W}|)\big)\Big)$. But even in these cases the number of regions may reach into the hundreds, which states a significant computational challenge to the MULTISECTION algorithm, both in terms of memory consumption and runtime. MULTISECTION was initially designed to handle *small* (say 4 or 9) right-hand-side nodes and may be inefficient on larger instances. We are indeed interested in a partitioning of cells to windows, only using the regions to choose the right amount of cells for assignment. So in practice we can accelerate the partitioning by two different methods.
The first one is motivated by the fact that an exclusive hard movebound can be treated independently during PARTITIONING – it is a blockage for other cells and other areas are blocked for movebound cells.
The second idea is based on the observation that movebound areas are frequently contained in *one* region $r \in \mathcal{R}_w$ of a window $w \in \mathcal{W}$. Assume M is such a movebound (and M is an inclusive hard movebound). This makes no partitioning decision necessary for the movebound cells $\mathcal{C}_M$, since all of them can be assigned to r and hence $\rho(c) \equiv w \ \forall c \in \mathcal{C}_M$. In order to reduce the number of options for the remaining cells, consider *maximal* regions and observe that if r is covered by more than one movebound, its area is included in $\bigcap_{N\in\mathcal{M}'} A(N)$, for some maximal movebound subset $\mathcal{M}' \subset \mathcal{M}$ with $M \in \mathcal{M}'$. In this case, as each region is uniform in terms of its covering movebounds, there exists a unique region $r' \in \mathcal{R}_w$, with:

$$r' = \bigcap_{N\in\mathcal{M}'} A(N) \setminus A(M). \tag{4.8}$$

Observe that except by M, r' and r are covered by the same movebounds. Thus, we can *merge* the region r to the area of r' by setting $A(r') := A(r') \cup A(r)$ and adjusting the capacity of r' to:

$$\text{capa}(r \cup r') := \text{capa}(r) + \text{capa}(r') - \sum_{c\in\mathcal{C}_M} \text{size}(c), \tag{4.9}$$

which is nothing other than the remaining capacity for the remaining cells of $\mathcal{M}'$. We refer to the procedure as the MERGEREGIONS step. Formally, r' is no longer a region in this instance, but this situation is similar to that of inclusive hard movebounds without cells. Such movebounds can be dropped. Putting both facts together yields a modified Algorithm 3. For this algorithm, in fact, no improvement in the worst-case behavior can be guaranteed in general, but it significantly reduces the Multisection runtime in practice.

Lemma 4.3.7 Assume the partitioning instance $(\mathcal{C}, \mathcal{M}, \mu, \mathcal{W})$ with movebounds is feasible. Then Algorithm 3 computes a legal partitioning with movebounds and finite assignment costs.

Proof: Follows directly from the remarks above. $\square$

Partitioning based placements which work recursively base on the fact that once a partitioning decision has been taken, the subsequent partitioning decisions can be made independently (of course dropping integrality requirements). When dealing with movebounds, we are able to exploit the same idea:

Lemma 4.3.8 (Partitioning Invariance with Movebounds) Let a set $\mathcal{C}$ of cells, a set $\mathcal{M}$ of movebounds, a cell-movebound function $\mu : \mathcal{C} \to \mathcal{M}$ and a set of windows $\mathcal{W} = \{W_i | i \in I\}$ be given, where for each $W_i \in \mathcal{W}$ there is a set of subwindows $\{w_{i_j} | j \in J_i\}$ of W_i. If there is a legal partitioning $\rho : \mathcal{C} \to \mathcal{W}$, there also exists a legal partitioning into subwindows $\{w_{i_j} | j \in J_i\}$ of W_i for each $i \in I$.

Proof: : For $i \in I$, $j \in J_i$ and each window w_{i_j} in W_i, consider the set of regions $\mathcal{R}_{i_j}$ in w_{i_j}. Then $\bigcup_{j \in J_i} \mathcal{R}_{i_j}$ is a set of regions in W_i with precisely the same capacities. Thus, an infeasible partitioning instance for $\{w_{i_j} | j \in J_i\}$ would already contradict the legality of the partitioning to windows in $\mathcal{W}$, by Remark 4.3.1. $\square$

In conclusion it should be mentioned that Multisection, in fact, computes a fractional solution first and then rounds the result. The fractional solution can be modified to an integral one for all but $|\mathcal{R}| - 1$ cells while maintaining the costs. The number of regions will be higher than the number of windows during partitioning, and although Algorithm 3 reduces the number of regions treated in Multisection simultaneously and thus reduces the number of rounding effects in practice, we emphasize the necessity of proper density handling. We will see in Section 5.4 how to cope with density violations resulting from roundings.

4.3.2 Movebounds and Density Constraints

Movebounds and density constraints may contradict each other. The density is translated into capacity as shown in Section 3.3. Density constraints can make a partitioning step with movebounds infeasible, even if the partitioning without movebounds can be solved. In order to obtain a unified strategy, we will consider hard movebound constraints to be stricter than density targets. The soft movebounds on their side should state a weaker condition than density.

Algorithm 3: PartitioningWithMovebounds2($\mathcal{C}, \mathcal{W}, \mathcal{M}, \mu$)

```
1  Set $\mathcal{R}$=$\mathcal{Q} = \mathcal{M}' := \emptyset$.
2  for $w \in \mathcal{W}$ do
3      $\mathcal{R} := \mathcal{R} \cup$ComputeRegions $(w, \mathcal{M})$
4  Set $\vartheta : \mathcal{M} \to \mathcal{P}(\mathcal{R})$ to $\vartheta(M) := \{r \in \mathcal{R} |\ M$ covers $r\}$.
5  for $M \in \mathcal{M}$ do
6      if $\vartheta(M) = \{r\}$ and $r \in \mathcal{R}_w$ then
7          //Direct Assignment:
8          $\rho(c) := w \qquad \forall c \in \mathcal{C}_M$
9          $\mathcal{Q} := \mathcal{Q} \cup \{r\}$
10         $\mathcal{R} := \mathcal{R} \setminus \{r\}$
11         $\mathcal{M}' := \mathcal{M}' \cup \{M\}$
12     else
13         if $M \in \mathcal{M}^{exclhard}$ then
14             //Exclusive Hard:
15             $\rho'$:=Multisection $(\mathcal{C}_M, \vartheta(M), \mathrm{dist}_{L_1})$
16             $\rho(c) := w$ iff $\rho(c) \in \vartheta(M) \cap \mathcal{R}_w,\ \forall c \in \mathcal{C}_M$
17             $\mathcal{R} := \mathcal{R} \setminus \vartheta(M)$
18             $\mathcal{M}' := \mathcal{M}' \cup \{M\}$
19 MergeRegions $(\mathcal{R}, \mathcal{Q})$
20 for $M \in \mathcal{M} \setminus \mathcal{M}'$ do
21     for $c \in \mathcal{C}_M$ do
22         for $r \in \mathcal{R}$ do
23             if $r \in \vartheta(M)$ then
24                 cost$((c, r))$ := dist$(c, r)$;
25             else
26                 cost$((c, r)) := \infty$;
27 $\rho'$:=Multisection $(\mathcal{C}_{\mathcal{M} \setminus \mathcal{M}'}, \mathcal{R}, \mathrm{cost})$
28 for $M \in \mathcal{M} \setminus \mathcal{M}'$ do
29     for $c \in \mathcal{C}_M$ do
30         $\rho(c) := w$ iff $\rho'(c) \in \mathcal{R}_w$
31 return $\rho$.
```

Although density constraints are considered not to impose hard restrictions in BonnPlace, they cannot be neglected in case of movebound constraints or trivially subordinated to them. Density adjustments are used for two purposes in BonnPlace. The first one is needed to cover infeasible inputs to the placement tool, the second is to deal with density violation caused by the placement engine itself. The latter can happen due to rounding effects, fixing some macros during global placement or during congestion avoidance.

Without movebounds, recall that the simple *uniform density adjust* from Remark 3.4.1 was applied to increase density by a certain percentage in each window. It should be noted that this strategy, although feasible if applied for each subset of movebounds in a partitioning instance, may cause serious problems. Depending on the order, this could lead to a dramatically larger increase of violated density conditions than necessary. It is important to mention that not only the *amount* of density violation has to be penalized, but increasing density in regions with high target density should be made more expensive than in regions which provide a large amount of white space.

For the purpose of increasing capacity of the right hand side of (4.4), instead of a simple greedy approach we propose a MinCostFlow model with non-linear convex costs. We apply a classical strategy for handling non-linear costs by piecewise linear interpolation as proposed by Ahuja, Batra and Gupta [1984]. To this end we introduce the following graph with parallel arcs:

$$G = \big(\{\mathcal{C}_M | M \in \mathcal{M}\} \cup \mathcal{R} \cup \{t\}, \{(\mathcal{C}_M, r) | M \text{ covers } r\} \bigcup_{r \in \mathcal{R}} E_r\big) \tag{4.10}$$

with $E_r := \{e_r^0, e_r^1, \dots, e_r^d\}$ and each $e_r^i = (r,t), i = 0, \dots, d,\ r \in \mathcal{R}$. We set $b(\mathcal{C}_M) := \sum_{c \in \mathcal{C}_M} \text{size}(c)$ for each $M \in \mathcal{M}$, $b(t) := -\sum_{c \in \mathcal{C}} \text{size}(c)$ and $b(r) = 0\ \forall r \in \mathcal{R}$. All $(\mathcal{C}_M, r)$ edges have costs zero and are uncapacitated. For the others, we introduce the following costs and capacities:

$$\text{capa}(e_r^0) := \text{capa}(r), \qquad \text{cost}(e_r^0) := 0 \tag{4.11}$$

$$\text{capa}(e_r^d) := \infty, \qquad \text{cost}(e_r^d) := \infty. \tag{4.12}$$

For a region $r \in \mathcal{R}$, its potentially remaining capacity amount of $\text{free\_area}(r) - \text{capa}(r)$ will be distributed among the arcs $e_r^i, \dots, e_r^{d-1}$ in chunks of $\text{step}(d,r) := \frac{1}{d}\big(\text{free\_area}(r) - \text{capa}(r)\big)$:

$$\text{capa}(e_r^i) := \text{step}(d,r)\ i = 1, \dots, d-1. \tag{4.13}$$

The costs for using these arcs increase with the growing density violation and should also reflect the initial region density: it should be cheaper to provide 10% more density in a region with target density of 30% than in one with target density of 50%. For these reason, we set the costs of arc $e_r^i,\ i = 1, \dots, d$ as the discretized monotone increasing values of the function $p_r : x \to \max\{0, \frac{\sqrt{1 - \text{capa}(r)/\text{free\_area}(r) + x}}{1-x}\}$, where x corresponds to the density thresholds in the interval $[\frac{\text{capa}(r)}{\text{free\_area}(r)}, 1)$. Thus:

$$\text{cost}(e_r^i) := p_r\Big(\frac{\text{capa}(r) + i\text{step}(d,r)}{\text{free\_area}(r)}\Big) - p_r\Big(\frac{\text{capa}(r)}{\text{free\_area}(r)}\Big) \text{ for } i = 1, \dots, d-1.$$

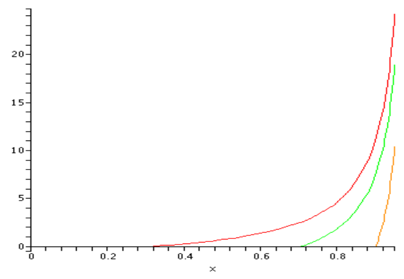


Figure 4.2: The density violation penalties p for initial densities of $30\%, 70\%$ and 90% in red, green and orange, respectively.

In this model, an additional price has to be paid for each higher density threshold until the highest possible density of 100% is reached. The prices increase with the amount of additional density required and are scaled with the square root of the *safety margin*, the distance to density values corresponding to 100%.
Once a MinCostFlow f of the above instance is computed, we set for each region $r \in \mathcal{R}$:

$$\text{capa}'(r) := \max\Big\{ \sum_{e \in \delta_G^+(r)} f_e, \text{capa}(r) \Big\} \tag{4.14}$$

It is straightforward to see:

Lemma 4.3.9 The necessary condition of Remark 4.3.1 is satisfied with $\text{capa}'(r) \leq \text{free\_area(r)} \quad \forall r \in \mathcal{R}$ if and only if the optimal solution of the above MinCostFlow instance has finite costs. □

We call this procedure *balanced density increase* and we make use of it globally to be able to deal with infeasible inputs as well as in a local manner, each time we need a partitioning decision. The latter is needed to cope with possible density violations caused by the placement itself.
The procedure `AdjustCapacityMB` checks for a given window W, whether all cells in this window belong to the same movebound. If this is the case, density is increased *uniformly* for all regions in W covered by this movebound, otherwise the balanced density increase is performed in W.

4.3.3 Soft Movebounds

Thusfar, we have considered only hard movebound conditions, and we now focus on the soft ones. Here, we will restrict ourselves to the cases where a cell has at most one movebound and the penalty function is monotone in the distance to the movebound area. In addition, as mentioned, soft movebounds should be interpreted as weaker constraints than densities. For each cell c associated to some inclusive and rectangular soft movebound M, one can introduce two nets connecting the movebound's lower left corner to the cell's lower left

corner, and the movebound's upper right corner to the cell's upper right corner. For other cost penalty functions based on the distance from a cell to its soft movebound's area, a piecewise linearization can be applied, resulting in two nets with two pins each per support point. This functionality is however covered by artificial nets.
In partitioning-based placement, one could alternatively model soft movebound violations with penalties during Partitioning, where an assignment to a region which is not covered by a movebound also depends on the distance between the region and the movebound. On the other hand, soft movebounds impose constraints rather than being part of the objective function and should be respected, if possible.
The third approach, in our opinion the preferable one, is to treat soft movebounds as hard ones during global placement. We will, however, use increased costs in partitioning to penalize exclusive soft area access for non-associated cells.
Depending on the convention that soft movebounds are subordinated to density constraints, the latter proposal would treat soft movebounds analogously to hard ones as long as density constraints are feasible. If some soft movebound is not feasible with a specified density, instead of a *density increase* for the movebound, an *area adjustment* for this movebound will be applied.
To this end, a similar MinCostFlow instance as (4.10) is created and solved, using maximal regions (i.e. regions which are computed on the coarsest grid), but the result only is used for the soft movebounds' area adjustment. If any non-zero-cost edge is used by the MinCostFlow incident to some region r covered by some inclusive soft movebound, instead of increasing the density in $A(r)$, we use this information to scale the rectangles of $\{a_1(r), \cdots, a_k(r)\} = A(r)$. Let $\text{capa}'(r)$ be the capacity for the region r required by the MinCostFlow, and $\text{capa}(r)$ the original target capacity. Then the area of each rectangle $a_i(r)$ with $i = 1, \ldots, k$ becomes scaled by the factor

$$\frac{\text{capa}'(r) \cdot \text{area}(BBox(A(r)))}{\text{capa}(r) \cdot \text{area}(A(r))},$$

which results in larger rectangles $a'_i(r)$ for $i = 1, \ldots, k$. The ratio between the area of the bounding box of $A(r)$ and the area of $A(r)$ covers the particular cases of non-rectangular movebounds. Finally, the new area of the region r is set to be

$$A'(r) := \bigcup_{i=1}^{k} a'_i(r), \tag{4.15}$$

thus, the scaled area $A'(r)$ is represented by rectangles without overlaps. We refer to this step as `AdjustAreaSoftMB`, and it is performed once globally. After this step, as local blockages and density constraint variations have been considered in an approximate way, a density increase for *all* movebound areas is postponed.

4.3.4 Quadratic Programs

The quadratic netlength minimization is not innately movebound aware, but in a placement algorithm the impact of movebounds should be communicated to the quadratic netlength minimization algorithm. To this end we propose the following solution: whenever a cell is situated outside of its movebound bounding box in before a QP, we add an artificial connection between the present cell's location and the projection of this location on the bounding box of that movebound. Another, more time consuming option, would be to perform terminal propagation at the movebound's bounding box boundary.

4.3.5 Congestion-Driven Placement

In congestion driven placement cells become artificially larger (*inflation*) to control routing congestion. This means that after cell inflation the capacity condition may get violated for different movebounds. In particular, the resulting density conditions may even exceed the available free area. In this case, a MinCostFlow approach can also be applied to uniformly reduce the cell inflations until the target densities are below 100%. We omit details here as an improved uniform model will be presented in Section 5.4.

4.3.6 Legalization

Cells are treated as shapeless objects during global placement. This is no longer the case when a final position for the cells has to be found. Similarly to BonnPlace without movebounds (Algorithm 1), we distinguish two different contexts here.
In the first case, the macro legalization, we are given a small number of cells with possible non-rectangular shapes. These cells are assigned to some grid window and consume their area within that particular window. As we have to make sure that each cell is placed entirely within its movebound, we proceed movebound-wise in each window and ask for a feasible placement cell by cell, restricting the possible locations to those within the movebound. As some cells cannot be fixed in a legal location during this greedy procedure, we give them another try within their movebound, but without the side constraint to remain within the window. There is however no guarantee for this heuristic to succeed.

In the second case we have possibly millions of cells and we can rely on the work of the global placement algorithm which has respected movebounds so far. A major difference from the previous steps is that we entirely drop the soft movebound conditions at this stage. During legalization with overlapping movebounds, we will proceed region-wise, beginning with *maximal* regions for the entire chip (i.e. we forget the assignment to present windows). However, if the regions are subivided into to many non-connected areas, we subdivide them to avoid large movement over non-connected parts of the chip. Global Placement has decided on the cells' locations while being aware of densities. In legalization we will, based on the global placement locations, compute a global partitioning of the cells to the regions. For a region r, during legalization the area outside $A(r)$ is temporarily considered as blocked. The cells are then legalized region by region, using the legalization algorithm Brenner and Vygen [2004]. There is of course no guarantee of success in such a procedure. In all regions in which legalization failed, the target density is decreased and we start from scratch by reassigning cells to regions and legalizing the modified ones. In practice, one backup step is sufficient to legalize even challenging designs and this heuristic works fine even for very high target densities.

Algorithm 4: BONNPLACE: Recursive Partitioning With Movebounds

```
1  Input(*A placement instance with movebounds*)
2  Output(*A placement with movebounds*)
3  AdjustAreaSoftMB
4  AdjustCapaMB ($\mathcal{C}, \mathcal{A}$)
5  **Let** $\Gamma_0, \dots, \Gamma_k$ be the subsequent grid refinements.
6  QP ($\Gamma_0$)
7  $\mathcal{C}_0 = \mathcal{C}$
8  **for** $i = 1; i < k; ++i$ **do**
9      MacroPlacement ($\Gamma_{i-1}$)
10     ConstraintQP ($\Gamma_{i-1}$)
11     **for** *each window* $W$ *in* $\Gamma_{i-1}$ **do**
12         Let $\mathcal{W}$ be the set of subwindows of $W$ in $\Gamma_i$
13         AdjustCapaMB ($\mathcal{C}_W, W$)
14         PartitioningWithMoveBounds2 ($\mathcal{C}_W, \mathcal{W}, \mathcal{M}, \mu$)
15     QP ($\Gamma_i$)
16     Repartitioning
17 $\mathcal{R}$ = ComputeRegions($\mathcal{A}, \mathcal{M}$)
18 $\rho$:=Multisection ($\mathcal{C}, \mathcal{R}, \mathrm{dist}_{L_1}$)
19 Set $\mathcal{C}_r := \{c \in \mathcal{C} | \rho(c) = r\}, \ \forall r \in \mathcal{R}$
20 **for** $r \in \mathcal{R}$ **do**
21     Legalization ($\mathcal{C}_r, A(r)$);
```

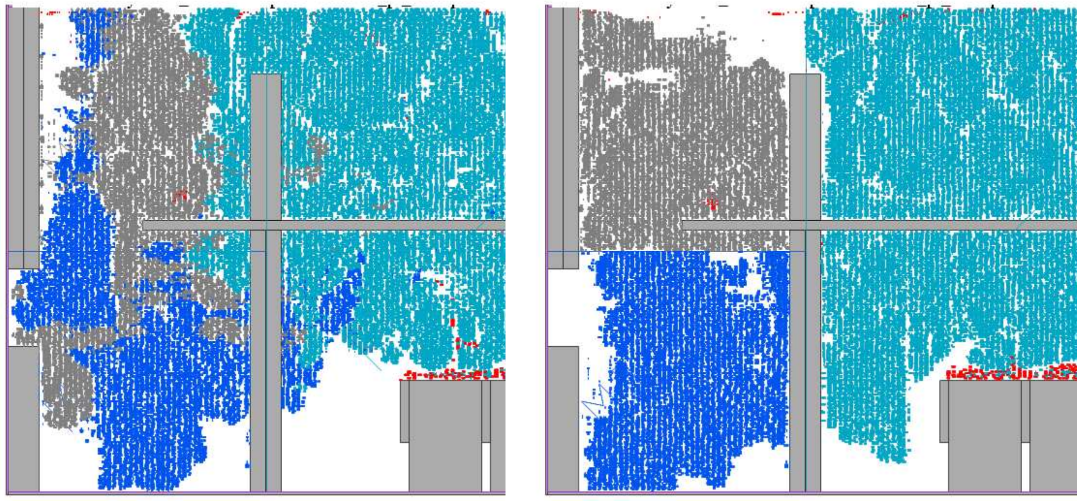


Figure 4.3: The chip Felix with 3 movebounds, cells colored w.r.t. the movebounds. On the left a placement without movebounds, on the right a placement respecting movebounds

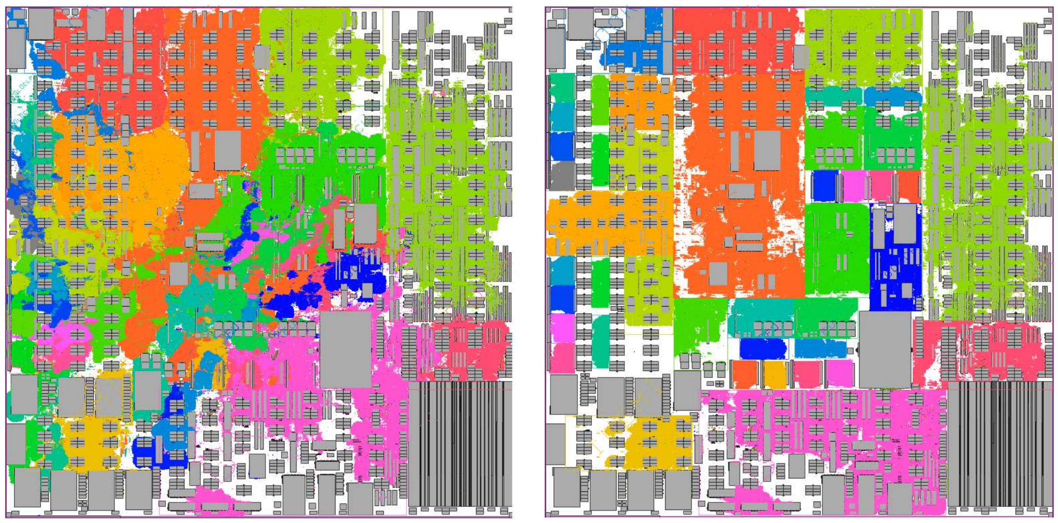


Figure 4.4: The chip Erhard with 43 movebounds, cells colored w.r.t. the movebounds. On the left a placement without movebounds, on the right a placement respecting movebounds

Chapter 5

New Global Partitioning Algorithm

Partitioning-based placement with MULTISECTION has proved its effectiveness, especially on large designs and in the presence of movebounds. However, a simple top-down partitioning scheme causes several problems. In this chapter we first discuss the essential problematic issues of subsequent partitioning and present a novel partitioning scheme unifying the advantages of the global view of the entire problem with excellent scalability and performance due to local realization steps.
The main ingredient of the new algorithm is a new MINCOSTFLOW model, where the supply is provided by the cell groups and the demand by windows' free space. Although MINCOSTFLOW approaches have been used in different contexts of VLSI placement, all of these methods suffer either from the algorithmic complexity of the MINCOSTFLOW problem itself, where the fastest known algorithms are too slow for many interesting cases if the number of nodes is in the millions, or from the poor flow realization afterward, or both. Even the fast MULTISECTION algorithm is limited to a small number of regions in practice.
Our method provides an inherent model to support movebound constraints and its computational complexity is very moderate: the MINCOSTFLOW instance itself does not depend on the number of cells, allowing extremely fast computations, especially on challenging designs. The realization, which takes connectivity into account, is then performed using a set of highly parallelizable local optimization steps.
We will show that this method also significantly improves the classic subsequent partitioning method in terms of runtime and wirelength. In this chapter we first motivate the idea, define the underlying MINCOSTFLOW model, then focus on realization. Finally, we present extensions to this algorithm, concluding with a brief summary of its properties.

5.1 Motivation

In partitioning based placement, as implemented in the previous version of BONNPLACE, the chip area is subsequently divided into smaller windows and the cells in some window are then assigned to its parts, where the assignment costs depend on the cells' locations. Such a strategy, however, has multiple disadvantages. First, it is extremely sensitive to slightly modified positions and often causes unnecessary movements. The overall strategy would be deficient, if revisions of the decisions could not be made (repartitioning). However, repartitioning is very time consuming and can take hours for a single level on large designs.

Partitioning decisions are currently made without explicit connectivity information. Even if additional connectivity information is provided indirectly (e.g. in form of ConstraintQP with center-of-gravity constraints), partitioning and induced movement over large distances is problematic if the design has large blockages. In this case, the cells are forced towards free area through ConstraintQP, and are often moved across large blockages. As the following partitioning decision is only based on locations, cell groups risk being torn into parts and these parts being forced to stay on opposite blockage sides, leading to inferior placements.
Attempts to cope with the problem by considering more and finer windows during the partitioning step (*partitioning with look-ahead*) and setting the costs of moving a cell to a window to be the distance from the cell to the *next free location* of the window did not yield the desired improvements as reported by Brenner [2005]. One reason for this is the fact that in principle partitioning cannot be performed independently and that the movement costs do not necessarily correlate with placement quality as the placement problem more closely resembles a *quadratic* assignment problem than a simple one. Higher movement costs are better if the cell groups will not be torn into pieces, hence it would be probably better to bring connectivity into this model together with movement.

Third, the old partitioning based placement uses the invariant that, once a feasible (regardless of rounding) partitioning decision in a window has been taken, subsequent partitioning decisions are also feasible. This is no longer true if the cell sizes increase during placement (e.g. during inflation from congestion driven placement) or the capacity of the windows decreases (e.g. after fixation of some macro blocks whose area may block some of the window capacity the macro was not assigned to before having been fixed). Such effects were freely ignored in the past by simply providing additional capacities, even if they exceeded 100% of a window's free area. It was then left to the designer to rerun the placement with fixed macros or increased cell sizes or to accept higher running time and modest results during legalization. In case of instances with movebounds, these problems state a serious challenge as the effects do not affect the entire design only but each single hard movebound. So they should not be neglected any more.

To face the first two points, Brenner [2005] proposed the concept of iterative partitioning (see Section 3.4.3). This algorithm helps to reduce the runtime by saving several repartitioning steps in general, but there is no guarantee on the number of iterations, a hard runtime bound depending on the number of windows was set. In addition, the implemented version is not parallelizable very well because the access to the global heap structure which decides over the priority of the windows causes concurrency problems. Both sometimes lead to higher runtimes than the extended use of repartitioning would need. A geometric partitioning of the chip and independent application of iterative partitioning on each of the parts with

safety margins does not necessarily help if overloads appear in one quarter of the chip, causing high CPU idle times.
Beyond the runtime problems, the profoundly bigger issue is that a feasible partitioning is not guaranteed (even without hard runtime bounds) and is indeed often not achieved in practice. But even the subsequent partitioning with many repartitioning steps offers many opportunities to round the Multisection result, and the rounding errors can sum up to significant violations. (see Fig. 5.1).
Additionally, the current implementation of the iterative partitioning does not take local densities into account which may differ considerably from the overall density target.

Apart from this, a placement with movebounds cannot be solved by this concept with a single capacity key at all. A partitioning step with movebounds has to decide not only about the amount of cell area which has to leave a window but also about the *kind* of area which has to move. This decision cannot be taken if only *local* information is available (see Fig. 5.2).
All the described problems lead to the question of how intensive repartitioning can be avoided, providing at the same time a fast, reliable, density and movebound respecting method of partitioning.

5.2 Flow-based Global Partitioning

Analogously to the concept of recursive partitioning with movebounds (see Section 4.3) for a set of cells $\mathcal{C}$ with movebounds $\mathcal{M}$ and a set of windows $\mathcal{W}$ induced by some regular grid Γ, we ask for a feasible partitioning $\rho : \mathcal{C} \to \mathcal{W}$ (cf. Definition 4.3.5).
The new partitioning method also starts with a potentially illegal partitioning: the cells in $\mathcal{C}$ first remain in their positions and are then assigned to windows $\mathcal{W}$ according to their locations. Then the directions in which cell groups are moved are computed *a priori* via a MinCostFlow algorithm. Finally, once the directions are computed, the realization is done by a sequence of *local repartitioning steps* which allow connectivity and movement to be taken into account at the same time. We first present the underlying network:

5.2.1 Computing Cirections

The Nodes

In our algorithm there are three types of nodes: cell groups, transit nodes and region nodes. For each window $w \in \mathcal{W}$ we consider several node sets. For the set $\mathcal{C}_w$ of cells in w, we cluster the members of $\mathcal{C}_w$ with respect to their movebounds and obtain a partition:

$$\mathcal{C}_w := \dot{\bigcup}_{M \in \mathcal{M}} \mathcal{C}_{Mw}.$$

For each such set we obtain a node C_{Mw} (we omit empty sets).

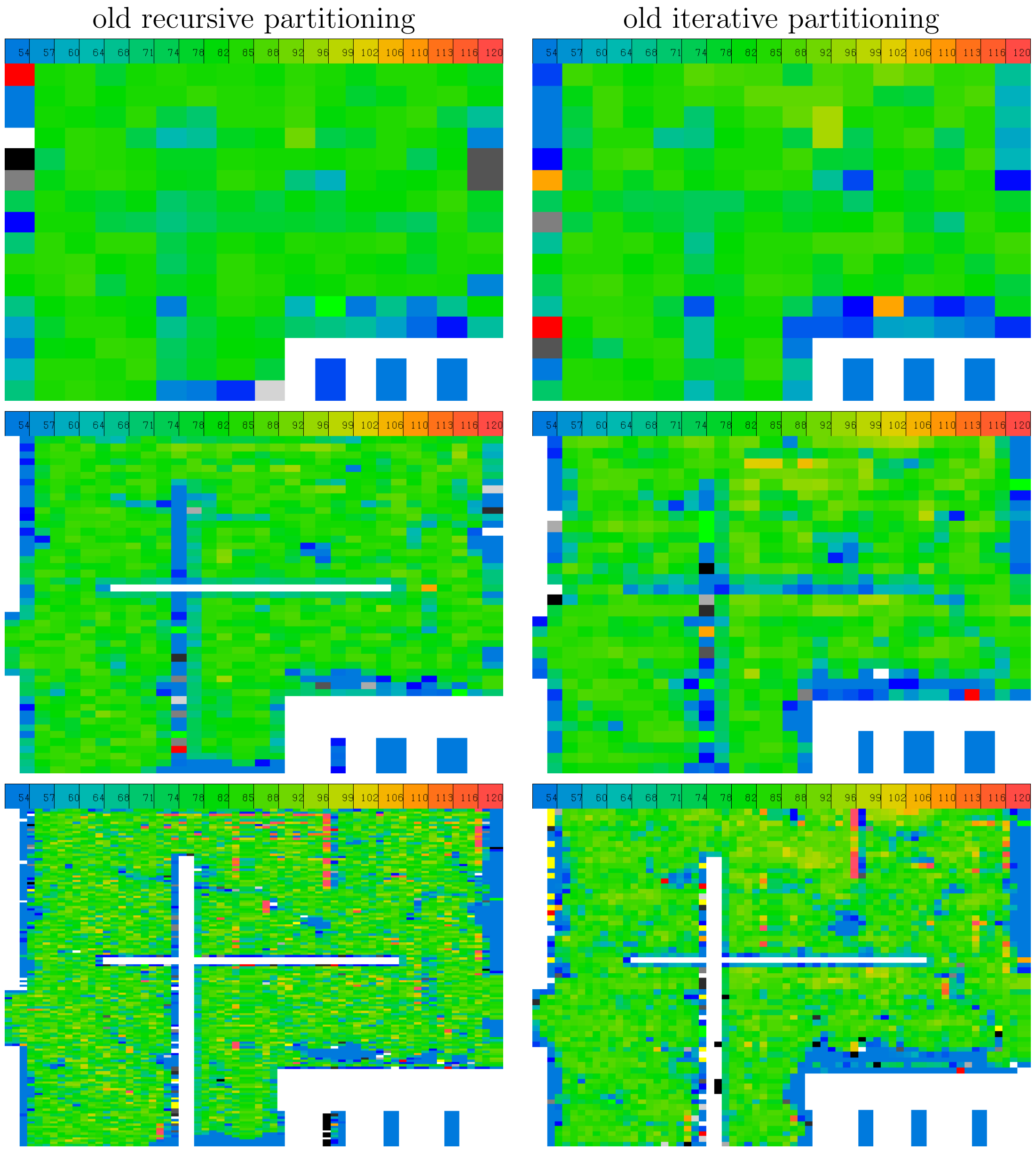


Figure 5.1: Density problems in old BonnPlace: The pictures show relative capacities of cells in windows where the overall target density was 88%. On the left from top to bottom, relative densities in windows of levels 4, 5 and 6 are shown, where the partitioning was done by recursive partitioning with up to 5 repartitioning steps. On the right from top to bottom, the relative densities are shown for the same levels, but where iterative partitioning was applied instead.
Already in level 4, there are windows with density violations of about 10% in the case of iterative partitioning. The problems drastically increase in level 6 for both runs, exceeding some capacities by more than 30%.

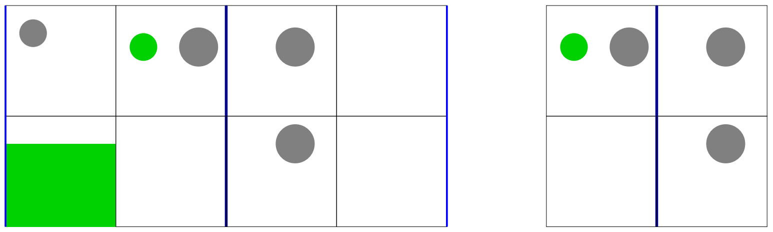


Figure 5.2: Iterative partitioning fails on instances with movebounds: Consider an instance with two coarse windows (blue), each subdivided into four finer windows (black). Let there be a hard movebound (green). Assume that each finer window can accommodate at most one of the cell groups (colored dots). Although a feasible partitioning into the coarse windows is given (left), the iterative partitioning step would have chosen the center window first (right), in which a feasible partitioning is not possible.

For each movebound $M \in \mathcal{M}$ and each window $w \in \mathcal{W}$, let $\mathcal{Z}_{Mw}$ be a set of *transit nodes* located in the north, south, east and west center of the window border:

$$\mathcal{Z}_{Mw} := \{z^N_{Mw}, z^S_{Mw}, z^E_{Mw}, z^W_{Mw}\}$$

We write $\mathcal{Z}_M := \bigcup_{w \in \mathcal{W}} \mathcal{Z}_{Mw}$ and $\mathcal{Z} := \bigcup_{M \in \mathcal{M}} \mathcal{Z}_M$.

In fact, we will see below (cf. Remark 5.2.3) that we do not need nodes for each movebound and each window: we will be able to skip, all sets $\mathcal{Z}_{Mw}$. if w does not intersect $A(M)$ bounding box. To simplify the consideration we do it tacitly from now on. Finally, we add a node for each region $r \in \mathcal{R}_w$.
Note that cell group and transit nodes belong to a unique movebound, but regions may be covered by multiple movebounds.

The Inner Edges

The first arc group consists of inner-window edges. The cell group nodes are sources, the regions sinks. Transit nodes allow through traffic. We thus obtain four different arc sets per window and per movebound.
For internal window transit, we want to allow traffic in either direction, and hence connect pairs of transit nodes of the same movebound by a pair of oppositely oriented edges:

$$E^{zz}_{Mw} := \{(z^x_{Mw}, z^y_{Mw}) | x, y \in \{N, W, S, E\}, x \neq y\}. \tag{5.1}$$

In order to connect cell group nodes to their local transit nodes, we define

$$E^{Cz}_{Mw} := \{(C_{Mw}, z^x_{Mw}) | x \in \{N, W, S, E\}\}. \tag{5.2}$$

We connect, moreover, each cell group node C_{Mw} to a region node $r \in \mathcal{R}_w$ wherever M covers r:

$$E^{Cr}_{Mw} := \{(C_{Mw}, r) | r \in \mathcal{R}_w, M \text{ covers } r\}. \tag{5.3}$$

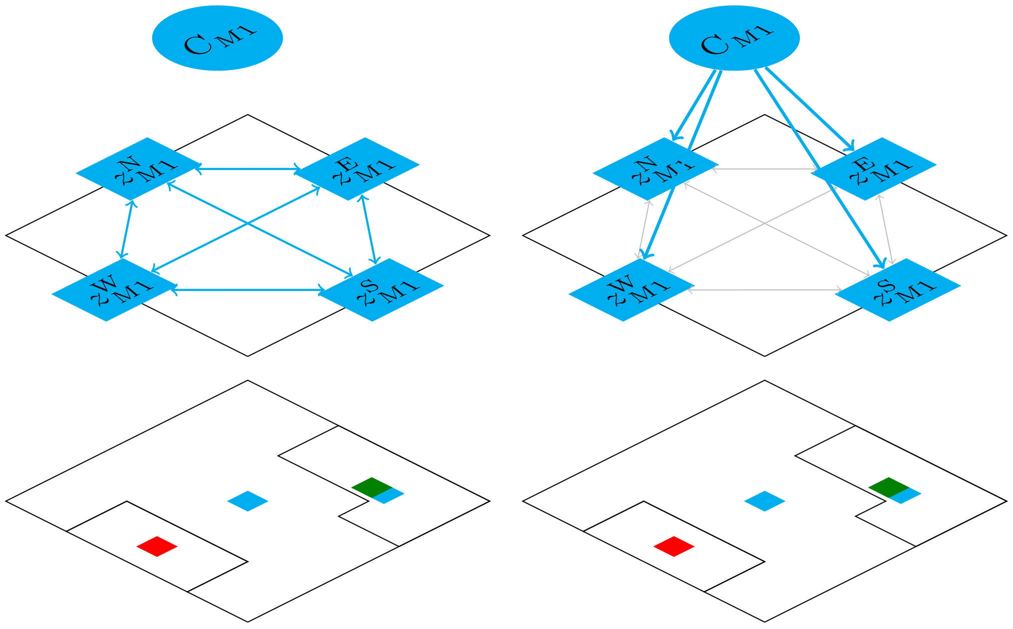


Figure 5.3: Example in window 1: A cell cluster node C_{M1} of movebound M, the four transit nodes and region nodes for this window. In addition, the inner-window transit arcs E^{zz}_{M1} for window 1 and movebound M are depicted (left). On the right the arcs E^{Cz}_{M1} connecting a cell cluster node C_{M1} to the transit nodes in $\mathcal{Z}_{M1}$ of this movebound are depicted.

Similarly, each transit node z^x_{Mw} is connected to a region node $r \in \mathcal{R}_w$ wherever M covers r:

$$E^{zr}_{Mw} := \{(z^x_{Mw}, r) | r \in \mathcal{R}_w, x \in \{N, W, S, E\}, M \text{ covers } r\}. \tag{5.4}$$

An example for the inner windows edges and nodes for one movebound and window with three regions is shown in Figures 5.3 and 5.4.

The External Edges

For two neighboring windows $v, w \in \mathcal{W}$ and a movebound M we connect the corresponding modules (if both exist) by connections between their matching neighboring transit nodes:

$$E^{\text{ext}}_{v,w,M} := \begin{cases} \{(z^S_{Mv}, z^N_{Mw}), (z^N_{Mw}, z^S_{Mv})\} & \text{if } w \text{ is lower neighbor of } v \\ \{(z^N_{Mv}, z^S_{Mw}), (z^S_{Mw}, z^N_{Mv})\} & \text{if } v \text{ is lower neighbor of } w \\ \{(z^W_{Mv}, z^E_{Mw}), (z^E_{Mw}, z^W_{Mv})\} & \text{if } v \text{ is right neighbor of } w \\ \{(z^E_{Mv}, z^W_{Mw}), (z^W_{Mw}, z^E_{Mv})\} & \text{if } w \text{ is right neighbor of } v \\ \emptyset & \text{otherwise.} \end{cases}$$

See also Figure 5.5.

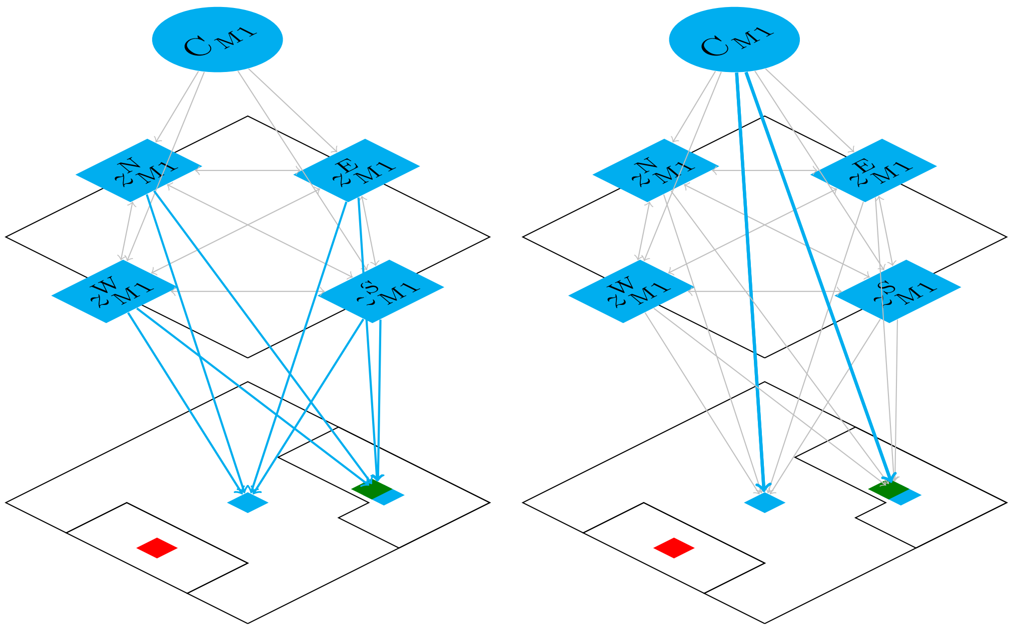


Figure 5.4: Completing the example in window 1. A cell cluster node C_{M1} of movebound M, the four transit nodes and region nodes for this window. The edges E_{M1}^{zr} connecting the transit nodes of movebound M to region nodes covering the movebound M are shown left. Finally, the arcs E_{M1}^{Cr} connecting a cell cluster node C_{M1} to the region nodes covering the movebound M are given (right).

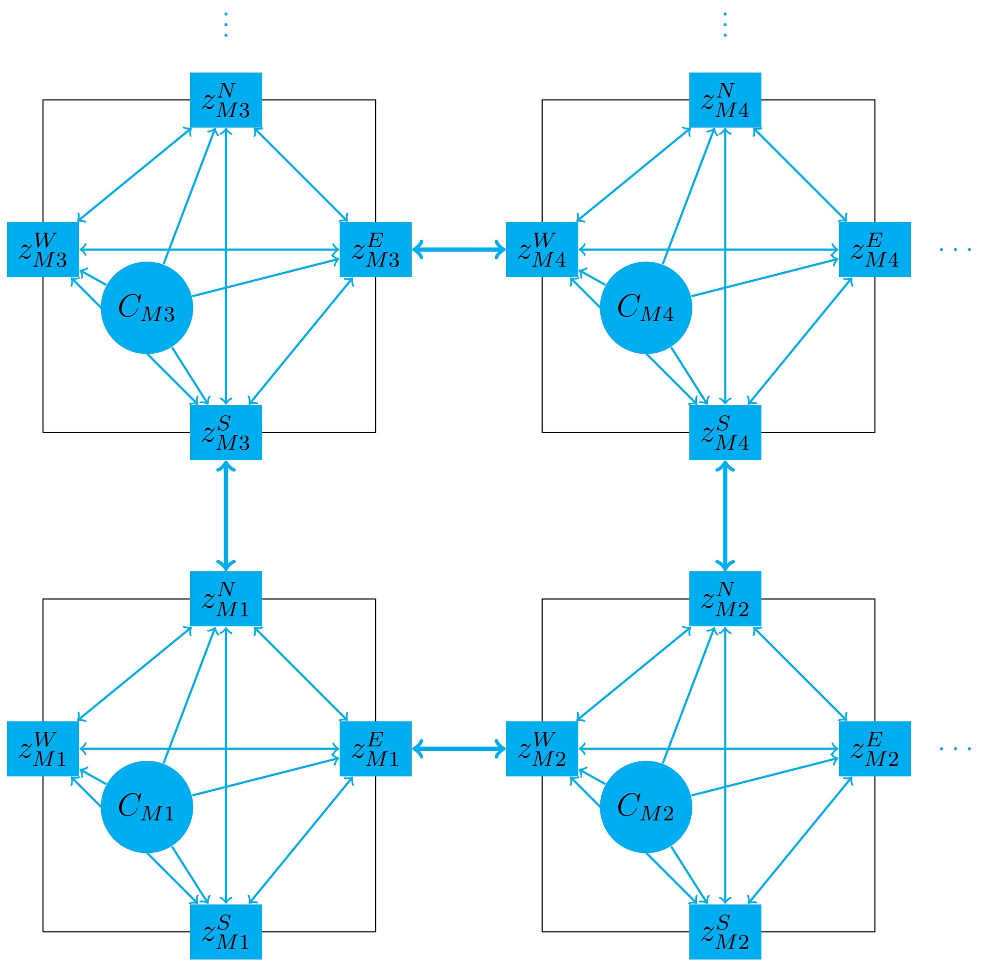


Figure 5.5: Example containing external edges: Subgraphs of different windows for the movebound M connected by edges between transit nodes. Region nodes are omitted here.

The costs and b-values

The overall cost structure in the MinCostFlow instance is based on two components: the movement cost to penalize cell movement and the soft movebound violation penalties. Here, as cells are hidden within cell group nodes, we must work with an approximation to express what the movement cost between cells and regions/transit nodes is.
We propose an embedding of the nodes in the plane, where cell group nodes get the position of the center of gravity (COG) of the members and the transit nodes, as mentioned, are set to the centers of the north, west, south and east cutline. The region nodes get the positions of the center of gravity of their free areas.

With this embedding, we can express movement costs as distances in the plane. The traffic between different windows can be done almost for free, thus we set for each external edge e with $e = (z^x_{Mv}, z^y_{Mw}) \in E^{\text{ext}}_{v,w,M}$ for some windows $v, w \in \mathcal{W}$ and $M \in \mathcal{M}$

$$\text{cost}(e) := \varepsilon \text{ for some small constant } \varepsilon > 0. \tag{5.5}$$

The movement is not completely for free to prevent cycles. The movement across the remaining arcs correlates to the covered distance, thus
for $e = (C_{Mw}, r) \in E^{Cr}_{Mw}$ we set

$$\text{cost}(e) := \text{dist}(\text{COG}(C_{Mw}), \text{COG}(r)),$$

for $e = (C_{Mw}, z) \in E^{Cz}_{Mw}$ we set

$$\text{cost}(e) := \text{dist}(\text{COG}(C_{Mw}), z) + \varepsilon,$$

for $e = (z, r) \in E^{zr}_{Mw}$ we set

$$\text{cost}(e) := \text{dist}(z, \text{COG}(r)),$$

and finally for $e = (z^x_{Mw}, z^y_{Mw}) \in E^{zz}_{Mw}$, representing the traffic within one window, we set

$$\text{cost}(e) := dist(z^x_{Mw}, z^y_{Mw}).$$

Note here that $\text{cost}((z, r)) > 0$ for each $z \in \mathcal{Z}$ and each $r \in \mathcal{R}$, as regions with empty interior do not show up.
At last, we address the b-values for this network which come in a very natural manner: we have some supply (cell groups and their sizes) which has to be distributed to regions. Thus we define:

$$b(C_{Mw}) := \sum_{c \in C_{Mw}} \text{size}(c) \qquad \forall M \in \mathcal{M},\ \forall w \in \mathcal{W},$$

$$b(r) = -\text{capa}(r) \qquad \forall w \in \mathcal{W},\ \forall r \in \mathcal{R}_w.$$

All transit vertices are pure transit nodes, hence $b(v) \equiv 0$ elsewhere. The reader should note that there are no capacity constraints on edges in this graph at all.

Soft movebounds

Again, we consider soft movebounds as hard ones for the movebound cells with possibly enlarged area as described in Section 4.3.3. For an exclusive soft movebound N however we want to allow other cells to enter N's area, but penalize such assignments. For this purpose we add a penalty constant to all edges entering a region covered by N and leaving some non-N transit or cell group node.

MinCostFlow instance

Merging all nodes and edges yields the uncapacitated MinCostFlow instance

$$G = (V(G), E(G), b, \text{cost}) \tag{5.6}$$

$$\begin{aligned} V(G) &:= \bigcup_{w \in W} \Big(\bigcup_{M \in M} (C_{Mw} \cup \mathcal{Z}_{Mw}) \cup \mathcal{R}_w \Big) \\ E(G) &:= \bigcup_{w \in W} \Big(\bigcup_{M \in M} E^{Cz}_{Mw} \cup E^{zz}_{Mw} \cup E^{Cr}_{Mw} \cup E^{zr}_{Mw} \Big) \cup \bigcup_{v,w \in W} \bigcup_{M \in M} E^{\text{ext}}_{v,w,M}. \end{aligned} \tag{5.7}$$

In the following we will switch the interpretation between regions and region nodes as well as between cell groups and cell group nodes in G.

5.2.2 Properties of the MinCostFlow Instance

Let $\mathcal{W}$ be a set of windows and $\mathcal{M}$ a set of movebounds. As movebounds are allowed to overlap or the movebound shapes can be non-path connected, in worst case the instance consists of $|\mathcal{M}| \cdot |\mathcal{W}|$ subgraphs

$$G_{Mw} := \big(C_{Mw} \cup \mathcal{Z}_{Mw} \cup \mathcal{R}_w, E^{Cz}_{Mw} \cup E^{zz}_{Mw} \cup E^{zr}_{Mw} \cup E^{Cr}_{Mw} \big) \qquad \forall M \in \mathcal{M}, \forall w \in \mathcal{W}, \tag{5.8}$$

connected by additional $\mathcal{O}(|\mathcal{M}| \cdot |\mathcal{W}|)$ external edges. In practical applications however, we do observe that the number of edges in the entire graph is linear in the number of nodes (see also Table 8.2), and the number of nodes is determined by the number of regions in the early levels and by the number of windows in the late ones. The number of cells does not contribute to the complexity. Assuming $|E(G)| = \mathcal{O}(|V(G|))$ we get:

Lemma 5.2.1 The MinCostFlow problem is solvable in $\mathcal{O}(|V(G)|^2 \log^2(|V(G)|))$ time.

Proof: Direct application of the algorithm of Orlin [1993]. □

Although the above MinCostFlow instance has local structure, we do not lose any feasibility properties:

Proposition 5.2.2 The partitioning instance is feasible (cf. Def. 4.3.5) if and only if the MinCostFlow instance $(G, \text{capa}, \text{cost}, b)$ is feasible.

Proof: To see the equivalence observe that each cell $c \in \mathcal{C}$ is contained in some cell group $C_{\mu(c)w}$ and for each region r covered by $\mu(c)$ there is an uncapacitated path, say P, starting with $C_{\mu(c)w}$ and ending with r. In addition, for each movebound $M \in \mathcal{M}$, the total supply is exactly the total size of associated cells:

$$\sum_{w \in \mathcal{W}} b(C_{Mw}) = \sum_{c \in \mathcal{C}:\mu(c)=M} \text{size}(c).$$

Define the MaxFlow instance:

$$\begin{aligned} \big(&\{s,t\} \dot\cup \{C_{Mw} | M \in \mathcal{M}, w \in \mathcal{W}\} \cup \mathcal{R}, \\ &\{(s,C)\} \cup \{e = (C,r) | \exists P = (C,..,r)\} \cup \{(r,t) | r \in \mathcal{R}\}, u\big) \end{aligned} \tag{5.9}$$

with $C = C_{Mw}$ for some $w \in \mathcal{W}, M \in \mathcal{M}$ and capacities u defined as $u(s,C) := b(C)$ for each cell group node C, $u(r,t) := \text{capa}(r)$ for each region r, and $u \equiv \infty$ elsewhere. Now the MaxFlow instance from Lemma 4.3.3 is equivalent to the instance (5.9). □

It is almost straight-forward to see that it is no restriction to have transit nodes only in windows which intersect the bounding box of their hard movebound area, if all the cells of this movebound are located within these windows.

Remark 5.2.3 If for any movebound $M \in \mathcal{M}$, some solution of the MinCostFlow uses a node in some window which does not intersect $A(M)'s$ bounding box, then the solution is not optimal. If some flow would leave $A(M)$'s bounding box, it would enter $A(M)$ somewhere again. Replacing such a path with another path inside $A(M)$'s bounding box is strictly cheaper. However, the restriction to the windows which intersect the movebound itself is not sufficient as non-path connected areas could appear.

Fix a min-cost flow f in $(G, \text{capa}, \text{cost}, b)$. We are now going to examine some properties of the flow-carrying subgraph G_f of G.

Lemma 5.2.4 The flow-carrying subgraph G_f of G is acyclic, thus G_f has a topological order.

Proof: This is almost obvious: all costs are non-negative. Any cycle could only appear under participation of some internal edge (z^x_{Mw}, z^y_{Mw}) for some $w \in \mathcal{W}$ and $M \in \mathcal{M}$. But $\text{cost}((z^x_{Mw}, z^y_{Mw})) = \text{dist}(z^x_{Mw}, z^y_{Mw}) > 0$. This fact would contradict the optimality of f. □

Another interesting observation is that, although external path lengths can be of length $\sqrt{|\mathcal{W}|}$, the interior ones are short:

Lemma 5.2.5 The maximum length of a flow-carrying sub-path within a window is one.

Proof: Assume there is a path of length two in a window $w \in \mathcal{W}$. The possible paths within a window are $C - z - r$, $C - z - z'$, $z - z' - r$ or a $z - z' - z''$, where C is some cell group node, z, z', z'' some transit nodes and r some region node in w. Consider the first case of an $C - z - r$ path. Then by the triangle-inequality:

$$\begin{aligned} \text{cost}((C, r)) = \text{dist}((\text{COG}(C), \text{COG}(r)) \leq \text{dist}(\text{COG}(C), z) + \text{dist}(z, \text{COG}(r)) \\ < \text{cost}((C, z)) + \text{cost}((z, r)) \end{aligned} \tag{5.10}$$

By a similar argument, the other three cases can be omitted. □

It follows also directly:

Corollary 5.2.6 Let $e = (z^x_{Mv}, z^y_{Mw})$ be an external flow-carrying edge. Then e is the only flow-carrying edge entering z^y_{Mw}.

Proof: By Lemma 5.2.5 and Lemma 5.2.4 there is no interior flow-carrying edge entering z^y_{Mw}. As z^y_{Mw} has exactly one external neighbor, the connecting edge is the only one which carries flow. □

5.2.3 Realization

Definition 5.2.7 The *realization* of the flow at some external edge $e = (z^x_{Mv}, z^y_{Mw})$ means to choose a set C of cells with total size f_e which belong to movebound M out of v, assign them to a node in w associated to movebound M and remove e from G_f.

Of course we have to proceed in topological order to make sure that for each external flow-carrying edge $e = (z^x_{Mv}, z^y_{Mw})$ there are enough cells of movebound M in window v to be pushed towards w. Now, consider some oracle which realizes external flow-carrying edges one by one. The next statement tells us that it is indeed sufficient to stop if flow on all external edges has been realized.

Theorem 5.2.8 Given a solution of the MinCostFlow instance (5.6), proceed realizing external edges in topological order and delete the external edges after realization. When there is no external flow-carrying edge, the partitioning is feasible.

Proof: For a given window $w \in \mathcal{W}$ and each subset of movebounds $\mathcal{M}' \subseteq \mathcal{M}$ we have to check the condition (4.7). Fix w and $\mathcal{M}'$. Let $\mathcal{C}^{\text{old}}_{Mw} := \{c \in \mathcal{C}_w | \mu(c) \in M\}$ be the set of all cells with movebound M in w before any external edges incident to nodes in w are realized, and let $\mathcal{C}^{\text{new}}_{Mw} := \{c \in \mathcal{C}_w | \mu(c) = M\}$ be such a set of cells after that. Set $E^-_M := \{e \in \delta^-(\bigcup_{x \in \{N,W,S,E\}} \mathcal{Z}^x_{Mw}) | e \text{ has been realized}\}$, and $E^+_M := \{e \in \delta^+(\bigcup_{x \in \{N,W,S,E\}} \mathcal{Z}^x_{Mw}) | e \text{ has been realized}\}$.
Note that:

$$\sum_{c \in \mathcal{C}^{\text{new}}_{Mw}} \text{size}(c) = \sum_{c \in \mathcal{C}^{\text{old}}_{Mw}} \text{size}(c) + \sum_{e \in E^-_M} f(e) - \sum_{e \in E^+_M} f(e) \tag{5.11}$$

and

$$\sum_{c \in \mathcal{C}^{\text{old}}_{Mw}} \text{size}(c) = b(C_{Mw}). \tag{5.12}$$

Now,

$$\begin{aligned}
\sum_{M\in\mathcal{M}'}\sum_{c\in\mathcal{C}^{\text{new}}_{Mw}} \text{size}(c) &= \sum_{M\in\mathcal{M}'}\Big(\sum_{e\in E^-_M} f_e + \sum_{c\in\mathcal{C}^{\text{old}}_{Mw}} \text{size}(c) - \sum_{e\in E^+_M} f_e\Big)\\
&= \sum_{M\in\mathcal{M}'}\Big(\sum_{z\in\mathcal{Z}_{Mw}}\sum_{e=(x,z):x\notin\mathcal{Z}_{Mw}} f_e + \sum_{c\in\mathcal{C}^{\text{old}}_{Mw}} \text{size}(c) - \sum_{z\in\mathcal{Z}_{Mw}}\sum_{e=(z,y):y\notin\mathcal{Z}_{Mw}} f_e\Big)\\
&= \sum_{M\in\mathcal{M}'}\Big(\sum_{z\in\mathcal{Z}_{Mw}}\Big[\sum_{y\in\mathcal{Z}_{Mw}} f_{(z,y)} + \sum_{r\in\mathcal{R}_w} f_{(z,r)}\Big]\\
&\qquad + \sum_{c\in\mathcal{C}^{\text{old}}_{Mw}} \text{size}(c) - \sum_{z\in\mathcal{Z}_{Mw}}\sum_{e=(z,y):y\notin\mathcal{Z}_{Mw}} f_e\Big)\\
&= \sum_{M\in\mathcal{M}'}\Big(\sum_{z\in\mathcal{Z}_{Mw}}\Big[\sum_{y\in\mathcal{Z}_{Mw}} f_{(z,y)} + \sum_{r\in\mathcal{R}_w} f_{(z,r)}\Big]\\
&\qquad + \sum_{e\in E^{Cr}_{Mw}} f_e + \sum_{e\in E^{Cz}_{Mw}} f_e - \sum_{z\in\mathcal{Z}_{Mw}}\sum_{y\notin\mathcal{Z}_{Mw}} f_{(z,y)}\Big)\\
&= \sum_{M\in\mathcal{M}'}\Big(\sum_{z\in\mathcal{Z}_{Mw}}\sum_{r\in\mathcal{R}_w} f_{(z,r)} + \sum_{e\in E^{Cr}_{Mw}} f_e\\
&\qquad + \sum_{z\in\mathcal{Z}_{Mw}}\sum_{y\in\mathcal{Z}_{Mw}} f_{(z,y)} + \sum_{e\in E^{Cz}_{Mw}} f_e - \sum_{z\in\mathcal{Z}_{Mw}}\sum_{y\notin\mathcal{Z}_{Mw}} f_{(z,y)}\Big)\\
&= \sum_{M\in\mathcal{M}'}\Big(\sum_{z\in\mathcal{Z}_{Mw}}\sum_{r\in\mathcal{R}_w} f_{(z,r)} + \sum_{e\in E^{Cr}_{Mw}} f_e\Big)\\
&\le \sum_{r\in\mathcal{R}_w:r\text{ covered by}M\in\mathcal{M}'} \text{capa}(r).
\end{aligned} \tag{5.13}$$

□

Theorem 5.2.8 suggests that during realization we can indeed focus solely on external flow edges only. These arcs state just a fraction of all flow-carrying edges. Moreover, Lemma 5.2.5 says that operations within a window can be performed without flow graph scanning, by considering direct neighbors, i.e. paths of length 1.

From now on, consider flow-carrying arcs in G_f only. Additionally, let us ignore rounding effects for the time being and treat all cells as arbitrarily granular objects. During realization we will proceed edgewise in topological order and delete realized edges from the flow-carrying graph G_f.

Until now, we have computed the directions, the amount and the movebound, towards which the overloads will be pushed. Now we have to decide *which* cells will be moved. This decision requires partitioning decisions, except for the very simple case where $e = (z^x_{Mv}, z^y_{Mw})$ has to be realized and z^x_{Mv} is the only node in window v with non-zero in-flow. Here, simply

all cells in v with movebound M have to leave v towards w. In the general case, instead of performing simple MULTISECTION among cells currently stored in v and nodes in the window v's boundary, two novel ideas come into play:
First, we will not only consider the cells in v for this MULTISECTION purpose, but all cells *in a coarser window, say* $W \subset \mathcal{W}$*, which covers* v and some of its neighbors. Second, the partitioning decision will not be performed with movement costs as they are, but a local QP/CONSTRAINTQP will be computed for the cells in W to obtain more connectivity information.
Both ideas provide another significant difference to the old partitioning realization method: previously, the decision for cell assignment was based on movement from the cells' location to the regions only and was taken for each coarse window independently. Now, the presented procedure has a kind of built-in repartitioning: every time a partitioning decision has to be taken, it considers local connectivity and inertia's of the cells.
It is important to see that, if the partitioning comes with movebounds or if the coarse realization window W covering a window $w \in \mathcal{W}$ consists of more than two windows, we may be forced to perform partitioning decisions several times in w. This might happen, because in these cases the graph obtained from G_f by contraction of all nodes at the same windows is not acyclic, although G_f is. In this case we will allow the reassignment of each cell within the window w each time we perform some realization step within some coarser window including w.

Let us fade out integrality requirements in partitioning for a while and focus on fractional realizations.

In order to postpone partitioning decisions which can't be taken yet, we need a buffering mechanism for capacity. To this end introduce the function $\zeta : \mathcal{Z} \cup \mathcal{R} \to \mathbb{R}_+$, indicating the flow *excess* at each node $x \in \mathcal{Z} \cup \mathcal{R}$ at some realization stage.

$$\zeta(x) := \sum_{e \in \delta^-_{G_f}(x), e \text{ realized}} f_e - \sum_{e \in \delta^+_{G_f}(x), e \text{ realized}} f_e \qquad x \in \mathcal{Z} \cup \mathcal{R} \tag{5.14}$$

Note that, due to the topological order, the amount $\zeta(x)$ for each $x \in \mathcal{Z} \cup \mathcal{R}$ will not decrease until the last flow-carrying edge entering x is realized. As each cell group node does not have any predecessors, we can initially set

$$\zeta(x) = \sum_{e \in \delta^-_{G_f}(x) : e = (C_{Mv}, x)} f_e, \ \forall x \in \mathcal{Z} \cup \mathcal{R} \tag{5.15}$$

and remove all flow-carrying edges which start at some cell group node. We refer to this step as PUSHINITIALFLOW. We are now able to present the realization algorithm (Algorithm 5).

During realization, we will maintain the invariant that for each window $w \in \mathcal{W}$ and for each subset of movebounds $\mathcal{M}' \subseteq \mathcal{M}$ the total area of cells in w with movebounds in $\mathcal{M}'$

Algorithm 5: FlowBasedPartitioning $(\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{M}, \mu)$

```
1  ProjectCellsToMovebounds $(\mathcal{C}, \mathcal{M}, \mu)$
2  $\rho :=$ AssignCellsToWins $(\mathcal{C}, \mathcal{W})$
3  $G_f :=$ ComputeAndSolveMinCostFlow $(\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{M})$
4  $\zeta :=$ PushInitialFlow $(G_f)$
5  for $e = (z^x_{Mv}, z^y_{Mv}) \in E^{ext}(G_f)$ in topological order do
6      Choose $W \subseteq \mathcal{W}$ with $W \supseteq \{v, w\}$
7      $\rho_{|W} :=$ RealizeEdge($G_f, \zeta, W, e$)
```

Procedure RealizeEdge(G_f, ζ, W, e)

```
1  Let $V_W$ be the set of nodes in $V(G_f)$ which belong to the windows in $W$.
2  PushFlow($G_f, e, \zeta$)
3  LocalQP($\{c \in \mathcal{C} | \rho(c) \in W\}, W$)
4  LocalCenterOfGravityQP($\{c \in \mathcal{C} | \rho(c) \in W\}, W$)
5  $\mathcal{Q} := \emptyset$
6  for $z_{Mv} \in V_W \cap \mathcal{Z}$ do
7      interpret $z_{Mv}$ as region with:
8      capacity $\zeta(z_{Mv})$, $A(z_{Mv}) :=$ window of node $x$ with $e = (z_{Mv}, x)$ and movebound set
       $\{M\}$.
9      $\mathcal{Q} := \mathcal{Q} \cup \{z_{Mv}\}$
10 $\rho_{|W} :=$ Multisection $(\{c \in \mathcal{C} | \rho(c) \in W\}, \{x \in \mathcal{Q} \cup \mathcal{R}_W | \zeta(x) > 0\})$
```

Procedure PushFlow(G_f, e, ζ)

```
1  Assume $e = (x, y)$.
2  Set $E_{\text{realize}} := \delta^-_{G_f(x)} \cup \{e\} \cup \delta^+_{G_f}(y)$
3  for $e' = (a, b) \in E_{realize}$ in topological order do
4      $\zeta(a) := \zeta(a) - f_{e'}$
5      $\zeta(b) := \zeta(b) + f_{e'}$
6      if $e' \in \delta^-(r)$, $r \in \mathcal{R}$, $e'$ last edge entering $r$ then
7          $\zeta(r) := \max\{\zeta(r), \text{capa}(r)\}$
8      $E(G_f) := E(G_f) - e'$
```

does not exceed the total capacity provided by the buffer function ζ for transit nodes and regions covered by movebounds in $\mathcal{M}'$. This can be seen as "a feasible partitioning" in terms of Definition 4.3.5, where additional region capacities are provided by ζ at transit nodes.

Proposition 5.2.9 (Realization invariant) Before and after calling RealizeEdge, we have for each $w \in \mathcal{W}$ and each $\mathcal{M}' \subseteq \mathcal{M}$:

$$\sum_{c \in \mathcal{C}_w : \mu(c) \in \mathcal{M}'} \text{size}(c) \leq \sum_{r \in \mathcal{R}_w :\ r \text{ covered by } \mathcal{M}'} \zeta(r) + \sum_{M \in \mathcal{M}'} \sum_{x \in \{N,E,S,W\}} \zeta(z^x_{Mw}). \tag{5.16}$$

In particular, the (fractional) Multisection in w involving the regions obtained from transit nodes z^x_{Mw}, $x \in \{N, W, S, E\}, w \in \mathcal{W}$ and $M \in \mathcal{M}'$ is always feasible and RealizeEdge works correctly.

Proof: By induction on $|E^{ext}(G_f)|$. Initially, (5.16) holds by (5.15). Assume (5.16) holds after having realized $n-1$ edges. Now consider the n-th edge, say $e = (z^x_{Mv}, z^y_{Mw})$. W.l.o.g. we may assume that the realization of e is done in $W = \{v, w\}$. Now, by induction hypothesis we have (5.16) for any movebound subset $\mathcal{M}'_v$ in the window v and for any movebound subset $\mathcal{M}'_w$ in the window w. Thus, it also holds for any subset of the union $\mathcal{M}' \subset \mathcal{M}'_v \cup \mathcal{M}'_w$. Recall that this is precisely the feasibility condition (4.7) for the partitioning with movebounds if ζ represents the capacity of the right hand side. By (5.14), PushFlow swaps the total amount of f_e capacity units from transit nodes with movebound M in window w to transit nodes and regions in window w. The total amount does not decrease and is only increased for regions in w. Multisection in RealizeEdge hence sees at least the same total capacity for each movebound subset as before, and is hence feasible by the induction hypothesis and (4.7). In addition, as f_e units of capacity were moved from window v to w, so by correctness of Multisection, (5.16) holds for v and w after partitioning. $\square$

Beyond the issue of capacity, the question remains: how can (temporary) regions be made out of transit nodes. Each transit node $z_{Mw} \in \mathcal{Z}_{Mw}$ is associated with a movebound, say M, and we have to enforce that only cells with movebound M enter z_{Mw} for transit purposes. Thus, $\{M\}$ is the set of movebounds covered by z_{Mw}. More problematic is the question of the region's area. Leaving a transit node's window means to enter the transit node's neighboring window without having any information what happens then. On the other hand, when an inner-window transit node is used for flow buffering, then the cell assignment to this region should reflect the cost of remaining in the present window. Both cases motivate choosing the window of the outgoing neighbor as the area for z_{Mw}:

$$A(z_{Mw}) := u \text{ with } u \text{ is the window of a node in } \Gamma^+_{G_f}(z_{wM}). \tag{5.17}$$

Note that, by construction, all neighbors of z_{Mw} in $\Gamma^+_{G_f}(z_{wM})$ belong to the same window, (5.17) is thus well-defined.
As long as there are non-realized arcs entering region nodes, we have to reserve a certain amount of capacity for the cells realized later. However, if the last arc is being realized, we

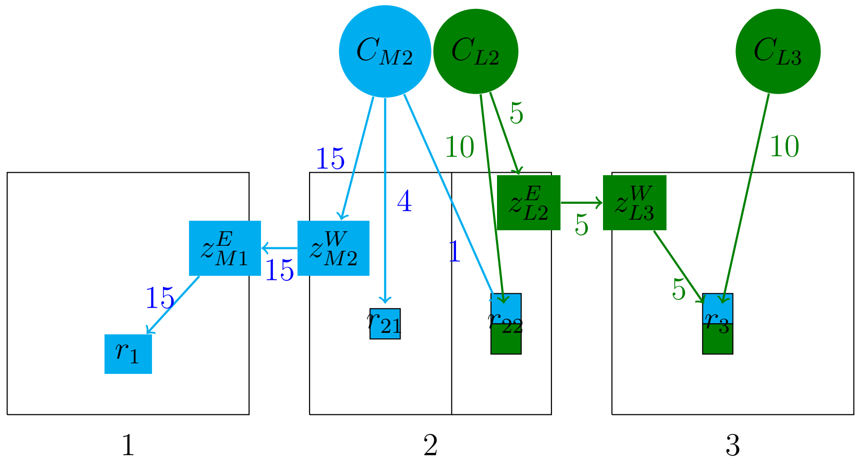


Figure 5.6: Example: Realizing the flow on 3 windows $\{1, 2, 3\}$ with 2 movebounds M,L. There are two external flow-carrying edges, namely (z^W_{M2}, z^E_{M1}) of movebound M and (z^E_{L2}, z^E_{L3}) of movebound L.

allow the use of the entire capacity of the region, not only the one computed by G_f, and leave the decision to the MULTISECTION algorithm, of whether the capacity will be used or not (cf. PUSHFLOW ll. 6–7).

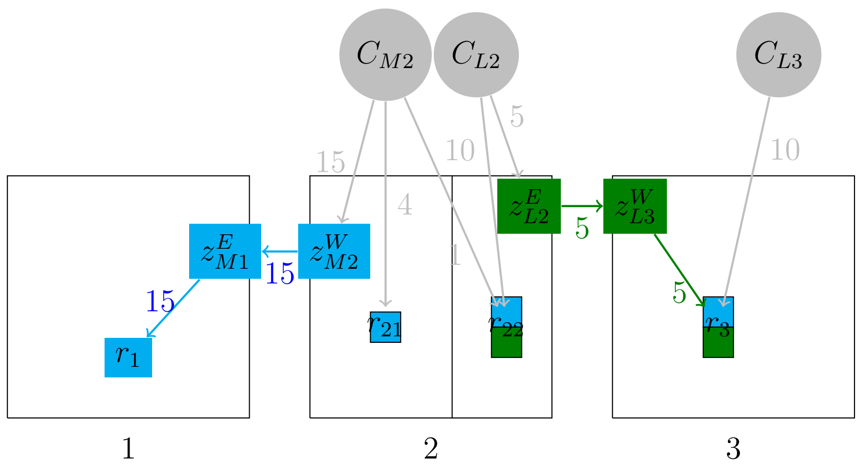
Finally, one aspect has to be addressed: the assignment costs for multisection cannot reflect the simple linear distance any more because linear cost could encourage cells to overtake others during that assignment. To prevent relative order changes, a superlinear tie breaker is necessary.

An example of the realization in a small instance is shown in Figures 5.6–5.7. In this example, there are 3 windows and 2 movebounds (Fig. 5.6).

Rounding issues

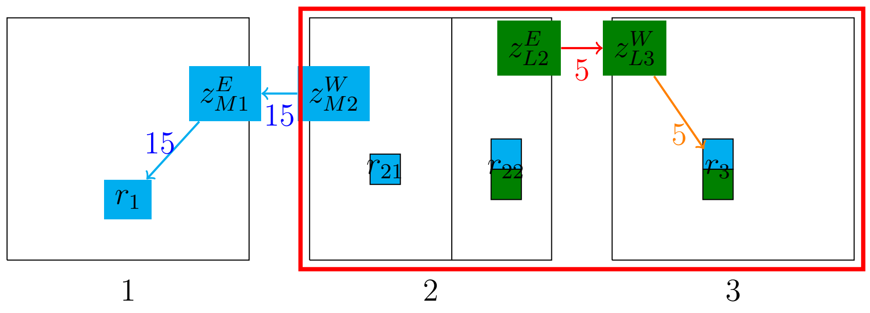
Until now, we have neglected the fact that flow realization cannot be done exactly due to rounding effects of MULTISECTION. Now rounding can have negative effects if transit nodes are affected. First, the flow may have drained away, in which case we may not have enough cells to realize in one window. In this case, PUSHFLOW can push no more than $\alpha = \min\{f_{(z,\cdot)}, \zeta(z)\}$ out of z. As the idea behind FLOWBASEDPARTITIONING is to minimize movement, stopping earlier than previously intended even has a positive effect.

On the other hand, by rounding, cells may also remain in some transit node, possibly in a window which is blocked or does not intersect its movebound. Here, we have to make sure that there is no remaining capacity at any transit node which cannot be pushed further. To this end, each time the *last* edge $e = (z^x_{Mv}, y)$ leaving z^x_{Mv} is realized, we check if all $\zeta(z^x_{Mv})$ flow has been pushed. There are two different cases: if e is an external edge, we know that y is the only successor of z^x_{Mv}, and so the entire amount of $\zeta(z^x_{Mv})$ is pushed towards y. Otherwise, e is an internal edge in v and current realization takes place in v. Then we enforce $\zeta(z^x_{Mv}) \equiv 0$ and let the balanced density increase (cf. Section 4.3.2) to



$a \in \mathcal{R} \cup \mathcal{Z}$	$\zeta(a)$
z^W_{L3}	
z^E_{L2}	5
z^W_{M2}	15
z^E_{M1}	
r_3	10
r_{22}	11
r_{21}	4
r_1	

Figure 5.7: Example: The flow-carrying subgraph after PushInitialFlow (left). Initial values of the buffer function ζ (right).



$a \in \mathcal{R} \cup \mathcal{Z}$	$\zeta(a)$
z^W_{L3}	$+5 - 5$
z^E_{L2}	$5 - 5$
z^W_{M2}	15
z^E_{M1}	
r_3	$10 + 5$
r_{22}	11
r_{21}	4
r_1	

Figure 5.8: Example: The flow-carrying subgraph when realizing the edge (z^E_{L2}, z^W_{L3}) in the coarse window $W = \{2, 3\}$. Values of the buffer function ζ on the right after pushing flow along (z^E_{L2}, z^W_{L3}) in red and after pushing flow along (z^W_{L3}, r_3) in orange. These capacities are used within Multisection in RealzeEdge in W.

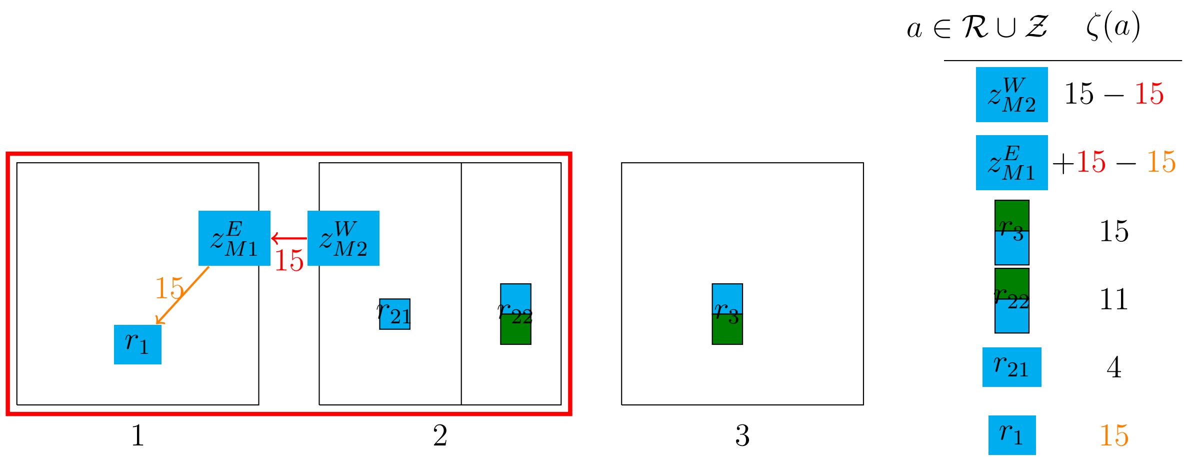


Figure 5.9: Example: The flow-carrying subgraph when realizing the edge (z^W_{M2}, z^E_{M1}) in the coarse window $W' = \{1, 2\}$. Values of the buffer function ζ on the right after pushing flow along (z^W_{M2}, z^E_{M1}) in red and after pushing flow along (z^E_{M1}, r_1) in orange. These capacities are used within MULTISECTION in REALZEEDGE in W'.

decide which nodes will absorb the missing capacity within the window v.

Scheduling

Let us recall the structure of G_f and for each window $w \in \mathcal{W}$ contract all nodes which belong to the same window w to one single node. We allow parallel edges in the contracted graph. We obtain a grid graph $G = (\mathcal{W}, E_1 \cup \cdots \cup E_m)$, where $m = |\mathcal{M}|$ and, by abuse of notation, $E_i := E^{ext}_M / \mathcal{W}$ is the set of external edges which belong to movebound M obtained by contraction. Lemma 5.2.4 tells us that each of the edge subsets $E_i,\ i = 1, \dots, m$ has a topological order. Now, realization is performed on coarser windows. In this context this corresponds to xy subgraphs of G, where an xy subgraph of G is an induced subgraph of G consisting of x subsequent horizontal and y subsequent vertical nodes. We can ask for the shortest possible *schedule*, i.e. for a number $t \in \mathbb{N}$ and a collection $F_1, \dots, F_t$ of xy subgraphs and a mapping $s : \bigcup_{i=1}^m E_i \to \{1, \dots, t\}$ with:

- $\forall e \in \bigcup_{i=1}^m E_i : \quad e \in E(F_{s(e)})$ (realize each edge)
- $\forall j \in \{1, \dots, m\}, \forall e, f \in E_i$: if there is a directed path in E_i visiting e, f in this order, then $s(e) \le s(f)$ (topological order).

This scheduling problem is not known to be solvable in polynomial time, even in this simple form of constant realization times per edge. Apart from the theoretical complexity, in practice, the use of multiple CPUs and the uncertain a-priori realization times make this problem more complicated. Additionally, we observe in our instances that the path lengths in G_f are in general very short, often involving one single external edge. Thus, most parts of the realization do not rely on predecessors and can be treated almost independently

window by window. To cope with some longer paths, we propose to compute distances from each node $v \in V(G_f)$ to some sink in G_f. We start the topological ordering with the longest path segments first (with respect to this distance). This is reasonable because in our realization the window in which some external flow edge e can be realized does not only cover the windows of nodes of e itself, but also takes neighbors into account, blocking the neighboring windows for realization steps. Such a policy reduces the risk of remaining paths, which cannot be processed in parallel.
As in repartitioning (see Section 6.3.1) we perform the parallelized realization in *phases*, using Π processors simultaneously, where a phase is one iteration of the second internal **while** loop (ll. 15–17) in procedure ScheduleRealization. To ensure determinism, we work again with a-priori stored safe locations for cells not owned by a particular thread and update location between phases. For a flow-carrying edge $e = (z^x_{Mv}, z^y_{Mw})$ in $E(G_f)$ with some movebound M, a coarse window W is chosen with $W \supseteq \{v, w\}$.

Procedure `ScheduleRealization`($G_f, \mathcal{W}, \zeta, \Pi$)

```
1  $E := E_{\text{ext}}(G_f)$
2  while $E \neq \emptyset$ do
3      Set $Q := \emptyset$
4      Set $E' := \emptyset$
5      for $w \in \mathcal{W}$ do
6          Set block($w$):=false
7      while $e \in E$ do
8          if $e$ does not have predecessors in $E$ then
9              if exists coarse window $W$ consisting of unblocked windows in $\mathcal{W}$ covering $e$
               then
10                 Choose such a $W$
11                 Set $Q := Q \cup \{(e, W)\}$
12                 Set $E' := E' \cup \{e\}$
13                 for $w \in W$ do
14                     Set block($w$):= true
15     StoreSafeLocations
16     for $\pi = 1, \ldots, \Pi$ in parallel do
17         while $Q \neq \emptyset$ do
18             $Q := Q \backslash \{(W, e)\}$
19             $\rho_{|W}$:=RealizeEdge ($G_f, \zeta, W, e$) //(using safe location)
20     $E := E \backslash E'$
```

Lemma 5.2.10 Given two coarse window-edge pairs (W, e) and (W', e') as chosen by Procedure ScheduleRealization in lines 7–8, then e and e' can be realized in W and

Algorithm 9: ParallelFlowBasedPartitioning $(\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{M}, \mu, \Pi)$

```
1 ProjctCellsToMovebounds $(\mathcal{C}, \mathcal{M}, \mu)$
2 $\rho :=$ AssignCellsToWins $(\mathcal{C}, \mathcal{W})$
3 $G_f :=$ ComputeAndSolveMinCostFlow $(\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{M})$
4 $\zeta :=$ PushInitialFlow $(G_f)$
5 $\rho :=$ ScheduleRealization $(G_f, \mathcal{W}, \zeta, \Pi)$
```

W' respectively, in either order.

Proof: Note that there is no predecessor for e and e'. In particular there is neither a directed $e - e'$ path nor a directed $e' - e$ path. Moreover $W \cap W' = \emptyset$, so no local operation in one coarse window affects the other one. □

Putting it all together yields the major result:

Theorem 5.2.11 Fix a set of cells $\mathcal{C}$, movebounds $\mathcal{M}$, $\mu : \mathcal{C} \to \mathcal{M}$ and a grid Γ. Let $\mathcal{W}$ be the set of windows induced by Γ. Then FlowBasedPartitioning (Algorithm 5) works correctly, i.e. it computes a feasible (fractional) partitioning $\rho : \mathcal{C} \to \mathcal{W}$ with movebounds if one exists.

Proof: By Proposition 5.2.2 the MinCostFlow instance is feasible if and only if the partitioning instance is feasible. Each cell is assigned to its movebound's bounding box, so the restriction to the movebound bounding boxes as in Remark 5.2.3 does maintain the feasibility. An optimal solution of the MinCostFlow is then computed. PushInitialFlow initializes temporary buffer regions as in (5.15) and guarantees the realization invariant (Prop. 5.2.9) at the beginning. Lemma 5.2.4 ensures that G_f is cycle-free and can be processed in topological order. Then, after the realization of some edge, the correctness of RealizeEdge is guaranteed again by Proposition 5.2.9. The algorithm terminates when there is no flow-carrying external edge in G_f, which concludes the proof by Theorem 5.2.8. □

Corollary 5.2.12 ParallelFlowBasedPartitioning works correctly.

Proof: Immediately from Theorem 5.2.11 and Lemma 5.2.10. □

5.2.4 Refinement of the Model

Cell Cluster Refinement

Instead of providing one cell group node per movebound and window, we propose to refine the cell groups geometrically in order to obtain an improved image of the cell distribution in each window. To this end we subdivide each window into a set of smaller subwindows and group cells which are located in the same subwindow and have the same movebound. The nodes C_{Mw} with edge sets E^{Cz}_{Mw} and E^{Cr}_{Mw} are then replaced by corresponding bipartite

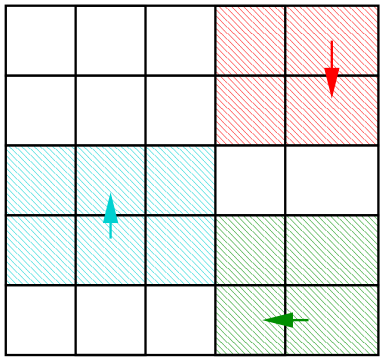


Figure 5.10: Example: three external flow-carrying edges with their realization windows.

graphs for each window and movebound wherever necessary. The subwindows serve to define cell group nodes only. After these have been defined, the subwindows are dropped. Our implementation uses uniform 5×5 to 2×2 subwindows, depending on the number of original windows for efficiency reasons. The smaller the number of windows, the finer subdivision.

Costs over Blockages

Movement over large blockages should be penalized, and cells should be encouraged to take a detour over free space instead of jumping over large obstacles. Long connections over placement blockages cannot be buffered properly, and pose the risk of timing problems. For this purpose, the costs for *internal* edges between transit nodes in which no accessible region exists are penalized by a constant factor. These penalties apply to blocked windows which are obstacles to all cells as well as to windows which do not contain any accessible region for some particular movebound.

Realization and Window Sizes

We have not specified the size of the coarse window W, in which an external flow-carrying edge $e = (z^x_{Mv}, z^y_{Mw})$ will be realized. There is of course a trade-off between the local view of the minimal realization size $W = \{v, w\}$, which does not affect any other fine window, and the global view of $W = \mathcal{W}$. In our tests the best results were in general obtained by a 3×2 realization window for the vertical case (i.e. $x, y \in \{N, S\}$) and 2×3 for the horizontal case. In both cases we leave a "margin" of 1×2 (resp. 2×1) window in W on either side of $\{v, w\}$ when realizing e. If such a two-sided margin is not possible, because the edge is near to the chip boundary, we choose the one-sided one (cf. Fig. 5.10).

Realizing More External Edges

In practice, we very often observe that once an edge e and a realization window W of e are chosen, there are other realizable edges in W. We use this opportunity and realize *all* realizable edges in W at the same time.

5.3 Extensions

Once we have an algorithm which guarantees a global feasible partitioning solution at any stage of the placement, we can drop the partitioning invariant from the recursive partitioning.

5.3.1 Level Skipping

The first, perhaps simplest idea is that of dropping some level, say k, and proceeding from $k-1$ directly to $k+1$. This makes sense in particular, where the global partitioning decisions are bad and do not reflect the chip's floorplan. This is the case especially for chips with large degrees of freedom and relatively large blocked areas in the center, which leave some free space at the boundary.
In addition, the initial levels of placement were hardly parallelizable and pretty time consuming. This can also be resolved by skipping the most time consuming parts of the placement. In our experiments, we see that dropping levels 1 and 2, i.e continuing with level 3 (with its 8×8 grid in general) after the initial QP is not even harmless, but it has a positive effect for most test cases, especially if the later partitioning revisions are limited, as it is the case for 2×2 repartitioning.

5.3.2 Greedy Re-allocations

Another option is to allow different density-unaware algorithms to come into play and to resolve the density afterward through FlowBasedPartitioning. Here we have tested a simple greedy heuristic in which we allow each cell to be put into its favorite location, e.g. w.r.t. the linear netlength. Other methods, for instance incorporating timing optimization, are also possible: one can think of an algorithm which focuses on critical paths, places them optimally, distributes inertias to the timing critical cells and continues realization.

5.3.3 Densities

The new global partitioning method provides a self-correcting method for density violation. As in any level, the capacities and cell sizes per window are recomputed, and the algorithm tends to even all density overloads up, even those caused by rounding. This is one of the major advantages of this approach. Density safety margins, which are required by the recursive and iterative partitioning algorithms as used in old BonnPlace's global placement, are no longer needed. The entire capacity can thus already be revealed to the placement algorithm in the early levels, leaving more space for global optimization. But this is not the only advantage: density violations which appear locally now have the chance to be resolved at a global stage. This allows it to cope with increased cell sizes as well as areas blocked during global placement. Density snapshots from levels 2–7 after flow-based partitioning are shown in Figure 5.11.

5.4 Simultaneous Inflation and Movement Model

In congestion-driven placement, given a partitioning, routability is evaluated and in congested areas the cells are placed loosely to leave more space for routing. One option to achieve such a placement is to increase cell sizes artificially (inflation) and perform another partitioning as proposed by Brenner and Rohe [2002] (cf. also Section 3.4.5). However, the cell positions are not conclusively defined in early and intermediate placement levels, and in coarse grids the routability estimation may also be inaccurate, so in many cases the congestion problems cannot be detected at an early stage. On the other hand, for congestion problems detected late, it might be too late to get them resolved by a local partitioning method like iterative partitioning. In instances with movebounds the problem becomes even harder, as iterative partitioning cannot be applied. The recursive partitioning algorithm, which is used instead, has an even more restricted line of action. But this is not the only problem. Beyond congestion avoidance, one has to make sure that the partitioning instances do not become infeasible by cell inflation — the inflated cell size may exceed the total free area.

Flow-based partitioning can easily be applied to resolve late congestion problems: due to its global view, the excess of cell size can be shifted globally respecting movebounds. There is no restriction to local partitioning steps. Cell inflation is an auxiliary method to reduce real cell density in a window but it is also suitable to address local routing problems (*pin access*). Another option to address pin density would be to decrease regions' target densities in congested areas. Congestion-driven placement should be applied with care. On easily routable designs the application of the congestion avoidance methods would lead to unnecessary movement and netlength increase. Additional capacity should be provided in the neighborhood of congested areas if possible. To this end, simultaneous control of inflation, movement and density is desirable.
For a cell $c \in \mathcal{C}$ let $\mathrm{infl}(c) \in \mathbb{R}_+$ denote the additional amount of area for the cell c by inflation. When referring to $\mathrm{size}(c)$ for $c \in \mathcal{C}$, we assume that $\mathrm{size}(c)$ is the area demand of cell c *including inflation*.
Recall the MinCostFlow instance $G = (V(G), E(G), b, \mathrm{cost})$ used in FlowBasedPartitioning as in Section 5.2.1. We define a new capacitated MinCostFlow instance:

$$G' = (V(G'), E(G'), b', \mathrm{cost}', \mathrm{capa}') \tag{5.18}$$

by introducing a new node t, and series of parallel edges between region nodes and the node t:

$$V(G') := V(G) \dot\cup \{t\} \tag{5.19}$$

$$E(G') := E(G) \dot\bigcup_{r \in \mathcal{R}} E_r, \tag{5.20}$$

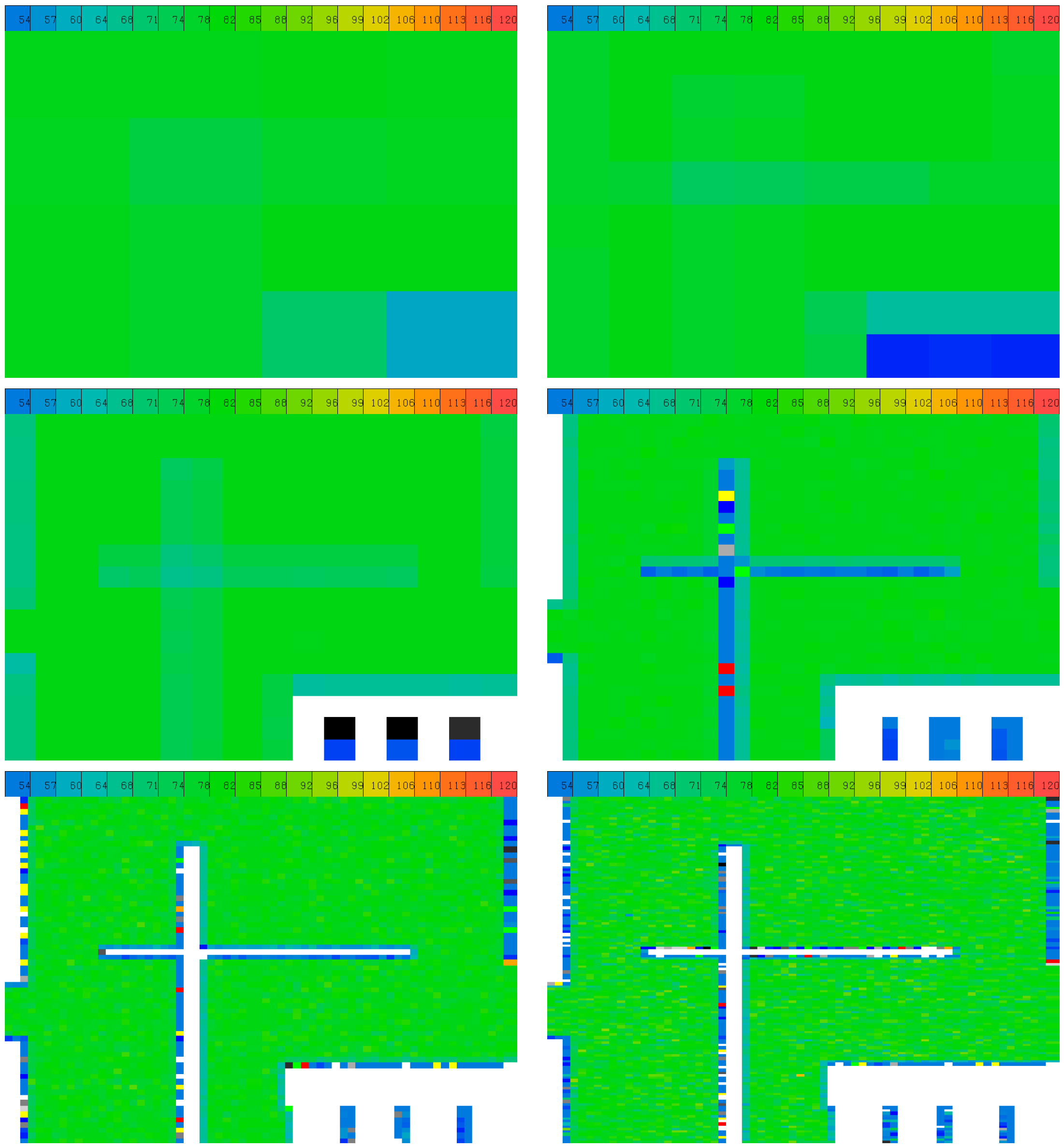


Figure 5.11: Density maps in the new partitioning algorithm in level 2 (top left) to 7 (lower right corner). Unlike old partitioning methods (cf. Fig. 5.1), the new partitioning method repects the overall density target of 88% even during late levels.

where for $r \in \mathcal{R}$, E_r is a collection of parallel, capacitated edges between the region node r and t (as in Section 4.3.2) reflecting the monotonic increasing costs for density adjustments in region r. Set $b'(r) := 0$ for all original region nodes $r \in V(G') \cap \mathcal{R}$, set $b'(t) := \sum_{c \in \mathcal{C}} -\text{size}(c)$ and $b' \equiv b$ for $v \in V(G) \setminus \mathcal{R}$. We set $\text{cost}'(e) := \text{cost}(e)$ for all $e \in E(G)$, and for parallel arcs $e \in E_r$, $r \in \mathcal{R}$ we set the costs as in Section 4.3.2, multiplied by a constant corresponding to the average window diameter. With this construction, movement cost can be appropriately merged into the concept of balanced density increase for regions. One sees immediately:

Lemma 5.4.1 The MinCostFlow instance $(G' = (V(G'), E(G'), b', \text{cost}', \text{capa}')$ has a solution with finite costs if and only if the partitioning instance is feasible.

Proof: Direct combination of Proposition 5.2.2 and Lemma 4.3.9. $\square$

Following the same procedure as in Section 4.3.2, the capacity in regions is increased and we are almost done. However, cell inflation could have destroyed feasibility. To recover a feasible solution, the inflated cell sizes have to satisfy the flow condition as in Lemma 4.3.3. This can be achieved by reducing the inflation values, but again, one has to act with care to maintain the desired congestion reduction effect.
The common movement-density increase in the MinCostFlow instance G' as in (5.18) can serve as a starting point. We propose to treat the inflation excess as capacity which will be absorbed by new artificial regions. Thus, we first extend G' to a new MinCostFlow instance G'' by adding new region nodes, one per movebound:

$$V(G'') := V(G) \dot\cup \{r_M | M \in \mathcal{M}\} \tag{5.21}$$

Then, for a movebound $M \in \mathcal{M}$, the new region node r_M is connected by new edges to all cell cluster nodes on one side and by new sequences of parallel edges E_M to the sink t on the other side:

$$E(G'') := E(G) \dot\cup \bigcup_{M \in \mathcal{M}} \{(C_{Mw}, r_M) | w \in \mathcal{W}\} \dot\cup \bigcup_{M \in \mathcal{M}} E^M, \tag{5.22}$$

We maintain all costs and capacities for all arcs of $E(G')$ in $E(G'')$ and all b'' values for all vertices $V(G')$ in $V(G'')$. For the newly introduced edges we set $\text{capa}((C_{Mw}, r_M)) := \infty$ and $\text{cost}''((C_{Mw}, r_M)) := 0$ for all $M \in \mathcal{M}$, $w \in \mathcal{W}$, and we set $b''(r_M) := 0$ for the new regions for each $M \in \mathcal{M}$.
We treat r_M as buffers for inflation excess, whose violation is expensive. In the consequence, in a simillar manner as in Section 4.3.2, r_M is a region with target density 0 and free area corresponding to $\sum_{c \in \mathcal{C}_M} \text{infl}(c)$ accessible for cells in $\mathcal{C}_M$. The costs of edges in E_M are chosen as in Section 4.3.2 for the parallel arcs connecting regions and the sink node. We scale these costs with much higher constant values, in order to allow a certain cell movement across the chip first before starting to decrease inflation values. This leads to:

Corollary 5.4.2 The MinCostFlow instance $G'' = (V(G''), E(G''), b'', \text{cost}'', \text{capa}'')$ for any inflation values has a solution with finite costs if and only if the partitioning instance is feasible.

Proof: Immediately from Lemma 5.4.1 using the fact that for any movebound the total inflation values of cells in $\mathcal{C}_M$ can be absorbed by flow through r_M. □

Remark 5.4.3 Assume the partitioning instance is feasible. Given a solution f of the MinCostFlow instance from Corollary 5.4.2, then for each movebound $M \in \mathcal{M}'$ the inflation values of cells in $\mathcal{C}_M$ are reduced uniformly by the factor of ε_M, where

$$\varepsilon_M := \frac{\sum_{c \in \mathcal{C}_M} \text{infl}(c) - \sum_{e \in \delta^-_{G''}(r_M)} f_e}{\sum_{c \in \mathcal{C}_M} \text{infl}(c)}. \tag{5.23}$$

This means that for each cell $c \in \mathcal{C}$, the inflation value $\text{infl}(c) := \varepsilon_{\mu(c)}\text{infl}(c)$

We summarize construction of G'', the MinCostFlow computation with the following (potential) inflation reduction as in Remark 5.4.3 as the *flow-based density and inflation adjustment*.

The partitioning method in congestion-driven placement consists — as in Section 3.4.5 — of a flow-based partitioning step, followed by rough congestion estimation. Then cells are inflated in congested areas, a flow-based density and inflation adjustment is done and finally another flow-based partitioning step is performed.

5.5 Discussion and Outlook

The presented global partitioning algorithm unifies the advantages of partitioning-based placement schemes. High efficiency with formidable parallelization opportunities is combined with robust density control. Moreover, improved movebound handling is provided using the global view of the entire placement instance. The cells are not moved to distant windows if this is not necessary as in iterative partitioning.
Furthermore, wherever movement is needed, connectivity is considered in a much more precise way than during recursive partitioning. By the choice of coarse realization window (cf. Fig. 5.10) for a horizontal (vertical) flow-carrying edge, the cells in the windows below and above (left and right) contribute to improved connectivity information and priority choice for partitioning.
Another advantage is the prescribed amount of capacity which has to be shifted along an edge: this leads to an improved usage of ConstraintQP. Prescribing center of gravity to cells prior to partitioning may have the negative effect that some cells are moved too far away by ConstraintQP, and by movement minimization they are assigned to distant windows, even if a feasible partitioning could be done in proximate ones. When prescribing capacities, it is the relative positions which count and not the absolute ones.
The realization is currently performed via Multisection. However, the algorithm is in principle generic — it requires *some oracle* which shifts a prescribed amount of cells from

one window to another. Especially in congested regions or in cases where a shift over blockages has to be performed, it is also possible to apply some *MinCut*–approach.
In our implementation, we use an equidistant grid for the window computation, which is not mandatory: the grid lines could be refined and adjusted to the movebound/blockage structure in order to enable a further refinement of the a-priori movement costs.
The flow-based global partitioning offers a method that allows us to start with *any* given placement, in particular it is interesting for incremental global placement approaches.
Finally, our current implementation computes one single MinCostFlow and realizes the solution until all external flow-carrying edges vanish. An iterative process which computes a MinCostFlow, realizes some of the edges, and restarts with MinCostFlow computation based on the recently computed positions is also possible.

Chapter 6

Data Structures and Implementation

In this chapter we first focus on netlist and clustering data structures and their efficient handling during global placement. Then, we present other implementation details addressing aspects of industrial use. Later we show how efficient parallelization of core routines in BonnPlace is implemented.

6.1 Generalized Netlists and Clustering

6.1.1 Motivation

Recall the definition of a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$. The essential placement task is to find a position for each cell $c \in \mathcal{C}$. A *cell* represents standard circuits (gates) acting for some logical functions (ANDs, ORs), memory elements (latches) and non-standard circuits, such as gate-arrays or larger memory blocks (DRAMs, SRAMs). The complexity of chips increased and design costs play an important role, so one uses hierarchical units (RLMs, SoCs, IP-cores) which are (if one forgets the hierarchical structures) another class of placeable objects. But logic and memory cells are not the only circuits a placement has to deal with: DECAPs and ENDCAPs address particular electric constraints without having any influence on the logical function.

For mixed-size designs, Adya and Markov [2002] proposed to subdivide large blocks into small fragments connected by highly weighted nets in order to obtain objects of similar size. To these modified instances, global placement is applied, then the original block structure is recovered, the large blocks are fixed in legal positions and one starts another iteration of the placement. Hence, the *fragments* of a circuit can also be seen as a cell.

Apart from circuits and their fragments, groups of circuits *(clusters)* are of particular interest for multiple reasons and purposes. Electrical properties impose tight resistance and capacitance bounds between *pass-gates* and latches. In recent technologies local clock buffers (LCBs) have been decomposed into several circuits (atoms) with extremely tight resistance bounds on connecting nets. In both cases the bounds force the cells to be placed in proximity to each other. Another reason is power consumption, which restricts DECAPs

to the vicinity of some large block to prevent power drops. Routability is another reason: latch groups are often merged to blocks to make routing easier; the same applies to buses, in which several parallel nets connect groups of cells leaving basically no room for relative position changes. Spare-logic groups and error-correction code (ECC) are other object classes which should be placed together.
Design rules are not the only reasons for the application of cell clusters in placement. Clusters reduce the instance sizes and may serve to accelerate any placement algorithm. For quadratic placers another reason is also important: clustering provides some linearization effects which help to minimize linear netlength for the price of the quadratic one. Finally, for partitioning-based placements clustering may help to prevent cell groups from being torn into pieces and assigned to distant regions.

Beyond cells, nets are also not homogeneous connections. Nets, which are responsible for signal transmission among cells, are one kind of connection. But nets can also represent artificial connections to preplaced locations on the chip (see Section 3.3.2) for cell spreading, for movebound attractors (see Section 4.3.4) as well as for modeling *fixations*. A fixation is a weighted virtual net connected to some position on the chip modeling a certain affinity of the cell to this location. Fixations are important for incremental placements. Infinite weight means that the cell is fixed at this location.
Virtual connections can also be used between movable objects, for example to stress the timing criticality of a particular 2-pin connection in a larger net, or to attract DECAPs to macros or latch groups to the driving LCB. Finally, they are also used for mixed size placement to connect circuit fragments.

6.1.2 Clustering

Here we make the concept of clustering and clustering data structures more precise. When dealing with groups of cells it is reasonable to allow the general *hierarchical* approach: clusters should be able to contain other clusters. This is motivated by two different issues. First, hierarchical clusterings are indeed used in placement and seem natural where for example pass-gates and latches as well as different LCB atoms can be seen as the finest clustering hierarchy. These clusters can then be grouped together for power-consumption or timing reasons.
In VLSI placement, clustering is used in two different ways. The *persistent* clustering, in which the netlist is clustered *once* and the modified netlist is presented to the placement tool (see e.g. Hu and Marek-Sadowska [2003] as well as Yan, Chu and Mak [2010]) and the non-persistent one, where clustering is modified within the placement run (Wang, Yang and Sarrafzadeh [2000], Karypis et al. [1999] and Nam et al. [2006]).
The hierarchical approach allows the semi-persistent use of clusterings which seems natural for bottom-up clustering techniques in combination with top-down window refinement (see also Nam et al. [2006]). In the top-down partitioning scheme, the cell clusters are resolved only if they are too big relative to the window sizes. To avoid long re-clustering runtimes,

we maintain the option to resolve the hierarchy wherever necessary but to keep several other clustering decisions at the same time. Of course, the hierarchical cluster handling allows the use of a single hierarchy level, the *flat* clustering. For all these reasons we decide in favor of the general hierarchical clustering. Formally:

Definition 6.1.1 (Clustering) For a set of cells $\mathcal{C}$, a *clustering* is a branching $\mathcal{K} = (V(\mathcal{K}), E(\mathcal{K}))$ with a bijection $\varkappa : \mathcal{C} \to \{v \in V(\mathcal{K}) | \delta^+(v) = \emptyset\}$. A node $v \in V(\mathcal{K})$ is called a *cluster*. For a cell $c \in \mathcal{C}$, the corresponding node $\varkappa(c) \in V(\mathcal{K})$ is called the *leaf cluster* of c. A node $r \in V(\mathcal{K})$ with $\delta^-(r) = \emptyset$ is called a *root cluster*. Note that, by the branching property of $\mathcal{K}$, each cluster has a unique associated root cluster, namely the root of its arborescence. We postulate $|\delta^+(v)| > 1$ for each non-leaf $v \in V(\mathcal{K})$, i.e. we forbid clusters which do not create new hierarchies.

Clustering can thus be seen as a special laminar decomposition of the underlying cell set: the roots are the top-level clusters and the cells can be seen as leaves. During placement, the cells in a common cluster are hidden by the cluster and should be moved together, hence a root determines the positions of all the cells in its arborescence.
One can treat the roots as particular cells, but should keep in mind that this approach can cause several problems. First, the relative positions of the cells within a cluster are in general not known a-priori, raising another problem with cluster shapes and relative pin positions. Furthermore, hiding the cells in a cluster also hides essential properties of the cluster members, in particular movebounds and possible orientations.
We propose two different cluster types, the hard and soft clusters. A *hard cluster* is a collection of cells *with fixed relative positions* and *orientations*. Thus, if a position of any member is determined, all other members are placed with the prescribed orientation and relative positions. In particular, the relative pin offsets, the width and the height of the clusters are fixed and there is no need for hierarchical clustering of hard clusters. Hard clusters will be treated as macro cells in the following.
In global placement, the cells are treated as shapeless objects and we do not pay attention to grids. Hence we will also treat *soft clusters* as shapeless objects demanding some capacity. In top-down partitioning based placement, cells are fixed as soon as the edge width exceeds a certain percentage of the window edge. Therefore, we will store the maximal width and height of the member cells, partially revealing the shape structure of the cluster. On the other hand, movebound constraints must be entirely revealed to the global placement. We require all leaves in a cluster to have the same movebound.

Definition 6.1.2 Let $\mathcal{C}$ be a set of cells, $\mathcal{K} = (V(\mathcal{K}), E(\mathcal{K}))$ a clustering with $\varkappa : \mathcal{C} \to \{v \in V(\mathcal{K}) | \delta^+(v) = \emptyset\}$. For a root $r \in V(\mathcal{K})$ let $\lambda(r)$ be the set of leaves in the arborescence of r. For each root $r \in V(\mathcal{K})$ we define a new *root cluster cell* c_r with:

$$\mathrm{size}(c_r) := \sum_{c \in \varkappa^{-1}(\lambda(r))} \mathrm{size}(c),$$

$$\mathrm{size}_x(c_r) := \max_{c \in \varkappa^{-1}(\lambda(r))} \mathrm{size}_x(c), \qquad \mathrm{size}_y(c_r) := \max_{c \in \varkappa^{-1}(\lambda(r))} \mathrm{size}_y(c).$$

By the remark above, the movebound $\mu(c_r)$ is the same for all its members, thus $\mu(c_r) := \mu(c),\ c \in \varkappa^{-1}(\lambda(r))$. Let $\mathcal{C}^{\mathcal{K}}$ be the set of all root cell clusters.
We define the map $\kappa : \mathcal{C} \to \mathcal{C}^{\mathcal{K}}$ with $\kappa(c) := c_r$ where r is the root of $\varkappa(c) \in V(\mathcal{K})$. For each cell $c \in \mathcal{C}$, $\kappa(c)$ is then the root cell cluster of c.

6.1.3 Clustered Netlist

Clustering yields new placeable objects, the root cluster cells. Now we focus on pins and nets. If there were only 2-terminal nets in a netlist, then nets connecting two leaves in the same arborescence would be hidden, and the others would simply be attached to the corresponding roots. In general nets, the situation becomes more complicated, as *parts* of a net can be hidden by a cluster. To this end we extend the net and pin definitions:

Definition 6.1.3 Fix a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$ and a clustering $\mathcal{K} = (V(\mathcal{K}), E(\mathcal{K}))$, $\varkappa : \mathcal{C} \to \{v \in V(\mathcal{K}) | \delta^+(v) = \emptyset\}$. The clustering decomposes each net $N \in \mathcal{N}$ into a family of disjoint subsets:

$$N = N_1 \dot\cup \ldots \dot\cup N_l \tag{6.1}$$

where $N_i = \{p \in N | \gamma(p) \in \mathcal{C}$ and $\varkappa(\gamma(p))$ are in the same arborescence$\}$ or $N_i = \{p | \gamma(p) = \square\}$ for $i = 1, \ldots, l$. For a pin $p \in N_i \subseteq N$ the *multiplicity of* p *in* N is the number $\varrho(p) := |N_i|$.
Then for a pin $p \in P$ the *clustered pin* $p^{\mathcal{K}}$ of p is a pair $p^{\mathcal{K}} := (\pi(p), \varrho(p))$ with $\pi(p) = N_i$. For a net N the *clustered net* $N^{\mathcal{K}} := \{p_1^{\mathcal{K}}, \ldots, p_l^{\mathcal{K}}\}$ with $p_i^{\mathcal{K}} = (N_i, |N_i|)$ is a set of clustered pins obtained from clustering pins in N.
Canonically, we set $P^{\mathcal{K}}$ to be the set of all clustered pins and define $\gamma^{\mathcal{K}} : P^{\mathcal{K}} \to \mathcal{C}^{\mathcal{K}}$ by

$$\gamma(p^{\mathcal{K}}) := \begin{cases} \kappa(\gamma(p)) & \text{if } \gamma(p) \in \mathcal{C}, \\ \square & \text{else.} \end{cases} \tag{6.2}$$

The clustered netlist of $(\mathcal{C}, \mathcal{N}, P, \gamma)$ is the quadruple $(\mathcal{C}^{\mathcal{K}}, \mathcal{N}^{\mathcal{K}}, P^{\mathcal{K}}, \gamma^{\mathcal{K}})$.

Note that the map $p \mapsto p^{\mathcal{K}}$ is in general not injective: $\varrho(p)$ pins in N end up in a single clustered pin $p^{\mathcal{K}}$. Because of the uncertainty of the final position of the cells within the cluster, we put all clustered pins of real clusters to the root cell cluster's center:

Definition 6.1.4 Given a netlist, a clustering and the induced clustered netlist, for a pin $p \in P$ we define the offsets for its clustered pin $p^{\mathcal{K}}$

$$\big(x_{\text{offs}}(p^{\mathcal{K}}), y_{\text{offs}}(p^{\mathcal{K}})\big) := \begin{cases} \big(x_{\text{offs}}(p), y_{\text{offs}}(p)\big) & \text{if } \gamma(p) \in \mathcal{C} \text{ and } \varkappa(\gamma(p)) \text{ is root in } \mathcal{K}, \\ (0,0) & \text{else.} \end{cases} \tag{6.3}$$

Proposition 6.1.5 For a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$ and a clustering $(\mathcal{K}, \varkappa)$, the clustered netlist can be constructed in $\mathcal{O}(|P| + |\mathcal{C}|)$ time.

Proof: By Def. 6.1.1, $|V(\mathcal{K})| = \mathcal{O}(|\mathcal{C}|)$ and we can store for each $c \in \mathcal{C}$ the root of c in $\mathcal{K}$ first. Then, for each net $N \in \mathcal{N}$ and for each $p \in N$, we ask for the root of $\gamma(p)$, which can now be queried in constant time. For each net N, the clustered net $N^{\mathcal{K}}$ can thus be constructed in $\mathcal{O}(|N|)$ time. As $|P| = \sum_{N \in \mathcal{N}} |N|$ we are done. $\square$

We now define the analogous quadratic net models for the clustered nets.

Definition 6.1.6 Let $N^{\mathcal{K}} = \{p_1^{\mathcal{K}}, \ldots, p_l^{\mathcal{K}}\}$ be a clustered net with clustered pins $p_i^{\mathcal{K}} = (\pi_i, \varrho(p_i))$, $i = 1, 2, \ldots, l$. Set $n = \sum_{i=1}^{l} \varrho(p_i)$. Then we define

$$CL'_{2,x}(N^{\mathcal{K}}) := \frac{1}{n-1} \sum_{i=1}^{l} \sum_{j=i}^{l} \varrho(p_i)\varrho(p_j)\big(x(p_i^{\mathcal{K}}) - x(p_j^{\mathcal{K}})\big)^2, \tag{6.4}$$

and

$$ST'_{2,x}(N^{\mathcal{K}}) := \sum_{i=1}^{l} \varrho(p_i)\Big(x(p_i^{\mathcal{K}}) - \frac{1}{n}\sum_{j=1}^{l} \varrho(p_j)x(p_j^{\mathcal{K}})\Big)^2, \tag{6.5}$$

and set $CL'_2(N^{\mathcal{K}}) := CL'_{2,x}(N^{\mathcal{K}}) + CL'_{2,y}(N^{\mathcal{K}})$ as well as $ST'_2(N^{\mathcal{K}}) := ST'_{2,x}(N^{\mathcal{K}}) + ST'_{2,y}(N^{\mathcal{K}})$. For $l = 1$ we set $CL'_2(N^{\mathcal{K}}) = ST'_2(N^{\mathcal{K}}) := 0$.

Theorem 6.1.7 Given a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$ and a clustering $(\mathcal{K}, \varkappa)$, for a net $N \in \mathcal{N}$ let $N^{\mathcal{K}} = \{p_1^{\mathcal{K}}, \ldots, p_l^{\mathcal{K}}\}$, $l > 1$ be the clustered net of N. Let $N = N_1 \dot\cup, \ldots, \dot\cup N_l$ be a decomposition of N w.r.t. the clustering, i.e. $p_i^{\mathcal{K}} = (N_i, |N_i|)$ for $i = 1, \ldots, l$. Now, for each $i = 1, \ldots, l$ set the pin positions in N_i to $x(p_i^{\mathcal{K}})$. Then:

$$CL_2(N)_{|\forall i \forall p \in N_i : x(p) = x(p_i^{\mathcal{K}})} = CL'_2(N^{\mathcal{K}}) \tag{6.6}$$

$$ST_2(N)_{|\forall i \forall p \in N_i : x(p) = x(p_i^{\mathcal{K}})} = ST'_2(N^{\mathcal{K}}). \tag{6.7}$$

In particular, clustering the pin locations of the unclustered net model is the same as applying the clustered net model to the clustered pins.

Proof: We focus again on the horizontal part only. Assume $l > 1$.

$$\begin{aligned}
CL_{2,x}(N)_{|\forall i \forall p \in N_i : x(p) = x(p_i^{\mathcal{K}})} &= \frac{1}{|N|-1} \sum_{i=1}^{l} \sum_{p \in N_i} \sum_{j=i}^{l} \sum_{q \in N_j} (x(p_i^{\mathcal{K}}) - x(p_j^{\mathcal{K}}))^2 \\
= \frac{1}{|N|-1} \sum_{i=1}^{l} \sum_{j=i}^{l} |N_i||N_j|(x(p_i^{\mathcal{K}}) - x(p_j^{\mathcal{K}}))^2 &= \frac{1}{\sum_{m=1}^{l} |N_m| - 1} \sum_{i=1}^{l} \sum_{j=i}^{l} |N_i||N_j|(x(p_i^{\mathcal{K}}) - x(p_j^{\mathcal{K}}))^2 \\
= \frac{1}{\sum_{m=1}^{l} \varrho(p_m) - 1} \sum_{i=1}^{l} \sum_{j=i}^{l} \varrho(p_i)\varrho(p_j)(x(p_i^{\mathcal{K}}) - x(p_j^{\mathcal{K}}))^2 &= CL'_{2,x}(N^{\mathcal{K}}).
\end{aligned} \tag{6.8}$$

For the star model observe that $\frac{1}{|N|}\sum_{i=1}^{l}\sum_{p\in N_i} x(p_i^{\mathcal{K}}) = \frac{1}{\sum_{i=m}^{l}\varrho(p_m)}\sum_{i=1}^{l}\varrho(p_i)x(p_i^{\mathcal{K}})$. Then,

$$\begin{aligned} ST_{2,x}(N)_{|\forall i \forall p\in N_i : x(p)=x(p_i^{\mathcal{K}})} &= \frac{1}{|N|-1}\sum_{i=1}^{l}\sum_{p\in N_i}\Big(x(p_i^{\mathcal{K}}) - \frac{1}{|N|}\sum_{j=1}^{l}\varrho(p_j)x(p_j^{\mathcal{K}})\Big)^2 \\ &= \frac{1}{\sum_{m=1}^{l}\varrho(p_m)-1}\sum_{i=1}^{l}\varrho(p_i)\Big(x(p_i^{\mathcal{K}}) - \frac{1}{|N|}\sum_{j=1}^{l}\varrho(p_j)x(p_j^{\mathcal{K}})\Big)^2 \\ &= ST'_{2,x}(N^{\mathcal{K}}). \end{aligned} \tag{6.9}$$

The cases for $l = 1$ are trivial. □

Corollary 6.1.8 For a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$ and a trivial clustering $\mathcal{K} = (\mathcal{C}, \emptyset)$ with $\varkappa = \text{id}$, we have for each net N and its trivial clustered version $N^{\mathcal{K}}$ the equalities: $CL_2(N) = CL'_2(N^{\mathcal{K}})$ and $ST_2(N) = ST'_2(N^{\mathcal{K}})$.

Proof: Immediately from Theorem 6.1.7, with $p_i^{\mathcal{K}} = (\{p_i\}, 1)$, $i = 1, \ldots, |N|$. □

Hence, the clustered net models reflect the behavior of the unclustered ones. Their essential advantage is the runtime for computation:

Corollary 6.1.9 For a clustered net $N^{\mathcal{K}} = \{p_1^{\mathcal{K}}, \ldots, p_l^{\mathcal{K}}\}$ with clustered pins $p_i^{\mathcal{K}} = (p_i, \varrho(p_i))$, $i = 1, \ldots, l$ we have:

$$2\sum_{i=1}^{l}\varrho(p_i)ST'_2(N^{\mathcal{K}}) = \Big(\sum_{i=1}^{l}\varrho(p_i) - 1\Big)CL'_2(N^{\mathcal{K}}). \tag{6.10}$$

In particular, both net models can be computed in $\mathcal{O}(l)$ time.

Proof: Follows immediately using the equivalence of the unclustered net models and Theorem 6.1.7. □

Clustering may have merged all pins of an initial net in a single cluster. In this case the clustered net would consist of a single pin and provide no additional connectivity information. Thus, we omit such nets during clustered netlist construction. On the other hand, other nets may now connect the same groups of cluster cells even if they do not connect the same cells. It would be reasonable to merge them into one net with accordingly adjusted weights, but beyond the 2-terminal case it is not possible in general (see Lauff [2010]).

One should note that for a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$ and a clustering $(\mathcal{K}, \varkappa)$, the induced clustered netlist $(\mathcal{C}^{\mathcal{K}}, \mathcal{N}^{\mathcal{K}}, P^{\mathcal{K}}, \gamma^{\mathcal{K}})$ can again be seen as a starting point for further clustering.

Definition 6.1.10 (Hierarchically clustered netlist) For a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$ and a clustering $(\mathcal{K}, \varkappa)$ of $\mathcal{C}$, let $(\mathcal{C}^{\mathcal{K}}, \mathcal{N}^{\mathcal{K}}, P^{\mathcal{K}}, \gamma^{\mathcal{K}})$ be the clustered netlist induced by $(\mathcal{K}, \varkappa)$. Let $(\mathcal{K}', \varkappa')$ be a clustering of $\mathcal{C}^{\mathcal{K}}$.

Then $\mathcal{C}^{\mathcal{K}'}$ is the canonically defined set of clustered cells of $\mathcal{C}^{\mathcal{K}}$. For a net $N \in \mathcal{N}$, consider the clustered pins $p_1^{\mathcal{K}}, \ldots, p_l^{\mathcal{K}} \in P^{\mathcal{K}}$, with $p_i^{\mathcal{K}} = (\pi(p_i), \varrho(p_i))$ for $i = 1, \ldots, l$, whose cells have been merged to one arborescence in $\mathcal{K}'$. Define the clustered pin $p^{\mathcal{K}'} := (\bigcup_{i=1}^{l} \pi(p_i), \sum_{i=1}^{l} \varrho(p_i))$. For a pin $p^{\mathcal{K}} \in P^{\mathcal{K}}$ and its corresponding clustered (w.r.t. $\mathcal{K}'$) pin $p^{\mathcal{K}'}$, define $\gamma^{\mathcal{K}}(p^{\mathcal{K}'})$ to be the root cell cluster of $\varkappa'(\gamma^{\mathcal{K}}(p^{\mathcal{K}}))$ if $\varkappa'(\gamma^{\mathcal{K}}(p^{\mathcal{K}}))$ is a root in $\mathcal{K}'$, and $\gamma^{\mathcal{K}'}(p^{\mathcal{K}'}) := \Box$ else. For a net $N^{\mathcal{K}} \in \mathcal{N}^{\mathcal{K}}$ let $N^{\mathcal{K}'}$ be the set of clustered pins $p^{\mathcal{K}'}$ corresponding to pins in $N^{\mathcal{K}}$ and $P^{\mathcal{K}'} := \bigcup N^{\mathcal{K}'}$.

Corollary 6.1.11 Fix a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$, a clustering $(\mathcal{K}, \varkappa)$ of $\mathcal{C}$, and the induced clustered netlist $(\mathcal{C}^{\mathcal{K}}, \mathcal{N}^{\mathcal{K}}, P^{\mathcal{K}}, \gamma^{\mathcal{K}})$. For a clustering $(\mathcal{K}', \varkappa')$ of $\mathcal{C}^{\mathcal{K}}$, the induced clustered netlist $(\mathcal{C}^{\mathcal{K}'}, \mathcal{N}^{\mathcal{K}'}, P^{\mathcal{K}'}, \gamma^{\mathcal{K}'})$ can be computed in $\mathcal{O}(|\mathcal{C}^{\mathcal{K}}| + |P^{\mathcal{K}}|)$ time.

Proof: Combining the Definition 6.1.10 with Proposition 6.1.5. $\Box$

For hierarchical clusterings, we obtain the result:

Theorem 6.1.12 Given a netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$, a clustering $(\mathcal{K}, \varkappa)$ of $\mathcal{C}$, and an induced clustered netlist $(\mathcal{C}^{\mathcal{K}}, \mathcal{N}^{\mathcal{K}}, P^{\mathcal{K}}, \gamma^{\mathcal{K}})$. For a clustering $(\mathcal{K}', \varkappa')$ of $\mathcal{C}^{\mathcal{K}}$ let $(\mathcal{C}^{\mathcal{K}'}, \mathcal{N}^{\mathcal{K}'}, P^{\mathcal{K}'}, \gamma^{\mathcal{K}'})$ be the induced clustered netlist. Let $\mathcal{K}''$ be the branching $\mathcal{K}'$ in which every leaf has been replaced by the corresponding arborescence in $\mathcal{K}$ and $(\mathcal{C}^{\mathcal{K}''}, \mathcal{N}^{\mathcal{K}''}, P^{\mathcal{K}''}, \gamma^{\mathcal{K}''})$ the clustered netlist of $(\mathcal{C}, \mathcal{N}, P, \gamma)$ induced by $(\mathcal{K}'', \varkappa)$. Then there is a one-to-one correspondence between the netlist $(\mathcal{C}^{\mathcal{K}''}, \mathcal{N}^{\mathcal{K}''}, P^{\mathcal{K}''}, \gamma^{\mathcal{K}''})$ and $(\mathcal{C}^{\mathcal{K}'}, \mathcal{N}^{\mathcal{K}'}, P^{\mathcal{K}'}, \gamma^{\mathcal{K}'})$.

Proof: For root cell clusters r, r', r'' in $V(\mathcal{K}), V(\mathcal{K}')$ and $V(\mathcal{K}'')$, respectively let $\lambda(r)$,$\lambda(r')$ and $\lambda(r'')$ denote the corresponding set of leaves in $V(\mathcal{K}), V(\mathcal{K}')$ and $V(\mathcal{K}'')$. The number of roots in $\mathcal{K}'$ and $\mathcal{K}''$ is the same. For a root cluster cell $c_{r'} \in \mathcal{C}^{\mathcal{K}'}$ with root $r' \in V(\mathcal{K}')$, we have:

$$\text{size}(c_{r'}) = \sum_{c_r \in \lambda'(r')} \text{size}(c_r) = \sum_{r \in \lambda'(r')} \sum_{c \in \varkappa^{-1}(\lambda(r))} \text{size}(c) = \sum_{c \in \varkappa^{-1}(\lambda''(r''))} \text{size}(c) = \text{size}(c_{r''}) \tag{6.11}$$

for some root cell cluster $c_{r''}$ in $\mathcal{C}^{\mathcal{K}''}$. In the same way x_{size} and y_{size} can be deduced. Thus, $\mathcal{C}^{\mathcal{K}'}$ and $\mathcal{C}^{\mathcal{K}''}$ are equivalent. For $p \in P$ with $\gamma(p) = \Box$, we have for the corresponding pins $p^{\mathcal{K}'} = p^{\mathcal{K}} = (\{p\}, 1) = p^{\mathcal{K}''}$ and $\gamma^{\mathcal{K}'}(p^{\mathcal{K}'}) = \Box = \gamma^{\mathcal{K}''}(p^{\mathcal{K}''})$, so consider pins with cells. A pin $p^{\mathcal{K}'} \in P^{\mathcal{K}'}$ is obtained from clustering of $p_1^{\mathcal{K}}, \ldots, p_m^{\mathcal{K}}$ with $p_i^{\mathcal{K}} = (\pi(p_i), \varrho(p_i))$ for $i = 1, \ldots, m$. For $p_i^{\mathcal{K}}$ let $N_i \subset N$ be the set of pins merged to $p_i^{\mathcal{K}}$. Then $N_i := \varrho(p_i)$, $i = 1, \ldots, m$. By branching composition, for the root $r' \in V(\mathcal{K}')$ of $\varkappa'(\gamma^{\mathcal{K}'}(p^{\mathcal{K}'}))$, the set of pins in P merged to the common root r' is thus the same as the set of pins merged to r'', where $r'' \in V(\mathcal{K}'')$ is the root of the arborescence in $\mathcal{K}''$ in which the leaves of r' have been replaced by the corresponding arborescences in $\mathcal{K}$. Thus,

$$p^{\mathcal{K}'} = \Big(\bigcup_{i=1}^{m} \pi(p_i), \sum_{i=1}^{m} \varrho(p_i)\Big) = \Big(\bigcup_{i=1}^{m} N_i, \sum_{i=1}^{m} |N_i|\Big) = p^{\mathcal{K}''}. \tag{6.12}$$

Collecting the pins net-wise from each net $N \in \mathcal{N}$ leads by the same argument to the equality $N^{\mathcal{K}'} = N^{\mathcal{K}''}$, and in the following $P^{\mathcal{K}'} = P^{\mathcal{K}''}$. For $p \in P$, let $p^{\mathcal{K}} \in P^{\mathcal{K}}$ be the corresponding clustered pin w.r.t. $(\mathcal{K}, \varkappa)$. Let r' be the root of $\varkappa'(\gamma^{\mathcal{K}'}(p^{\mathcal{K}'}))$ and $c_{r'}$ the corresponding root cluster cell in $\mathcal{C}^{\mathcal{K}'}$. Again, by the construction of the branching, $\varkappa(\gamma(p))$ is merged to the root $r'' \in V(\mathcal{K})$, which corresponds to r'. Using (6.11) we have $\gamma^{\mathcal{K}''}(p) = c_{r''}$, and this concludes the proof. □

This property applies to several hierarchy levels:

Corollary 6.1.13 Given a collection of hierarchical netlists $(\mathcal{C}^{\mathcal{K}^i}, \mathcal{N}^{\mathcal{K}^i}, P^{\mathcal{K}^i}, \gamma^{\mathcal{K}^i})$, $i = 1, \ldots, h$, where $(\mathcal{C}^{\mathcal{K}^{i+1}}, \mathcal{N}^{\mathcal{K}^{i+1}}, P^{\mathcal{K}^{i+1}}, \gamma^{\mathcal{K}^{i+1}})$, $i = 1, \ldots, h-1$, was induced by a clustering $(\mathcal{K}^{i+1}, \varkappa^{i+1})$ of $\mathcal{C}^{\mathcal{K}^i}$ and $(\mathcal{C}^{\mathcal{K}^1}, \mathcal{N}^{\mathcal{K}^1}, P^{\mathcal{K}^1}, \gamma^{\mathcal{K}^1})$ is the unclustered netlist. Let $(\mathcal{K}, \varkappa^{\mathcal{K}^1})$ be the clustering of $\mathcal{C}^{\mathcal{K}^1}$ obtained by replacing all leaves in $\mathcal{K}^{h-1}$ by the roots of $\mathcal{K}^{h-2}$, then all new leaves by the roots in $\mathcal{K}^{h-2}$, and so on. Then there is a one-to-one correspondence between $(\mathcal{C}^{\mathcal{K}^h}, \mathcal{N}^{\mathcal{K}^h}, P^{\mathcal{K}^h}, \gamma^{\mathcal{K}^h})$ and $(\mathcal{C}^{\mathcal{K}}, \mathcal{N}^{\mathcal{K}}, P^{\mathcal{K}}, \gamma^{\mathcal{K}})$.

Proof: By induction, applying Theorem 6.1.12. □

Corollaries 6.1.11 and 6.1.13 show that for constructing new clustered netlists one has to consider only one hierarchy level and not the finest netlist. This fact is interesting for speeding up the (clustered) netlist construction, when hierarchical clusterings are modified, for example when some large clusters are resolved and the netlist has to be reconstructed.

6.2 Implementation

Separating data structures from algorithms is self-evident. There are different implementations of placement netlist data — see e.g. Xi (Muuss [1996]), LEFDEF, OA (Si2 [2009]) or the bookshelf data format proposed by Caldwell, Kahng and Markov [1990], the latter also used for the ISPD benchmarks (see Nam et al. [2005] and Nam et al. [2006]). For the easy portability, the programs should be designed to be able to migrate different data structures into algorithms quickly.
Apart from different external data representations, placement also has to deal with heterogeneous objects internally, as mentioned in Section 6.1.1. The different cell types (circuits, fragments, clusters), for example, have many different properties but only few of them actually play a role in placement. Moreover, the global placement is basically interested in the location, the orientation, the sizes, and the pins.
Therefore it is reasonable to hide implementation behind homogeneous interfaces, as already done for the formal netlist definitions in Chapter 2.

The new BonnPlace global placement algorithm has been implemented in C++, which innately offers the concept of virtualisation. Virtual classes help to cope with both the internal heterogeneous objects and the external data representations. Although virtual function calls have a certain runtime overhead (see e.g. Driesen and Hölzle [1996]), we choose the virtualisation method to handle different object types which occur simultaneously during placement because of the clarity of the interfaces and easy extendability of the models.

6.2.1 Netlist and Clustering

The base components of the netlist, according to the Definition 2.2.1, consist of a cell, a pin and a net class. These classes define the interfaces and are purely virtual. The implementations and instantiation of circuits, hard clusters, physical pins as well as signal nets base on Xi (Muuss [1996]) — the corresponding classes act as wrappers. Other objects, such as root cluster cells, the induced clustered pins as well as user-defined anchors and nets are implemented independently and store their data explicitly.
Unlike Xi and LEFDEF, in BonnPlace there is no explicit mechanism for storing common data for different instances of the same circuit or pin (*CktDef, DEFs and PinDef*). This might be beneficial, especially for pins, as soon as pin areas play a role in placement. But for cells, several parts of the common data for different instances of the same DEF, such as bounding-box sizes and area demand, are subject to manipulation and require instance-dependent storage anyway. From a placement point of view, other common data should be and are indeed grouped: pointers to movebounds, cell tables and physical shape information (for non-standard cells). All cells and net implementations store pointers to their pins in arrays for caching reasons.
The clustering fully supports laminar hierarchies as described in Section 6.1.2. For implementation details we refer to Lauff [2010].
Netlists can be manipulated within BonnPlace. The bounding boxes of cells can be expanded and cell fixations can be defined. Singular pins or nets can be ignored during placement. Net weights can be set for x- and y-parts independently and artificial nets can be defined. The latter allow a combined approach of clustering and anchors. Several cells are connected by highly weighted artificial nets and hidden in a common cluster. Then, when the cluster is resolved, the artificial nets keep the cluster members together.

6.2.2 Domain

The cells of the netlist have to be placed into the rectangular chip area, subject to density, movebound and blockage constraints. Movebound areas can consist of sets of rectangles and movebound areas are allowed to overlap. Density constraints are provided in the form of area with a density target in this area. It is reasonable to provide a common density target and store the deviations for performance reasons: in many cases the density target function is constant.
All this information is area-dependent, and typical queries during placement ask for the capacity or movebounds in a given rectangle. Usually there are up to a few millions of blockages and up to hundreds of movebound areas. To enable fast queries, we use QuadTree structures (see e.g. Samet [1984]) to store rectangles. We combine all these data to the common abstract class, the *domain*. In our implementation, it would also be possible to provide several different domains on a single chip and perform placements independently for

a given netlist partition on each of them. This approach might be interesting for incremental placement.

6.2.3 Grid, Windows and Regions

The implementation of the netlist and the domain is in principle independent from the partitioning-based approach of BonnPlace. We focus on the partitioning-specific data structures now. Recall the overall structure of BonnPlace from Section 3.4 and the partitioning-based movebound placement algorithm from Section 4.3.
The sequence of the refined grids $\Gamma_0, \dots, \Gamma_k$ is currently determined by recursive, mainly equidistant, cuts. During level $i = 1, \dots, k$ one considers the set of windows induced by Γ_{i-1}. Each window has to store the associated cells. A level ends with a feasible partitioning to windows induced by the grid Γ_i. Given $i = 1, \dots, k$ we refer to Γ_{i-1} as the *coarse grid* and to Γ_i as the *fine grid*. In level $i+1$, Γ_i then becomes the coarse grid. In real-world instances, the finest grids induce up to several millions of windows. A coarse window does not store any particular data except a unique key to identify its coordinates, and an array containing associated cells.
Fine windows are destinations for cell partitioning. For this purpose, fine windows store the set of their regions, and the regions on their part provide all information relevant for partitioning: the capacity, the area and the density information. Blocked windows are useless, thus in BonnPlace, we store regions for unblocked windows only. The regions are chosen to be maximal w.r.t the movebounds and the windows. In particular, in instances without movebounds there is one region per fine unblocked window.
During level $i = 1, \dots, k$, after macro placement has been performed w.r.t. Γ_{i-1}, possibly blocking some areas, the domain does not change until the macro placement of level $i+1$. For these reasons, we compute and store the regions for the entire level after MacroPlacement. In addition, the center-of-gravity of the free area in the windows will also remain unchanged for this level and is thus also cached. Of course the partitioning to the fine windows can change. But then, only the cached values at the regions have to be adjusted without domain queries. This is particularly important for repartitioning, where millions of such queries can be saved during late levels compared to the old BonnPlace.

6.2.4 QP

The quadratic programs are time and memory consuming parts in BonnPlace. On the largest instances, the dimension of the involved matrices easily exceeds 10 million and the number of entries in the matrix would reach over to 200 million if one applied the clique model to large nets. Thus we use the hybrid clique-star model as proposed by Brenner and Struzyna [2005]. In the old BonnPlace the star net model was applied to nets with eight and more terminals, where smaller choices led to runtime loss. Our new implementation allows to lower the threshold to nets with five and more terminals without runtime degradation. In theory, however, the dimension of the Krylov space increases and

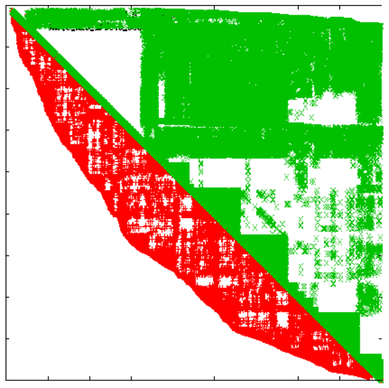


Figure 6.1: Nonzero entries of the original matrix A (green) and the reordered ones (red). As A is symmetric, the upper half is depicted for the original and the lower for the reordered one.

the conjugate-gradient (CG) algorithm could need more iterations (see Stoer and Bulirsch [2002]).

For clustered netlists, of course, the clustered net models from Definition 6.1.6 and the induced clustered hybrid net model are used.

During the CG algorithm, when solving the system $Ax = b$ the most time consuming operation is the matrix-vector multiplication Ap for some vector p. In our case, A is sparse and stored in the compressed-row-storage (CRS) format. In theory, the complexity of Ap depends on the number of non-zero entries of A, but in practice, small bandwidths reduce the risk of cache misses and accelerate the process. For bandwidth reduction, a popular method is to interpret the matrix as the adjacency matrix of an undirected graph, and to relabel the columns and rows w.r.t. the distance in the graph (Cuthill and McKee [1969]). The minimum bandwidth corresponds to the graph diameter. In our case, we are given a graph initially, so permutation can be done before matrix construction. Because of the low number of iterations of the CG algorithm in our case, we perform an approximated diameter search by applying BFS twice. This method is only beneficial for large QPs in early levels and saves up to 10% of the CG runtime in those cases. Figure 6.1 shows an example of an original and a reordered matrix.

6.2.5 Clustering

BonnPlace is able to handle external hierarchical clusterings. As clusters, either a set of cells with relative offsets is provided, which yields a hard cluster cell, or the cell group does not have offsets and it is interpreted as soft cluster. Hard clusters or cells are leaves in the clustering. For soft clusters, beside the set of clustered cells, a *resolve inertia* is specified which defines the ratio of the cluster and window size for resolving clusters. A resolve inertia of 70% means that the cluster will be resolved as soon as the corresponding root cluster cells attain 70% of the window size. However, clusters can be resolved earlier, if they contain wide or high cells to make such circuits visible to the macro placement. In

hierarchical clusters, the parent node's resolve inertia is set to the minimum of its inertia and the minimum of the inertias of its children. The default value is 5%.
After external clustering, BonnPlace clusters spare logic groups. Finally, internal clusterings are also used, see Chapter 7.

6.2.6 Partitioning

The ParallelFlowBasedPartitioning algorithm presented in Chapter 5 is used as the new global partitioning routine. For MinCostFlow computation we use a Network-Simplex implementation. By default, from level 8 on we use the old recursive partitioning because the grids become large, and the graph construction as well as the MinCostFlow computation begin to be expensive, especially for tiny and mid-size designs. Moreover, tests have shown that no significant netlength improvement is obtained when using the flow-based partitioning in late levels.

6.2.7 MacroPlacement

Large blocks are fixed in a legal position during macro placement when the cell's width or height becomes too big for a given window and a reasonable movement is not possible. To this end, legalization data structures are required and one has to make MacroPlacement aware of clustering.
For this purpose, we have implemented legalization data structures for every cell with determined shapes (circuits, fragments, hard cluster cells), generalizing the simple circuit row concept. Legalization data consists of a distinguished point, the anchor, and a set of grids with orientations. A cell is placed legally if it does not overlap any other cell or blockage and the anchor is one of the grid points and the cell orientation equals the grid orientation.
This model allows us to handle standard circuit rows, gate array circuit rows and special cell tables. It could also handle different grids in different areas of the chip, with application to voltage islands or flat placement of originally hierarchical netlists. For clusters, recall that clusters store the x_{size} and y_{size} values but do not provide shapes in general. Thus, before macro placement, the clustering hierarchy is resolved until all clusters' x_{size} and y_{size} values are smaller than the maximum movable cell sizes for this window. This reveals all macros and hard cluster cells that are important at this point. Macros and hard cluster cells are currently fixed in a greedy manner, as in the old BonnPlace.

6.2.8 Congestion-Driven Placement

In congestion driven placement, fast global routing is used to estimate routability. Pin density plays an essential role, which for performance reasons could also be determined from the clustered netlist using the pin multiplicities and cluster sizes. On the other hand, routing within a window can only be determined if the precise pin-net mapping is known.

In our opinion, it is reasonable to reveal the fine netlist to the routing estimation but use this to update the clustered cell sizes. When clusters are resolved, we distribute the parent node's inflation value to its children proportionally to the children's pin densities.

6.2.9 Legalization and Local Placement

We use the legalization algorithm of Brenner and Vygen [2004] to legalize the outcome of global placement with the modification to movebounds as proposed in Section 4.3.6. In addition, we use a simple local placement heuristic which allows us to exchange cells and move cells to a free area if it improves the netlength and does not increase density violations.

6.3 Parallelization

The improvement of processor frequencies has practically stalled during the last decade, while the number of cores per processor and the number of processors per system is still increasing. Parallelization is thus a formidable opportunity for performance gain, in particular for *partitioning* based placement, where data partition arises in natural manner.

Parallelization on UNIX and Linux systems can be achieved by creating new processes using the *fork* command. The decisive disadvantage of this method is the fact that for each fork call, a complete memory copy of the caller process is done. This can lead to exorbitant memory demand on VLSI instances. On instances with movebounds, as in Table 8.16, the industrial used the fork command and would have needed more than 220 GB (with four processors only). For this reason, shared-memory parallelization is the method of choice and we use pthreads (see Buttlar, Farell and Nichols [1996]) for this purpose. BonnPlace is compiled on different platforms, used on different systems, and interacts with several other tools. Hence, processor specific parallelization (see Intel TBB [2010]) or recent multithreading libraries as summarized by Hoffmann and Lienhart [2008] are not an option yet.

Beyond performance aspects, another issue is important for practical use, namely the *determinism* or at least the *repeatability*. Without entering into formal descriptions, for us the determinism postulates the same output on the same data, even on different machines (of the same type) and without paying attention to the number of processors used. The weaker repeatability requirement asks for the same outcome given the same input on the same machine type with the same number of processors. For software engineering purpose it would in fact be enough to be able to reproduce a run, possibly using some information from the previous run.

6.3.1 Common vs. Specific Data

The objects manipulated by placement are the cells. Cell movement induces pin movement, density and window assignment. A single cell coordinate cannot be manipulated by several processors simultaneously, but we allow the x coordinate to be manipulated independently from the y coordinate. New cell locations are based on the locations of other cells. In a multithreading environment, this was basically the reason for the undesired non-repeatable behavior of the old BonnPlace. To avoid such situations, we store additionally *safe coordinates* $(x_{\text{safe}}, y_{\text{safe}}) \in \mathbb{R}^2$ for each cell, and two *thread indices* (one for the vertical and one for the horizontal thread). Before a parallel code fragment begins, safe coordinates are stored. Then, cells are partitioned to threads and each thread stores indices to its cells.
Threads have exclusive write access for their own cell coordinates. For a cell $c \in \mathcal{C}$, reading a cell's horizontal coordinate by a thread π means to ask if the cell's horizontal thread index equals π and to return $x(c)$ if the answer is "yes" and $x_{\text{safe}}(c)$ else.
The memory overhead per cell is thus constant and independent from the number of threads used, in contrast to the old version of BonnPlace where an array of indices was stored per thread.
Another issue deals with the memory management itself. Major algorithmic blocks are called in parallel using threads. Each thread comes with its own memory management and uses a memory heap for its own allocations. The common netlist, domain and grid data is stored using a thread-safe memory manager to allow simultaneous memory allocation and releases. Currently, the system's malloc and free routines are used for this purpose.

6.3.2 Job Queues

For parallelization of core routines in BonnPlace, we hence need a partition of $\mathcal{C}$ to threads. We use the given cell-window partitioning of cells $\mathcal{C}$ induced by the grid Γ_i for some level $i = 1, \dots, k$.
With the hybrid net model, the global QPs (both the constrained and unconstrained ones) can be computed independently for each row (column), where a row (column) is a collection of windows induced by Γ with common y- (x-) coordinates, by terminal propagation, as already proposed by Brenner and Struzyna [2005]. For recursive partitioning, the partitioning decision in window W induced by Γ_i to its subwindows induced by Γ_{i+1} does not depend on other windows either.

Algorithm 10: RunJobsInParallel($\mathbb{F}, \mathcal{J}, \Pi$)

```
1 for $\pi = 1, \dots, \Pi$ do
2     StartThread ($\mathbb{F}, \mathcal{J}, \pi$)
3 for $\pi = 1, \dots, \Pi$ do
4     Wait ($\pi$)
```

Procedure `StartThread(`$\mathbb{F}, \mathcal{J}, \pi$`)`

```
1 while $\mathcal{J} \neq \emptyset$ do
2     Choose $J \subset \mathcal{J}$
3     Set $\mathcal{J} := \mathcal{J} \setminus J$
4     for $j \in J$ do
5         $\mathbb{F}(j, \pi)$
```

A schematic example is provided in Algorithm 10, where the input consists of a set of jobs $\mathcal{J}$, a function $\mathbb{F}$ working on each job, and a number of processors Π. The algorithm instantiates Π threads, and in each thread, as long as there is some job to be done, a subset of jobs is chosen and $\mathbb{F}$ is performed on each of them. The critical section (lines 2 and 3) in Procedure `StartThread` is protected by *a mutex* to avoid race conditions. As other threads may wait for the mutex, this critical section has to be efficient. For this reason we basically store only pointers in $\mathcal{J}$ and implement $\mathcal{J}$ as an array. In recursive partitioning, for example, the job queue $\mathcal{J}$ consists of coarse windows. $\mathbb{F}$ is then responsible for $j \in \mathcal{J}$ to pick all cells from window W with index j of Γ_i, all regions in the subwindows of W induced by Γ_{i+1} and perform a partitioning step. The partitioning is stored *directly* to the fine subwindows, which are indeed disjoint for different threads, but may trigger memory allocations as the cell arrays might have been expanded. The latter is the reason for the thread-safe memory manager for common data.
The repartitioning is more complicated. In level i, let us assign a pair of indices (α, β) to each window w induced by the level's fine grid Γ_i, where α (β) is the column (row) index of w in Γ_i. For fixed $a, b \in \mathbb{N}$, repartitioning then considers all coarse $a \times b$ windows in Γ_i. We propose performing repartitioning in *phases* (p, q), $p = 1, \ldots, a-1$, $q = 1, \ldots, b-1$, where a phase consists of all coarse $a \times b$ windows whose lower-left window with indices (α, β) satisfies $p = \alpha \bmod a$ and $q = \beta \bmod b$. Safe locations are updated after each phase.

Lemma 6.3.1 Let Γ_i and the fine window indices, $a, b \in \mathbb{N}$ be as above. Then two different coarse $a \times b$ windows from one phase do not have a fine window in common.

Proof: For two different coarse $a \times b$ windows W, W'. let (α, β) and (α', β') be the indices of their lower left fine windows. W.l.o.g., assume $\alpha < \alpha'$. Assume the phase considers horizontal windows with indices satisfying $p = \alpha \bmod a$ for some $p = 1, \ldots, (a-1)$. But then, as α and α' are both from the same phase, we have $la + p = \alpha < \alpha' = l'a + p$ for some $l, l' \in \mathbb{N}$, $l < l'$. This means that the rightmost fine window in W has index $la + p + (a-1)$ and $la + (a-1) + p < (l+1)a + p \leq l'a + p = \alpha'$ and is indeed smaller than any index of leftmost windows in W'. Similarly for vertical differences. $\square$.

Using Lemma 6.3.1 we are able to perform repartitioning in parallel in each phase:
Although there is no guarantee for deterministic behavior due to compiler optimization in the presence of floating-point arithmetic, we do observe deterministic behavior of the parallelized routines, even on different AMD and Intel machines running with different

Algorithm 12: ParallelRepartitioning(a, b, Γ, Π)

```
1  StoreSafeLocations
2  Let $\mathcal{W}$ be the set of windows induced by $\Gamma$
3  for $p = 1, \dots, a-1$ do
4      for $q = 1, \dots, b-1$ do
5          $\mathcal{J}$:= CollectWindows ($p, q, \mathcal{W}$)
6          for $\pi = 1, \dots, \Pi$ do
7              StartThread (Repartitioning,$\pi, \mathcal{J}$)
8          for $\pi = 1, \dots, \Pi$ do
9              Wait ($\pi$)
10         StoreSafeLocations
11 RunJobsInParallel (QP($\Gamma$), $\mathcal{J}, \Pi$)
```

Linux operating systems. One can ask if working with stored locations in place of the real and permanently updated ones has any impact on the netlength. Tests have shown that the degradation was less than 0.2% on average, making this approach well worth it.

6.3.3 A-priori Job Assignment

Another option for parallelization is to partition the set of cells a priori and assign each chunk to a thread. This method is preferable in cases where the execution times are relatively small and the cost for job queue maintenance is not justified. This is the case for flip code optimization FlipOpt, in which an orientation is chosen greedily for each cell in order to minimize the wirelength. This is also the case for greedy re-allocations (see Section 5.3.2). For repeatability reasons safe locations are used as above for non-thread cells. In order to maintain determinism, the job-thread assignment could be made independently from the number of threads used.
Of course, any other time consuming routines such as the computation of netlength, center-of-gravities, etc. can also be parallelized.

6.3.4 Speed-ups

Here, we present speed-up charts for the major routines in BonnPlace on different machines and different levels, exemplarily for the chip Erhard with movebounds. The Figures 6.2, 6.3 and 6.4 provide the speedup factors over the sequential run for the realization in flow-based partitioning (cf. Section 5.2.3), the global QP and repartitioning respectively. On the x-axis the level numbers are shown and the y-axis shows the speedup. Different colors represent different numbers of cores.
In all these tests one can see that, initially, the low number of jobs for the coarse grids results in low speedups. For the intermediate and late levels, the speed-ups become much

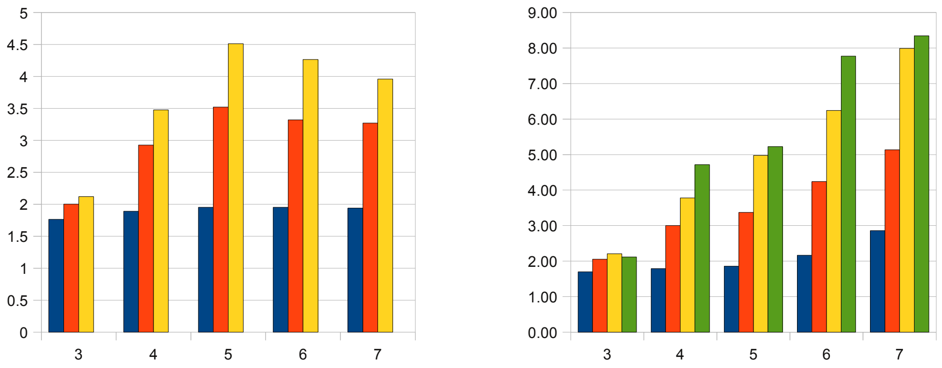


Figure 6.2: Speed-up of flow-based partitioning realization on Intel Xeon X5365 (2× Quad Core) (left) and AMD Opteron 8222 (8× Dual Core) (right), using 2, 4, 8 and 16 cores simultaneously.

better but finally in the latest levels the memory access dominates the computational effort and the speed-ups get worse. Differences between different machine types are remarkable. For the global QPs the speedup exceeds 5 with 8 cores on the Intel machine, on the AMD the speedups attain even 7 with 8 and almost 10 with 16 cores. All jobs during global QP are pretty memory consuming, so from a certain point on, memory access becomes a bottleneck.

Another fact is also interesting: for jobs which contain lots of queries to the *common* memory, the AMD's non-uniform-memory-architecture (NUMA) pays off, as cache can be shared among processors. The repartitioning speedups on Intel attain the factor 7, whereas on the AMD machine the speedups attain as much as 32.2 (!) in level 8 using 16 cores. During repartitioning, access to cells outside the window are frequent for QP matrix constructions and netlist evaluations. But these cells can be processed by another processor at the same time, and large parts of the data can be found in the caches, making slow off-cache memory access unnecessary. The speedups often exceed the number of processors used, as shown in Figure 6.4. Such tendencies already show up on AMD in flow-based partitioning (cf. Fig. 6.2), where the speedups are several times higher than the number of cores. In repartitioning, the probability is higher that proximate windows are processed at the same time and memory queries can be satisfied by the data stored in cache.

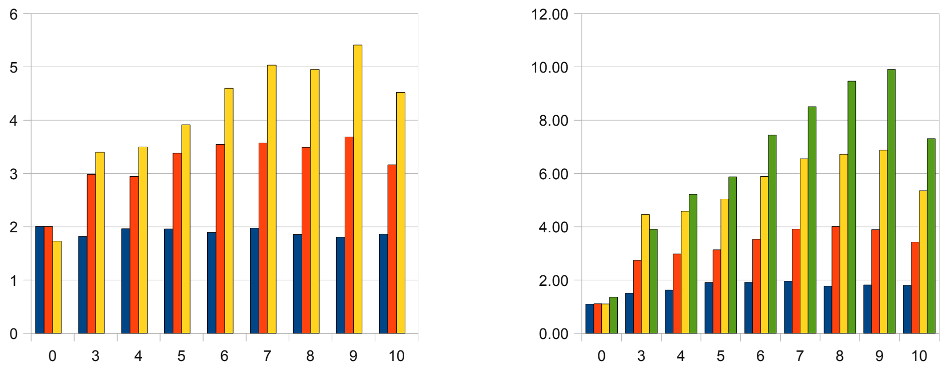


Figure 6.3: Speed-up of global QP on Intel Xeon X5365 (2× Quad Core) (left) and AMD Opteron 8222 (8× Dual Core) (right), using 2, 4, 8 and 16 cores simultaneously.

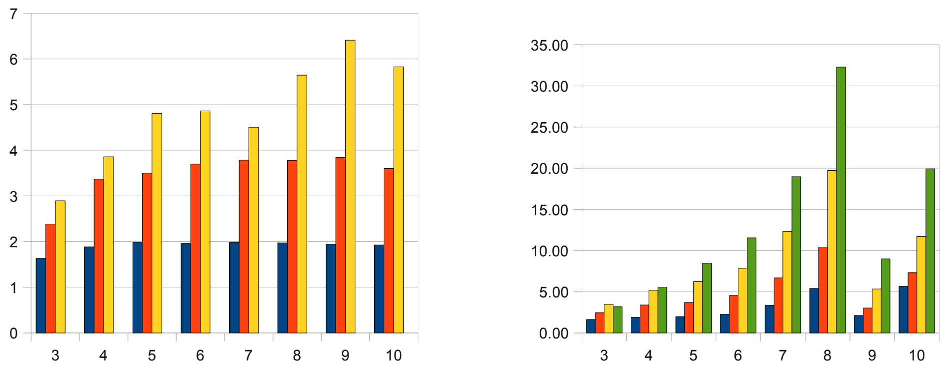


Figure 6.4: Speed-up of repartitioning on Intel Xeon X5365 (2× Quad Core) (left) and AMD Opteron 8222 (8× Dual Core) (right), using 2, 4, 8 and 16 cores simultaneously.

Chapter 7

Random Walk based Clustering

Clustering, which appears in different contexts and for various purposes, subsumes an entire class of combinatorial optimization problems, which ask, colloquially spoken, for groups of similar nodes in (hyper-)graphs or metric spaces. In VLSI placement, clusterings have been used for almost two decades to reduce the instance sizes and to compensate placement inaccuracies. In this chapter we summarize the recent developments in clustering within VLSI placement first. We then commit ourselves to random walks in VLSI placement and show how global random walk information from large hypergraphs can efficiently be retrieved. Finally, we use the random walk information for bottom-up clustering.

7.1 Clustering in VLSI Placement

An instance of the classical clustering problems consists of a finite set X, a distance function $d : X \times X \to \mathbb{R}$ and integers k, l. One problem asks for a partition of X into disjoint sets $X_1, \dots, X_k$, such that for $1 \leq i \leq k$, all pairs $x, y \in X_i$ have $d(x, y) \leq l$. This problem is $\mathbb{NP}$-complete (see Garey and Johnson [1979]). But this is not the only bad news. Let f denote a function which, for a given (X, d), returns a clustering of X. Kleinberg [2002] showed that if one postulates relatively natural properties at f, such as richness (i.e. by manipulation of d, every clustering is possible), scale invariance (multiplying d by a constant leads to the same clustering) and consistency (decreasing distances within and increasing distances outside clusters does not coarsen any clusters), then we find that f does not in fact exist.
Beyond these theoretical results, clusterings in VLSI placement, unlike in other domains such as pattern recognition, structure detection in networks, and general information retrieval do not end in themselves but are used to improve the placement performance — the runtime and the quality of result. To this end, classical formulations of clustering and clustering quality measures are of subordinated importance — the evaluation has to be done in terms of the netlength and runtime of the placement. Clusters are desirable which are formed out of highly connected parts of the netlist. At the same time, the computational effort for clustering is limited.

As initially mentioned in Section 6.1.2, the use of clustering is common in VLSI placement. Basically, two concepts can be distinguished here. The first one is the top-down clustering, which can also be seen as a particular graph partitioning problem: initially all cells form one big cluster and the clusters are subsequently divided into smaller ones. Such approaches are used, for example, in Dragon (Wang, Yang and Sarrafzadeh [2000]) and FengShui5 (Agnihotri, Ono and Madden [2005]).
For top-down clustering, traditional Min-Cut approaches as proposed by Fiduccia and Mattheyses [1982], multilevel-bisection (see Metis [1998]) and spectral methods (see Hall [1970], Kucar and Vannelli [2006]) are used. The advantage of top-down clusterings is the global view, and the major drawbacks of these methods are the long runtimes and that this concept does not seem to match well with top-down partitioning analytic placement schemes. Large clusters would have been resolved immediately to allow reasonable movement within the windows, so a deep hierarchy was necessary and most of this hierarchy structure would have been dropped.
Therefore, the other class of clusterings, the bottom-up methods, are well established in placement. In bottom-up clusterings, each cell initially forms a cluster which grow during clustering. Many different algorithms follow this idea: various edge, hyperedge and clique coarsenings based on matchings or single arc contraction (see Alpert, Huang and Kahng [1997] for *Heavy-Edge-Matching*, Karypis and Kumar [1999] for *Edge-Coarsening*, Li and Behjat [2006] for *NetCluster*, Tsota, Koh and Balakrishnan [2009] for *NetCluster* combined with a MinCut heuristics, and Karypis and Kumar [2000] for the *FirstChoice*-algorithm). Outside of the placement context, Osipov and Sanders [2010] present a single-edge coarsening approach using fast data-structures which allow multilevel clustering with contraction of a single edge at one level.

The choice of clustering partners in the algorithms above was made either randomly or using maximal matchings. Other authors propose to use some rank function and a heap structure in an iterative process. Such an approach is used in the *FineGranularity* clustering (see Hu, Zeng and Marek-Sadowska [2005]) and *Edge-Separability* (Cong and Lim [2004]). The heap stores edges with appropriate weights. Nam et al. [2006] and Yan, Chu and Mak [2010] propose storing a cluster in the heap together with its “best” neighbor. The last three papers are worth having a closer look at.
Most of the bottom-up clustering algorithms only have a local view and thus rank pairs of clusters by their direct connections, with and without size consideration. Cong and Lim [2004], however, try to incorporate more global information into the model. To this end, they use the approximated *edge separability* values, which for an edge provide the approximated minimum cut value separating the edge's end nodes. It would be too time consuming to compute exact minimum cut values, so the authors use a fast heuristics, which yields lower bounds. The obtained values are transformed into edge ranks by division of the approximated separability values by the minimum of the node degrees incident to the edge.

The *BestChoice* algorithm (Nam et al. [2006]) has successfully been promoted in the

VLSI placement community for the last four years. Despite its simplicity, the results obtained by the tool were for the first time predominantly better than the results without clustering (see also Table 8.20). In this algorithm, a heap is maintained containing a triple $\big(C, \text{bestneighbor}(C), \text{rank}(C, \text{bestneighbor})\big)$, where C is a cluster (initially each cell forms a cluster) and for two clusters C, C' the value

$$\text{rank}(C, C') := \frac{1}{\text{size}(C) + \text{size}(C')} \sum_{N \in \mathcal{N}:\ C, C' \in N} \frac{\omega(N)}{|N|} \tag{7.1}$$

acts as heap key (by abuse of notation $C \in N$ means that there exist a cell c in the cluster C and a pin p with $\gamma(p) = c$ and $p \in N$). A major ingredient in *BestChoice* is the idea of *lazy update*: most of the clustering operations do not increase the key values, so clustered nodes are only labeled as "changed" and the rank computation is reduced to the necessary one. This helps to accelerate the clustering process. The clustering halts when the number of clusters reaches the prescribed percentage of original cells.
In a recent publication by Yan, Chu and Mak [2010] the authors directly address netlength minimization within clustering. In their method, a cluster is called *safe* if its members can move to the same location without netlength degradation. Then, a criterion for pairs of cells is presented, which basically evaluates the mutual (local) movement of two cells towards each other. If such a local movement does not make the netlength worse in *any* placement, it is safe to cluster the pair. As it is intractable to enumerate all placement, the authors significantly break down the number of enumerations to a small number. Finally, the enumerated netlength gradients are combined with cluster sizes to a rank function and a heap is used, similarly to the *BestChoice* approach.

Currently, almost all leading-edge placements tools use the *BestChoice* algorithm: RQL (Viswanathan et al. [2007]), APlace (Kahng and Wang [2006]), mPL6 (Chan et al. [2006]), FastPlace3 (Viswanathan, Pan and Chu [2007]). NTUPlace3 (Chen et al. [2005]) uses *FirstChoice*. CAPO (Roy et al. [2005]) and FengShui5 (Agnihotri, Ono and Madden [2005]) make use of heavy-edge matching, and finally Vaastu (Agnihotri and Madden [2007]) uses a geometric approach, in which cells are clustered if they belong to the same grid window. In BonnPlace we have also implemented the *BestChoice* and the *NetCluster* algorithms, the comparison of which unambiguously showed better results with *BestChoice*. On the other hand, *BestChoice* did not achieve the results reported by Nam et al. [2006] with BonnPlace: only low clustering ratios lead to similar netlengths (see also Tables 8.6 and 8.20). A clustering ratio of $\alpha = 5$ already led to inferior results of 2% on average. For a more detailed description and more results, we refer to Lauff [2010], who also shows examples where *BestChoice* does not detect natural clusters, even in simple cases.

7.2 Random Walks

Consider a connected, weighted, undirected graph $G = (V, E, \omega)$. A random walk within G is a discrete-time stochastic process, in which at some node $v \in V$ the random walker decides with probability $\frac{\omega(\{v,w\})}{\sum_{u\in\Gamma(v)}\omega(\{v,u\})}$ to proceed towards $w \in V$. When considering random walks, two functions are of particular interest for us:

Definition 7.2.1 Fix $G = (V, E, \omega)$. The expected number of steps which are needed by a random walker starting at $v \in V$ to reach a node $t \in V$ is called the *hitting time* from v to t and denoted by $h(v|t)$. The *commute time* between two nodes $v, w \in V$ is denoted by $\text{commute}(v, w) := h(w|v) + h(v|w)$.

Random walks appear in different contexts in VLSI design. Vygen [2007] has shown that the quadratic placement itself can be interpreted as a random walk. Strong links between random walks and networks are established via the *effective resistance*. We summarize the most important properties in the following. For simplicity, assume $V(G) = \{1, \ldots, n\}$ and G is a connected graph and we are given weights $\omega : E(G) \to \mathbb{R}_+$. For the weighted adjacency matrix $A(G)$ of the graph G, let $D(G) := \text{diag}(d_1, \ldots, d_n)$ be the diagonal matrix with $d_i := \sum_{j:\{i,j\}\in E(G)} \omega(\{i, j\})$, storing the weights of arcs incident to node i, $i = 1, \ldots, n$. The *Laplacian* $L(G)$ is the symmetric matrix $L(G) := D(G) - A(G)$. It is well known that $L(G)$ is positive semidefinite and has rank $n - 1$. The vector $\mathbf{1} := (1, \ldots, 1)^T \in \mathbb{R}^n$ is an eigenvector with eigenvalue 0. Now, let $e_i \in \mathbb{R}^n$ define the i-th unit vector. The effective resistance between two nodes $i, j \in V(G)$ is defined as:

$$r_{\text{eff}}(i, j) := (e_i - e_j)^T L^+(G)(e_i - e_j) \tag{7.2}$$

where $L^+(G)$ is the Moore-Penrose-pseudo-inverse of $L(G)$ (A Moore-Penrose-pseudo-inverse A^+ for a real matrix A is the unique matrix which satisfies: $AA^+A = A^+, A^+AA^+ = A, (AA^+)^T = AA^+$ and $(A^+A)^T = A^+A$. For more details we refer to Barnett [1990]). For $L^+(G)$ there exists even an explicit formula (Rao and Mitra [1972]), and $L^+(G)$ acts as the inverse linear operator on the image of $L(G)$. Observe that $\langle (e_i - e_j), \mathbf{1}\rangle = 0$ for all $i, j \in V(G)$, thus r_{eff} is well defined for each pair of nodes.
The results of Klein and Randic [1993] show that r_{eff} is a distance function on graphs and, even if the graph theoretical distance is the same between nodes, the nodes with low effective resistances are connected by more short paths (see Doyle and Snell [1984], Yen et al. [2005]). The fact that effective resistances and commute times are proportional (see Chandra et al. [1989]) is another reason for capturing connectivity within the graph by random walks. For more details on relations between random walks and networks, we refer to the surveys by Lovász [1993] and Doyle and Snell [1984].
Spielman and Srivastava [2008] show in their work that arcs with high effective resistances are indeed responsible for essentials of the Laplacian as the quadratic form. Let $q_{L(G)} : \mathbb{R}^n \to \mathbb{R}$ be the quadratic form $q_{L(G)}(x) \mapsto x^T L(G)x$. For a Laplacian $L(G)$ they prove the existence of a Laplacian $\tilde{L}(G)$ such that, for all $x \in \mathbb{R}^n$, the quadratic form $q_{\tilde{L}(G)}$ induced by $\tilde{L}(G)$ is an ε-approximation of the quadratic form induced by $L(G)$:

$$(1 - \varepsilon)q_{L(G)} \leq q_{\tilde{L}(G)} \leq (1 + \varepsilon)q_{L(G)} \; \forall x \in \mathbb{R}^n \tag{7.3}$$

and $\tilde{L}(G)$ contains $\mathcal{O}(n \log n\varepsilon^{-2})$ arcs, with $\varepsilon > 0$. In this approach, arcs are sampled with probability proportional to their weights and the effective resistance between the edge's endpoints. Thus, the higher the weights and the commute times, the higher the probability of appearing in $\tilde{L}(G)$. Conversely, omitting arcs with low product of weight and effective resistance differences (which actually means to cluster the incident nodes) does not change the quadratic form significantly.

7.3 Clustering with Random Walks

7.3.1 Previous Approaches

In VLSI placement, there were two approaches used random walks for clustering in the past. Cong, Hagen and Kahng [1991], and Hagen and Kahng [1992] explicitly simulated a random walk. After simulation, Cong, Hagen and Kahng [1991] compute maximal cycles in this walk for each node. Two nodes v, w are clustered if both appear in the same maximal cycle. As extension, Hagen and Kahng [1992] relax the clustering condition and compute the *sameness* of a pair of nodes, which reflects the number of common maximal cycles in which the nodes appear. Both methods, however seem intractable for memory and runtime reasons, not to mention the determinism.
In other domains, there are several publications on clustering with random walks. Harel and Koren [2001] use a deterministic approach in which the transition probability matrix (which can directly be obtained from $L(G)$ by normalization) and its powers are used to determine similarities in the graph. These numbers are then used to re-weight edges and the authors claim that the similarity numbers tend to separate natural clusters. Fouss et al. [2007] use, among other methods, hitting and commute times for similarity computations in data bases. The authors use explicitly the formula (7.2) and by combination of projection methods with Cholelsky sparsification, the non-sparse structure of $L^+(G)$ can be by-passed and columns of $L^+(G)$ computed on demand.
Random walks based on commute times are also used for community searches in social networks (Firat, Chatterjee and Yilmaz [2007]), in image recognition Yen et al. [2005], and based on hitting times in Chen, Liu and Tang [2008] for small directed-graph benchmarks.
Now we return back to the hitting times. For a matrix $A = (a_{ij})_{i,j=1}^n$, let $A_{\setminus ij}$ denote the matrix obtained from A by deleting the i-th row and the j-th column. The decisive advantage of the hitting times, as is well known, is their easy computation:

Proposition 7.3.1 Given a weighted, connected graph $G = (V, E, \omega)$ with $V = \{1, \ldots, n\}$ and $t \in V$, consider the $n - 1$ dimensional linear system with $L = L(G)$:

$$(L_{\setminus tt})x = b, \tag{7.4}$$

where the vector $b \in \mathbb{R}^{n-1}$ is defined by the diagonal entries of $L_{\setminus tt}$:

$$b := (l_{11}, \ldots, l_{t-1,t-1}, l_{t+1,t+1}, \ldots, l_{nn})^T.$$

Then, the hitting times $h(j|t)$, $j = \{1, \ldots, n\}$ are $h(j|t) = x_j$ for $j = 1, \ldots, t-1$, $h(j+1|t) = x_j$ for $j = t, \ldots, n$ and $h(t|t) := 0$. In particular, $L_{\setminus tt}$ is symmetric, diagonally dominant, positive definite and sparse, if $L(G)$ is.

Proof: Observe that for $j \in \{1, \ldots, t-1, t+1 \ldots, n\}$ the hitting times satisfy the recursion:

$$h(j|t) = 1 + \sum_{\{k,j\} \in E} \frac{\omega(\{k,j\}) h(k|t)}{\sum_{\{l,j\} \in E} \omega(\{l,j\})}, \tag{7.5}$$

for each node $j = 1 \ldots, t-1, t+1, \ldots, n$ with $h(t|t) := 0$. Multiplying by the denominators yields the result. □

Of course several targets can be chosen simultaneously and each target reduces the matrix dimension by 1. The additional targets appear as additional diagonal and right-hand side entries in the system (7.4). In the following we consider single targets only.

7.3.2 Hitting Times in Hypergraphs

Existing random walk computations were made on graphs. In the netlist, we are given a hypergraph, and the trivial approach would be to represent hyper edges by weighted cliques (cf. (2.2)). Large hyper edges, however, would lead to quadratically many entries. Fortunately, we can use the star model again:

Theorem 7.3.2 Given a weighted, connected, hypergraph $H = (V, \mathcal{N}, \omega)$ with $V = \{1, \ldots, n\}$ and a node $t \in \{1, \ldots, n\}$ as hitting time target, let G be the graph obtained from H by replacing each $N \in \mathcal{N}$ with a clique model. Then, the hitting times in G can be computed by solving a *sparse* linear system.

Proof: Let $N \in \mathcal{N}$ be a hyperedge. We show that we can replace the clique representation of N by a star, which will consequently lead to the sparse representation of N and of G. We may assume $N = \{1, \ldots, l\}$ for some $l = 2, \ldots, n$.
Let L be the Laplacian of G, and L' the Laplacian of the graph G, where N was modeled by a star. We may assume that the star $s = s_N$ node of N has index 0. Canceling the t-th row in both, leads to the linear systems

$$(L_{\setminus tt})x - b = 0 \tag{7.6}$$

and

$$(L'_{\setminus tt}) \begin{pmatrix} x_s \\ x \end{pmatrix} - b' = 0 \tag{7.7}$$

with b corresponding to the diagonal of $L_{\setminus tt}$ and b' to the diagonal of $L'_{\setminus tt}$. Now consider the i-th rows $i = 1, \ldots, n$, $i \neq t$ of (7.6) and (7.7). Combining the i-th rows and canceling the common terms (all elements induced by hyper edges in $\mathcal{N}$ except N) one obtains:

$$\sum_{j=1}^{l} \frac{-\omega(N)}{l-1} x_j + \frac{\omega(N) l}{l-1} x_i - \omega(N) = \frac{-\omega(N) l}{l-1} x_s + \frac{\omega(N) l}{l-1} x_i - \frac{\omega(N) l}{l-1}, \tag{7.8}$$

which solving for x_s yields:

$$x_s = \frac{-1 + \sum_{j=1}^{l} x_j}{l}. \tag{7.9}$$

Now, replacing x_s in (7.7) shows: $b'_0 = \frac{-\omega(N)l}{l-1}$. Hence, the (possibly) dense system (7.6) can be replaced by the equivalent sparse system (7.7) by induction on $|\mathcal{N}|$. □

Thus, we can indeed compute hitting times by considering even large hyper edges.

7.3.3 Hitting Times and Bottom-up Clustering

Ideally, we would have directly used the commute time distances or hitting times between pairs of cells in our bottom-up clustering to measure similarities. But storing pairs of all (or at least connected) objects is too expensive, not to mention the computation runtime. On-the-fly methods for commute time computation, as proposed by Fouss et al. [2007], seem also too slow for this purpose. An option, at least for small instances, would be the approach presented by Spielman and Srivastava [2008] where an approximated commute time computation is based on a random projection to much lower spaces.

The idea behind our approach is a pretty pessimistic one: if we cannot find out which cells should be clustered, we shall at least prevent wrong clustering decisions. In the previous section, we have seen that commute times can serve as a similarity measure. For the pessimistic approach, the symmetry of commute times is not mandatory – for performance reasons we can restrict ourselves to the hitting times, using different targets.
For a pair of cells c, c', if there are differences between the hitting times $h(c|t)$ and $h(c'|t)$ to the same target t, the cells c, c' are clearly in different stronger connected groups and should not be merged into one cluster. Assuming that we have computed τ random walks in the netlist hypergraph to targets $t_1, \ldots, t_\tau$, then we store the vector $h(c) := (h(c|t_1), \ldots, h(c|t_\tau))^T \in \mathbb{R}^\tau$ for each cell $c \in \mathcal{C}$. The new rank function for clusters C, C' (see also (7.1)) is then:

$$\text{rank}_{RW}(C, C') := \frac{1}{||h(C) - h(C')||_\infty + \varepsilon} \cdot \frac{1}{\text{size}(C) + \text{size}(C')} \sum_{N \in \mathcal{N}:\ C, C' \in N} \frac{\omega(N)}{|N|} \tag{7.10}$$

with $h(\{c\}) := h(c)$ for initial clusters and $h(C \cup C') := \frac{\text{size}(C)h(C) + \text{size}(C')h(C')}{\text{size}(C) + \text{size}(C')}$ for merged objects, and some $\epsilon > 0$ to prevent division by 0.
This rank function discourages clustering of objects which differ in hitting time values to *some* target. Then, the high-level description of our clustering algorithm with random walks is presented in Algorithm 13. In this approach, τ targets are chosen, then τ hitting time numbers are computed for each cell $c \in \mathcal{C}$ and stored in vectors $h(c)$. Finally the *BestChoice* algorithm is used with the new rank function (7.10) with cluster ratio parameter $\alpha \in \mathbb{R}_+$. We did not specify how the targets are computed and how many of them are used. This will be done in the next section.

Algorithm 13: RandomWalkBottomUpClustering($\mathcal{C}, \mathcal{N}, P, \gamma, \alpha, \tau$)

```
1 Compute $\tau$ targets $\{t_1, \ldots, t_\tau\}$
2 for $i = 1, \ldots, \tau$ do
3     Compute $h_{\cdot|i} : \mathcal{C} \to \mathbb{R}_+$
4 BestChoice $(\mathcal{C}, \mathrm{rank}_{RW}, \alpha)$
```

7.4 Fast Random Walk Model

Computing hitting times on the entire netlist is expensive for each particular target, and for memory reasons not too many targets can be considered simultaneously either. For performance reasons we propose considering a modified problem, presented below.
Fix a weighted, connected graph $G = (V, E, \omega)$ with $V = \{1, \ldots, n\}$ and recall Proposition 7.3.1. To choose a node $t \in V$ in the graph as target and to compute hitting times from other nodes to t is to delete t's row and column from $L(G)$. Now, instead of computing $h(v|t)$ for $v \in V \setminus \{t\}$, we propose to compute hitting times in a different graph.
To this end define $G^t := (V^t, E^t, \omega^t)$ with $V^t := V \dot{\cup} \{0\}$, $E^t := E \dot{\cup} \{t, 0\}$, $\omega^t(e) := \omega(e)$ for all $e \in E$, and $\omega^t(\{0, t\}) = c$ for some $c \in \mathbb{R}_{>0}$. Now consider hitting times $h(v|0)$ for nodes in $v \in V^t$ instead of $h(v|t)$ in V. As we are interested in hitting time differences in V, nodes which have different hitting times in G to t will also have different hitting times in G^t to 0, as $\{0, t\}$ is the only arc connecting 0 to other nodes.
This model has several advantages. The first one is the fact that for two different targets the Laplacians and the right-hand side vectors from Proposition 7.3.1 are almost the same and are based on the Laplacian $L(G)$ of G:

Proposition 7.4.1 Let $G = (V, E, \omega)$ be a weighed, connected graph with $V = \{1, \ldots, n\}$ and $t \in V(G)$, and L the Laplacian of G. Let G^t be the graph as above and L^t its Laplacian. Then the hitting times $h(\cdot|0)$ in G^t can be obtained from the solution $x \in \mathbb{R}^n$ of the system:

$$L^t_{\setminus 00} x = b \tag{7.11}$$

with $h(0|0) := 0$ and $h(v|0) := x_v$ for $v \in V(G)$ and

$$L^t_{\setminus 00} = L + \mathrm{diag}(0, \ldots, 0, \overset{\downarrow t}{c}, 0, \ldots, 0) \tag{7.12}$$

$$b = (l_{11}, \ldots, l_{nn})^T + ce_t \tag{7.13}$$

where e_t is the t-th unit vector in $\mathbb{R}^n$.

Proof: Immediate from Proposition 7.3.1 by constructing the Laplacian L^t of G^t and deleting the 0-th row and column. □

In particular, when dealing with different targets $T \subset V$ the matrices L^t for $t \in T$ do not have to be generated separately for different targets in the netlist. This fact also allows memory efficient use of the conjugate gradient method in parallel. Instead of storing $|T|$ matrices L^t, $t \in T$ in memory, one can do better:

Corollary 7.4.2 Given $G = (V, E, \omega)$ as above and a set $T \subset V$ of targets, let m be the number of non-zero entries in $L(G)$. Then computing $|T|$ hitting time vectors in G^t, $t \in T$ in parallel requires the simultaneous storage for at most $6|T||V(G)| + 2m + \mathcal{O}(T)$ entries.

Proof: The matrix can be stored in the classical compressed-row-storage format, which requires $|T| + 2m$ entries. The conjugate gradient can be implemented using 5 vectors of size $|V|$ per target and a small constant amount of numbers (Stoer and Bulirsch [2002]). As the matrices $L(G^t)_{\setminus 00}$ for $t \in T$ can be expressed by a shared matrix $L(G)$ for different targets by Proposition 7.4.1, we are done. □

Beyond memory-efficient parallelization and the construction of a single matrix from the netlist, two other issues are essential for fast hitting time computations in G^t by iterative methods: the initial solution and the precision. For both, the lower bounds for hitting times to different targets can be explicitly deduced from G^t, which turn out to be graph invariants of G. These lower bounds help to estimate the initial solution much more accurately, and also contribute to the stopping criterion of the conjugate gradient method.

Theorem 7.4.3 Let $G = (V, E, \omega)$ be a graph as above and $V = \{1, \dots, n\}$. For $t \in V$ Let $G^t = (V^t, E^t, \omega^t)$ be the modification of G as above. Let $h(\cdot|0) : V^t :\to \mathbb{R}$ be the hitting time from any node of G^t to 0 with $h(0|0) := 0$. Then $h(t|0) = \frac{c+2\sum_{e\in E}\omega(e)}{c}$. In particular, the hitting time of the neighbor t of 0 to 0 does not depend on t.

Proof: Consider the Laplacian $L = L(G)$ and $L^t := L(G^t)_{\setminus 00}$ as above. Let $(l_{ij})_{i,j=1}^n = L$ and $(l_{ij}^t)_{i,j=1}^n = L^t$. Observe that $l_{tt}^t = l_{tt} + c$ and $l_{jk}^t = l_{jk}$ for $1 \leq j, k \leq n$, $k, j \neq t$. The claim is equivalent to $h(t|0) = \frac{\text{trace}(L)+c}{c}$. The hitting time from $t \in V$ to 0 in G^t is the t-th entry of the solution x of the system in Proposition 7.4.1, which by regularity of L^t translates by Cramer's rule to:

$$x_t = \frac{\det(l_{\cdot 1}^t, \dots, l_{\cdot t-1}^t, b, l_{\cdot t+1}^t, \dots, l_{\cdot n}^t)}{\det(L^t)} \tag{7.14}$$

with $b = (l_{11}, \dots, l_{nn})^T + ce_t$.
Now,

$$\begin{aligned}\det(L^t) =& \sum_{k=1}^n (-1)^{k+t} l_{kt}^t \det(L_{\setminus kt}^t) = \sum_{k=1}^n (-1)^{k+t} l_{kt}^t \det(L_{\setminus kt}) \\ =& \sum_{k=1}^n (-1)^{k+t} l_{kt} \det(L_{\setminus kt}) + (-1)^{2t} c \det(L_{\setminus tt}) = \det(L) + c\det(L_{\setminus tt}) \\ =& c\det(L_{\setminus tt}).\end{aligned} \tag{7.15}$$

The last equation follows by the singularity of L. Now, consider the denominator of (7.14). The determinant is invariant under simple row operations, so we can add to the t-th row $l_{t\cdot}^t$

of L^t any other row of L^t and obtain $l_{t\cdot}^{t'}$. The original row $l_{t\cdot}^t$ has the form:

$$l_{t\cdot}^t = (-\omega_{t,1}, -\omega_{t,2}, \dots, -\omega_{t,t-1}, l_{tt} + c, -\omega_{t,t+1}, \dots, \omega_{tn}). \tag{7.16}$$

Therefore,

$$\begin{aligned} l_{t\cdot}^{t'} = (-\omega_{t,1} - \sum_{k=2,k\neq t}^{n} \omega_{tk} + l_{11}, \dots, -\omega_{t,t-1} - \sum_{k=1,k\neq t-1,i}^{n} \omega_{tk} + l_{t-1,t-1}, \\ l_{tt} + c + \sum_{k=1,k\neq t}^{n} l_{kk}, \\ -\omega_{t,t+1} - \sum_{k=1,k\neq t,t+1}^{n} \omega_{tk} + l_{t+1,t+1}, \dots, -\omega_{t,n} - \sum_{k=1,k\neq t}^{n-1} \omega_{tn} + l_{nn}) \end{aligned} \tag{7.17}$$

As L is a Laplacian, all entries except the t-th cancel to 0, and we obtain:

$$l_{t\cdot}^{t'} = (0, \dots, 0, \sum_{k=1}^{n} l_{kk} + c, 0 \dots, 0)$$

Now, returning to (7.14), we expand the determinant of $\det(l_{\cdot 1}^t, \dots, l_{\cdot t-1}^t, b, l_{\cdot t+1}^t, \dots, l_{\cdot n}^t)$ by the t-th row.

$$\begin{aligned} \det(l_{\cdot 1}^t, \dots, l_{\cdot t-1}^t, b, l_{\cdot t+1}^t, \dots, l_{\cdot n}^t) &= \sum_{k=1}^{t} (-1)^{k+t} l_{tk}^t \det(L_{\setminus tk}^t) \\ &= \sum_{k=1}^{i} (-1)^{k+i} l_{tk}^t \det(L_{\setminus tk}) = \sum_{k=1}^{i} (-1)^{k+i} l_{tk}^{t'} \det(L_{\setminus tk}) \\ &= (\sum_{k=1}^{n} l_{kk} + c) \det(L_{\setminus tt}) = (\operatorname{trace}(L) + c) \det(L_{\setminus tt}). \end{aligned} \tag{7.18}$$

Combining (7.18) with (7.15) yields the desired result. □

The Theorem 7.4.3 is of particular interest for numerical reasons, as it provides lower bounds for the solution of (7.11). A proper estimation of the solution is necessary to provide good convergence of the conjugate gradient method. Moreover, in our instances we observe that the ratio between the highest hitting time and the lowest $h(t|0)$ differs by a small constant β, usually with $\beta \leq 10$. This issue has two further advantages. The first is the fact that if all hitting times to different targets t are in the range $[h(t, 0), \beta h(t, 0)]$, then the rank function is not dominated by a particular target choice, and different targets have similar weights. The second issue allows a proper choice of the stopping criterion for the iterative conjugate gradient method, which halts if the differences between two solutions did not change too much.
It remains an open question, whether any non-trivial bound for the maximum hitting time in G^t can be deduced from theory.

7.5 Target Choices

Let us return to the netlist view of the random walk model presented above. In this context, the nets are modeled by cliques and stars in order to obtain a graph G out of the netlist hypergraph as in Theorem 7.3.2. The modified graph G^t has a natural interpretation in terms of the netlist: it can be seen as deleting all preplaced pins, attaching a new virtual preplaced pin by a two terminal net with weight c. The Laplacian L of G corresponds to the QP matrix A without any side-constraints from (3.9), where all diagonal entries have been reset to the negative sum of the off-diagonal values. With this method, we can indeed simulate connectivity of cells to preplaced pins, artificial or not.

In the context of placement, for a fixed target t, to ask for the hitting times $h(c|0)$ in G^t means to ask for the expected number of steps a random walk starting in c (of the netlist modified by deletion of all preplaced pins and attachment of a pin at t) requires to reach the preplaced pin attached to t.

For this reason, we will indeed delete all preplaced pins in the netlist $(\mathcal{C}, \mathcal{N}, P, \gamma)$, as connections to preplaced objects can be simulated by the method above: other preplaced pins would cover the differences, at least partially. In order to obtain more information, targets should not be chosen in proximity to each other.

Given a distance function dist : $\mathcal{C} \times \mathcal{C} \to \mathbb{R}_+$, one asks for a set $T \subset \mathcal{C}$ which maximizes the minimum pairwise distances among elements in T. Let τ be the desired number of targets. The target choice is thus the classical τ-Dispersion Problem, which is known to be $\mathbb{NP}$-hard (Tamir [1991]). For this reason we use a fast greedy method, which picks initially some cell c_0 in $\mathcal{C}$ and sets $T_0 := \{c_0\}$. Once a set T_i of i targets is computed, choose $c_{i+1} \in \mathcal{C}$ with

$$c_{i+1} = \arg\max_{c \in \mathcal{C}} \min_{c_0, \ldots, c_i} \{\mathrm{dist}(c_0, c), \ldots, \mathrm{dist}(c_i, 0)\}, \tag{7.19}$$

until $i = \tau$. The set of targets is then $T := T_n \setminus \{c_0\}$. As a distance function, we have tested the simple (unweighted) hypergraph distance in $(\mathcal{C}, \mathcal{N}, P, \gamma)$ and the distances resulting from already computed hitting times, the latter for the price of sequential instead of parallel computation of random walks. Another method would be to consider geometric information in the form of preplaced pins in distant chip areas. For a detailed analysis we refer to Lauff [2010].

For performance reasons we compute the targets a-priori, using the breadth-first search method in the netlist. As we delete all preplaced pins from the netlist for random walk computation, the netlist can split into several connected components. Usually, there are one or two of them with a larger number of cells, the others consisting of a couple of cells each. Consequently, target choices and random walk computations are limited to the larger components only, which contain at least 1% of the cells. The number of targets is limited to 16 per connected component, which seems to be a reasonable trade-off between runtime and quality.

7.6 Geometric Information and the Overall Algorithm

Deletion of preplaced pins from the netlist $(\mathcal{C},\mathcal{N},P,\gamma)$ results in the loss of the geometric information from placement in the clustering. To compensate for this, we propose to compute the global QP (3.9) on the initial netlist and treat the (scaled) differences between vertical and horizontal coordinates between the cells similarly to the hitting time differences. Let $(\mathcal{C},\mathcal{N}',P',\gamma')$ denote the netlist with deleted preplaced pins. One has to prevent clusterings between cells of different movebounds.
For a cell c in some large connected component of $(\mathcal{C},\mathcal{N}',P',\gamma')$, we store its hitting time values to different targets in the vector $h(c)$. For a cell in some small connected component, for which no hitting time computation is performed, we set $h(c) := \mathbf{1} \in \mathbb{R}^\tau$. To prevent division by 0 for similar or equal values of h, we use a small $\varepsilon > 0$.

$$\operatorname{rank}_{RW'}(C,C') := \begin{cases} -\infty & \text{if } \mu(C) \neq \mu(C') \text{ or} \\ & C,C' \text{ in different components of } (\mathcal{C},\mathcal{N}',P',\gamma') \\ \frac{1}{\|h(C)-h(C')\|_\infty+\varepsilon} & \cdot\frac{1}{\operatorname{size}(C)+\operatorname{size}(C')}\sum_{N\in\mathcal{N}:\ C,C'\in N}\frac{\omega(N)}{|N|} \text{ else .} \end{cases} \tag{7.20}$$

The overall clustering algorithm then translates to:

Algorithm 14: RandomWalkClustering$(\mathcal{C},\mathcal{N},P,\gamma,\Gamma_0,\alpha,\tau,\Pi)$

```
1  QP ($\Gamma_0$)
2  $(\mathcal{C},\mathcal{N}',P',\gamma')$ := DeletePreplacedPins $(\mathcal{C},\mathcal{N},P,\gamma)$
3  Let $\mathcal{C}_1,\dots,\mathcal{C}_m$ be the large connected components of $(\mathcal{C},\mathcal{N}',P',\gamma')$
4  for $i = 1,\dots,m$ do
5      $T$ := ComputeTargets $(\mathcal{C}_i,\mathcal{N}',P',\gamma',\tau)$
6      $\mathcal{J} := \emptyset$
7      $L$ :=Laplacian obtained from $(\mathcal{C}_i,\mathcal{N}',P',\gamma')$ using clique/star
8      for $t \in T$ do
9          $\mathcal{J} = \mathcal{J} \cup \{(\mathcal{C}_i,L,t)\}$
10     RunJobsInParallel (ComputeHittingTimes, $\mathcal{J}$, $\Pi$)
11 BestChoice $(\mathcal{C},\alpha,\operatorname{rank}_{RW'})$
```

In Algorithm 14 one proceeds as follows: first, global QPs are computed on the netlist $(\mathcal{C},\mathcal{N},P,\gamma)$ to determine the locations of the cells. Then, the preplaced pins are removed from $(\mathcal{C},\mathcal{N},P,\gamma)$, which yields a modified netlist $(\mathcal{C},\mathcal{N}',P',\gamma')$. As $(\mathcal{C},\mathcal{N}',P',\gamma')$ can break down into several connected components, the large components are identified, and targets T are computed using (7.19) in each of them. In each connected component, the hitting times to targets in T are computed in parallel. For parallel computation, we use the shared Laplacian (see Theorem 7.4.3), obtained from the sparse graph model of the hypergraph induced by $(\mathcal{C}_i,\mathcal{N}',P',\gamma')$ (see Theorem 7.3.2).

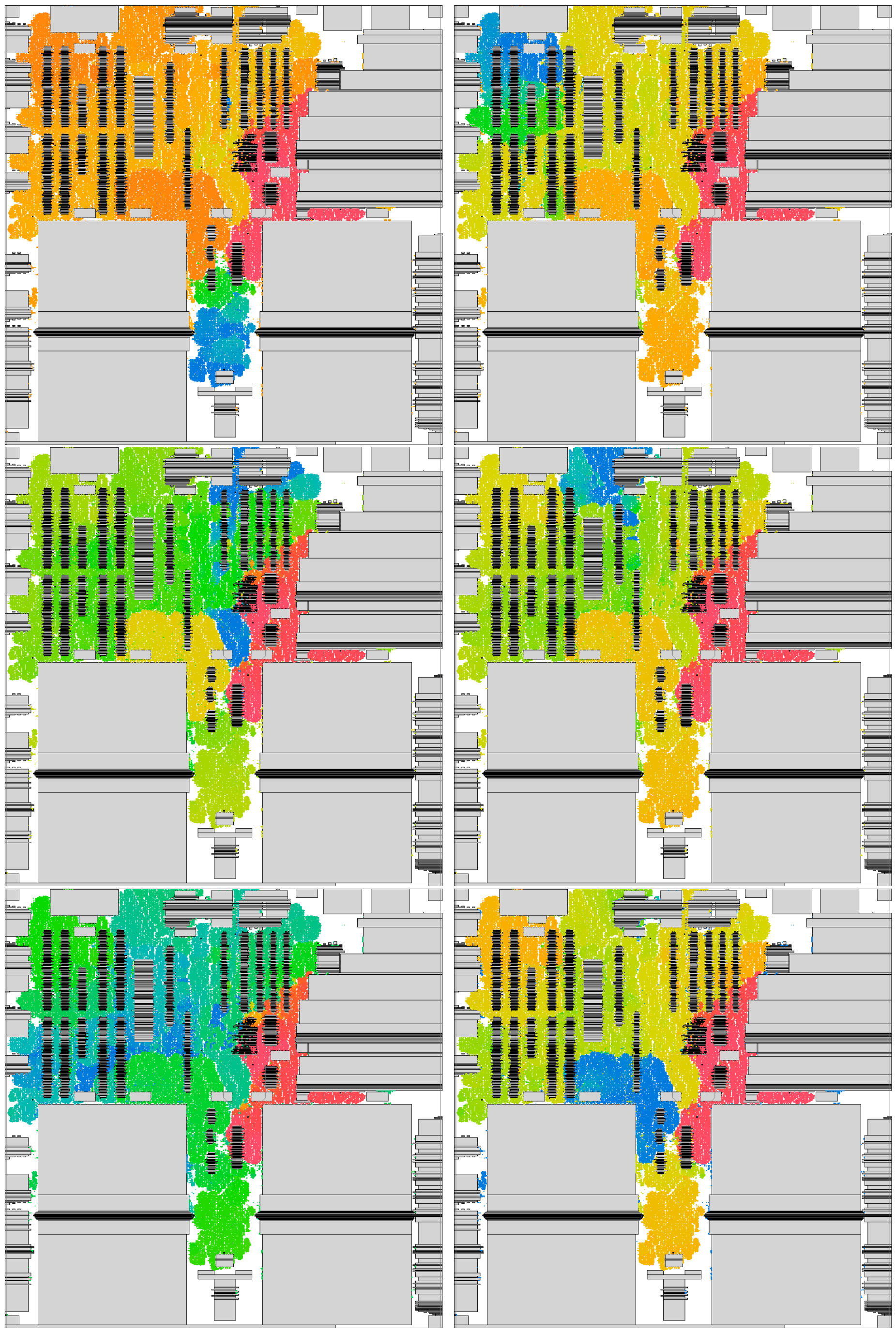


Figure 7.1: Hitting times to different targets on the chip Julia. Cells are colored w.r.t. the hitting times for targets 1-6. The range starts from blue (low hitting times) to red (high hitting times), revealing the netlist structure.

Figure 7.1 shows an example of six different random walks computed to targets based on (7.19) using the unweighted hypergraph distance in $(\mathcal{C}, \mathcal{N}, P, \gamma)$ of the chip Julia. The movable cells are colored according to their hitting time distances to the targets, and the gray objects are preplaced. The locations of the cells were obtained independently from the random walk computations by another placement run of BonnPlace. Blue color means low hitting times, red high hitting times. This picture shows several "sharp edges" in the netlist, identifying natural clusters. Cells which have different colors in *any* of the images should not be clustered. It should be noted that similar colors do not necessarily mean natural clusters.

7.7 Implementation

There exists a first implementation of the presented random walk-based bottom up clustering in BonnPlace. The target computation was implemented on the basis of the BFS algorithm. Once the targets are computed, a job queue is initialized, storing all the targets. The computation is performed in parallel mode, using the shared Laplacian among the different threads. In our preliminary implementation, the sparse Cholesky preconditioner is computed for each thread separately. It is not straightforward, how to compute a good shared preconditioner for all the threads, as the shared Laplacian is a singular matrix and the Cholesky decomposition requires a full rank operator. An easy method how to share the preconditioner among matrices would be to consider a perturbation of the Laplacian $L_\varepsilon = L + \operatorname{diag}(\varepsilon, \dots, \varepsilon)$ with some small $\varepsilon > 0$. To approximate the effect of the artificial $\{t, 0\}$ connections, $t \in T$, one can modify some entries in the thread-specific preconditioners for each thread. In our tests, for the shared sparse Cholesky decomposition $L_\varepsilon \approx PP^T$, the increased diagonal value $p_{tt} := p_{tt} + \sqrt{c}$ (where c is the weight of the connection $\{t, 0\}$) of P lead to $5\% - 15\%$ of more iterations than a specific preconditioner. It is thus quite an option to save memory.
Moreover, the preliminary implementation of the bottom-up method uses a hard bound for the maximal distance between nodes from the initial QP. If two cells are too far away w.r.t. the initial QP, the clustering of them is forbidden (i.e. $\text{rank}_{RW'} = -\infty$).

7.8 Discussion and Outlook

We will see in Section 8.8 that one can do much better than *BestChoice* using random-walk clustering.
Hitting time differences seem to identify "natural" cluster bounds in placement and guide a bottom-up clustering into the right direction, at least for the small and mid-sized designs. For the larger test cases, the information resulting from the small number of random walks is apparently insufficient to achieve results with the same quality as the unclustered runs. For runtime reasons, one will not be able to increase the number of global hitting time computations to the hundreds and thousands. But here, a divide-and-conquer approach

is possible: One can use the global hitting times from a small number of targets on the entire netlist to identify global cuts, and use the global cuts in the subsequent hitting time computations.
Moreover, the presented target choice and implementation are preliminary. For the target choice, weights should be incorporated into the model, and one could also think of the usage of the cuts identified by the global hitting time differences to be taken into account for target choice. Finally, the implemented method uses a very simple combination of geometric information and hitting times, with singular hitting times targets. In order to obtain more floorplan information into this model, the target choice itself could be expanded to consider more preplaced pins and connectivity to them.
The hitting time computation is done with the same preconditioner as the global QP. BonnPlace has been using the incomplete Cholesky decomposition (see e.g. Stoer and Bulirsch [2002]) as preconditioner. Without random walk computation, there are only two global QPs (namely those in level 0) which share the same matrix and spending more time on preconditioner computation was not justified. The more global linear systems of this size are computed, the more time one could spend for an improved preconditioning technique.
Another issue seems also interesting when using hitting times and random walk information for clustering. The cluster ratio, usually part of the input, does play an inferior role in placement. One is rather interested in better results than in the exact ratios and hitting times can well be used for this purpose.
In total, we believe that random walk information is well able to identify structures in large netlists and to improve the placement. The presented method allows us to retrieve global random walk information efficiently, in both runtime and memory, taking advantage of the increasing use of multi-core systems. This global information can then be used to obtain more knowledge about the local structures of the netlist.

Chapter 8

Experimental Results

In this chapter we present several experiments for particular subroutines and the overall results of our tool. We first have a closer look at the flow-based partitioning and its extensions. Then, we compare our placement tool to the old version of BonnPlace and a state-of-the-art industrial force-directed placement engine, using a large testbed of modern ASICs and RLMs in recent technologies. Later, we provide the results of our tool on the latest placement benchmarks, and conclude by presenting the first results of the random walk based clustering.

8.1 The Testbed Environment

Our test instances consist of a set of recent chips from industry. The smallest design is a 32 nm RLM with 50 thousand cells, the largest is a 9.3 million cells ASIC. In our testbed we have also used, thanks to the University of Texas at Austin, the large Trips design (Trips [2006]). The chips of the testbed are shown in Table 8.1, summarizing the instance in the form of the number of movable cells, nets and pins, the physical chip area, the cell density $\sum_{c \in \mathcal{C}} \text{size}(c)/\text{free\_area}(\mathcal{A})$, and the number of levels used for global placement. In addition to the industrial instances, we ran our tool on the latest placement benchmarks, presented at ISPD 2005 (Nam et al. [2005]) and ISPD 2006 (Nam et al. [2006]).

In the comparisons, we will make use of the geometric mean to obtain average values of improvements and degradations. To compare total runtimes, we will consider the ratio of the sums of underlying runtimes.

8.2 Flow-based Partitioning

In this section we present the experiments of the extensions of flow-based partitioning in detail.

The testbed								
chip	technology	$\|\mathcal{C}\|$	$\|\mathcal{N}\|$	$\|P\|$	$x_{\text{size}}(\square)$ (mm)	$y_{\text{size}}(\square)$ (mm)	non preplaced cell density	num levels
Dagmar	32 nm	50 534	55 788	204 979	0.38	0.50	36.9%	8
Elisa	90 nm	67 723	68 267	238 801	1.38	1.11	65.5%	8
Lucius	65 nm	77 679	78 400	266 759	2.30	1.70	47.1%	9
Felix	130 nm	84 966	87 733	270 519	2.11	1.44	74.3%	8
Paula	45 nm	129 026	127 556	469 492	1.11	0.95	67.4%	9
Rabe	130 nm	175 646	178 172	650 114	4.57	4.19	52.2%	9
Julia	130 nm	190 554	190 832	690 940	4.13	4.20	49.2%	9
Max	90 nm	328 789	330 998	1 157 377	4.85	4.85	55.1%	10
Roger	90 nm	456 700	469 489	1 609 933	6.30	6.30	48.3%	10
Ashraf	65 nm	866 777	852 782	2 889 162	6.06	6.06	35.8%	10
Patrick	45 nm	1 052 709	1 178 154	3 889 726	2.22	3.46	51.6%	10
Erhard	130 nm	2 578 246	2 618 187	8 682 845	13.81	13.88	38.6%	10
Arijan	65 nm	3 753 151	3 798 307	12 253 633	14.76	14.76	32.6%	11
Philipp	90 nm	3 945 833	3 943 770	13 129 567	14.83	14.76	51.3%	11
Tomoku	65 nm	5 296 120	5 248 201	18 655 371	14.80	15.60	44.2%	11
Trips	130 nm	5 747 007	5 990 756	21 161 241	18.31	18.38	54.4%	11
Valentin	90 nm	5 838 457	5 870 943	20 775 172	13.96	13.96	45.6%	11
Andre	65 nm	6 794 323	6 948 237	24 221 307	15.60	14.80	27.0%	11
Ludwig	65 nm	7 500 446	7 589 386	25 751 156	14.80	15.60	41.6%	11
Leyla	65 nm	8 472 478	8 546 713	30 336 029	18.80	18.80	33.3%	11
Erik	65 nm	9 316 938	9 271 613	32 275 635	16.40	16.40	45.7%	11

Table 8.1: The testbed for our experiments.

8.2.1 MinCostFlow Computation

The MinCostFlow instances presented in 5.2.1 have, despite the large number of movebounds and grid windows, a relatively small number of edges. The number of edges is about 4 times the number of nodes in the instance. Table 8.2 shows two exemplary chips with movebounds, which induce indeed the largest instances in our testbed. In this table we provide the number of nodes ($|V|$) and edges ($|E|$), the edge-node ratio $|E|/|V|$, as well as the number of windows $|\mathcal{W}|$ and regions $|\mathcal{R}|$ of the corresponding instance for the different levels. We also show the runtimes of the NetworkSimplex routine used to solve the MinCostFlow and the runtimes for realization. The runs were made on Intel XEON with 3.3 GHz and 64 GB. We see in this table that the time needed to create and solve the MinCostFlow instance, even for the two largest instances Ashraf and Erhard is negligible for levels 1-7. From level 8 on, the runtimes increase and in the final levels the largest flow-based partitioning instance in our testbed, Erhard, can be solved within 6 minutes. However, one should note that these runtimes basically do not depend on the number of cells: this would be too much, in particular for small instances. As we do not observe any improvement in terms of netlength during the last levels with the flow-based partitioning, we use the recursive method from level 8 on, by default.

8.2.2 Realization

The realization during the flow-based partitioning is done with windows of size 2×3 and 3×2. Smaller realization windows (1×2, 2×1) lead to worse results with a degradation of about 1% on average and save just 2% of the overall placement runtime. Larger windows

Ashraf (206 movebounds)							
Level	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E\rvert/\lvert V\rvert$	$\lvert\mathcal{W}\rvert$	$\lvert\mathcal{R}\rvert$	MinCostFlow runtime	
						flow computaton	realization
3	2 761	17 198	6.2	16	246	0:00:00	0:00:16
4	6 453	31 810	4.9	64	367	0:00:00	0:09:18
5	15 828	69 237	4.4	256	616	0:00:00	0:00:38
6	38 429	151 993	4.0	1 024	1 368	0:00:00	0:00:10
7	119 414	437 991	3.7	4 096	3 598	0:00:01	0:00:01
8	663 252	2 368 934	3.6	98 304	481 075	0:00:04	0:00:01
9	3 641 561	12 858 495	3.5	556 416	231 000	0:00:30	0:00:14
10	7 973 618	28 181 428	3.5	1 216 194	477 571	0:01:46	0:00:09

Erhard (43 movebounds)							
Level	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E\rvert/\lvert V\rvert$	$\lvert\mathcal{W}\rvert$	$\lvert\mathcal{R}\rvert$	MinCostFlow runtime	
						flow computaton	realization
3	2 783	15 243	5.5	16	94	0:00:00	0:00:26
4	7 773	38 366	4.9	64	206	0:00:00	0:00:11
5	22 184	102 538	4.6	256	520	0:00:00	0:00:06
6	55 346	238 084	4.3	1 536	2 118	0:00:00	0:00:06
7	165 414	668 414	4.0	9 216	9 464	0:00:01	0:00:04
8	930 027	3 685 136	4.0	55 296	45 643	0:00:08	0:00:07
9	5 198 211	20 365 870	3.9	331 776	239 885	0:01:20	0:00:11
10	13 004 080	50 960 038	3.9	1 414 080	931 157	0:05:53	0:00:28

Table 8.2: Flow-based partitioning: instance sizes of the underlying MinCostFlow instance graph $G = (V, E)$ in comparison to the number of windows $|\mathcal{W}|$ and regions $|\mathcal{R}|$. The runtimes reflect the time for construction and solving the MinCostFlow using the NetworkSimplex algorithm and its realization. The runs were made on Intel XEON with 3.3 GHz and 64 GB.

do not improve the results, but make the placement slower.
In the last column of Table 8.2, wall clock realization runtimes are provided, observed on clustered netlists with a cluster ratio of 5 and target density of 97%. The overall runtimes are very low, except for Ashraf in level 4. Here, a single Multisection instance dominated the realization and took more than 9 minutes.

8.2.3 Level Skipping

Table 8.3 summarizes the effects of level skipping, as proposed in Section 5.3.1. We present the results with 2×2 and 3×3 repartitioning. The default run is the scenario with levels 1 and 2 skipped and using 3×3 repartitioning. We compare global placement wall clock runtimes and the final, legalized netlengths. To avoid all density issues we set the target density to 97% and use *BestChice* clustering with cluster ratio 5. Comparing this scenario to the runs with levels 1 and 2 with 3×3 repartitioning, we see that skipping the levels does even slightly improve the netlengths on average. The differences are bigger when comparing the 2×2 repartitioning runs. Skipping levels has a much higher impact here. In particular for the chips Julia, Rabe, and Roger the flow-based partitioning responds better to the floorplan of the chips which all contain large blockages. The level skipping yields about 4% improvement for Julia and Roger and almost 14% for Rabe over the run without skipinng and 2×2 repartitioning.

Chip	new BonnPlace with flow-based partitioning							
	2x2 repartitioning				3x3 repartitioning			
			skipped level 1,2				skipped level 1,2	
	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss
Elisa	2.92	0:00:36	2.89	0:00:24	2.87	0:01:01	2.82	0:00:39
	103.5%	92.3%	102.5%	61.5%	101.5%	156.4%	100.0%	100.0 %
Lucius	3.77	0:00:54	3.7	0:00:33	3.78	0:01:14	3.87	0:00:50
	97.5%	108.0%	95.7%	66.0%	97.8%	148.0%	100.0%	100.0 %
Felix	7.42	0:00:30	7.41	0:00:20	7.36	0:00:44	7.31	0:00:35
	101.6%	85.7%	101.4%	57.1%	100.7%	125.7%	100.0%	100.0 %
Rabe	15.09	0:01:48	13.33	0:01:08	12.69	0:03:22	12.73	0:02:11
	118.6%	82.4%	104.7%	51.9%	99.7%	154.2%	100.0%	100.0 %
Julia	11.54	0:02:20	11.14	0:01:32	11.00	0:04:40	10.93	0:03:27
	105.6%	67.6%	101.9%	44.4%	100.6%	135.3%	100.0%	100.0 %
Max	20.72	0:04:14	20.7	0:02:44	20.26	0:05:53	19.93	0:03:54
	103.9%	108.5%	103.8%	70.1%	101.6%	150.9%	100.0%	100.0 %
Roger	30.04	0:05:18	28.85	0:03:33	29.03	0:09:40	28.53	0:06:10
	105.3%	85.9%	101.1%	57.6%	101.7%	156.8%	100.0%	100.0 %
Ashraf	61.89	0:06:41	62.46	0:05:02	60.34	0:09:23	60.57	0:09:35
	102.2%	69.7%	103.1%	52.5%	99.6%	97.9%	100.0%	100.0 %
Patrick	45.23	0:10:39	45.42	0:08:20	44.71	0:19:49	45.09	0:10:44
	100.3%	99.2%	100.7%	77.6%	99.2%	184.6%	100.0%	100.0 %
Erhard	421.55	0:31:10	417.11	0:20:49	410.61	0:53:13	406.78	0:38:28
	103.6%	81.0%	102.5%	54.1%	100.9%	138.3%	100.0%	100.0 %
Arijan	515.36	0:55:16	509.06	0:37:19	495.42	1:49:00	491.66	1:19:28
	104.8%	69.5%	103.5%	47.0%	100.8%	137.2%	100.0%	100.0 %
Philipp	380.24	0:58:02	379.08	0:42:51	364.17	1:34:02	363.38	1:11:34
	104.6%	81.1%	104.3%	59.9%	100.2%	131.4%	100.0%	100.0 %
Tomoku	368.17	0:54:25	375.24	0:38:29	363.85	1:24:07	362.28	1:21:24
	101.6%	66.9%	103.6%	47.3%	100.4%	103.3%	100.0%	100.0 %
Ludwig	631.61	1:29:32	628.67	1:08:54	613.51	2:53:15	610.34	1:58:43
	103.5%	75.4%	103.0%	58.0%	100.5%	145.9%	100.0%	100.0 %
Leyla	779.22	0:53:07	781.41	0:50:59	740.47	1:21:56	738.55	0:56:06
	105.5%	94.7%	105.8%	90.9%	100.3%	146.0%	100.0%	100.0 %
Average	104.1%		102.5%		100.4%		100.0%	
Total		6:14:32		4:42:57		10:51:19		8:03:48
		77.4%		58.5%		134.6%		100.0%

Table 8.3: Flow-based global partitioning with and without skipping of levels 1 and 2. Runtimes are the wall clock runtimes of global placement, BBox is the netlength after legalization. The runs were performed on Intel XEON with 3.3 GHz and 64 GB in parallel mode with up to 8 CPUs

The runtime differences are also remarkable. In both repartitioning scenarios the gain of skipping levels 1 and 2 is about one third of runtime against an otherwise identical run without level skipping. This option allows us to use 3×3 repartitioning as a default feature. As an option, the fast mode with 2×2 repartitioning with and without level skipping can be used. Skipping more and later levels leads to degradations.

From now on, all reported results use the default settings with level skipping and 3×3 repartitioning, unless otherwise noted.

Chip	2x2 repartitioning without re-allocation	2x2 repartitioning + greedy re-allocation	3x3 repartitioning without re-allocation	3x3 repartitioning + greedy re-allocation
	BBox(m)			
Elisa	2.89 102.0%	2.92 103.3%	2.79 98.7%	2.83 100.0%
Lucius	3.81 99.8%	3.67 96.0%	3.86 101.1%	3.82 100.0%
Felix	7.55 103.8%	7.57 104.1%	7.46 102.7%	7.27 100.0%
Rabe	13.07 102.3%	13.09 102.4%	12.87 100.7%	12.78 100.0%
Julia	11.60 107.8%	11.24 104.5%	11.24 104.6%	10.75 100.0%
Max	21.35 111.5%	20.97 109.6%	19.77 103.3%	19.14 100.0%
Roger	30.00 105.2%	29.71 104.1%	29.63 103.8%	28.53 100.0%
Ashraf	64.05 106.4%	61.97 102.9%	61.67 102.4%	60.22 100.0%
Patrick	45.85 101.4%	45.5 100.6%	46.18 102.1%	45.24 100.0%
Erhard	416.40 102.1%	418.3 102.6%	411.98 101.0%	407.79 100.0%
Arijan	513.14 104.8%	517.07 105.6%	493.02 100.7%	489.46 100.0%
Philipp	375.77 103.2%	377.38 103.7%	365.15 100.3%	364.06 100.0%
Tomoku	377.90 104.3%	375.47 103.6%	365.21 100.8%	362.34 100.0%
Ludwig	634.66 103.7%	633.23 103.5%	615.18 100.5%	612.01 100.0%
Leyla	753.03 104.9%	755.16 105.1%	720.25 100.3%	718.17 100.0%
Average	104.2%	103.4%	101.5%	100.0%

Table 8.4: Legalized bounding box netlength for runs with flow-based global partitioning with and without greedy re-allocation of cells. Runtime of greedy re-allocation was neglectable ($\approx 1\%$).

8.2.4 Greedy Re-allocation

Greedy re-allocation was introduced in Section 5.3.2. Table 8.4 shows the results of the greedy re-allocation for 2×2 and 3×3 repartitioning with high density and using *BestChoice* with cluster ratio 5. The additional runtimes for this routine are extremely small: unlike greedy re-allocations (*postopt*) in detailed placement, this method neither needs to pay attention to blockages nor to shapes. The netlength just has to be evaluated and updated. In total, this method yields another 0.8-1.5% of netlength improvement on average, and in several cases the improvement exceeds 3%. We use this method in each level, before the flow-based partitioning is applied. The implementation is done in parallel, using a-priori job assignment of a chunk of cells to the threads (cf. Section 6.3.3).

8.3 New vs. Old BonnPlace

The new version of BonnPlace replaced the old version as the default global placement engine. We provide the comparisons to the old BonnPlace in different scenarios in this section.

8.3.1 Recursive Partitioning

In this section we compare the implementations of the old and new version of BonnPlace. Despite its generalized data structures, the repeatability, and virtual classes, the new BonnPlace data structures and algorithms work efficiently. Table 8.5 summarizes the results and shows bounding box netlengths after legalization (BBox) and wall clock runtimes of global placement runs.
Both tools used up to 4 CPUs simulaneously. The target density was set to 70%. The old and new implementations share only the same core conjugate gradient solver and multisection algorithm, but differ in the implementation of all other routines.

When comparing the same partitioning algorithms, the new version is faster by more than 2.3 times on average. The comparison to the standard version of old BonnPlace using iterative partitioning shows that even recursive partitioning in the new BonnPlace is more than 2 times as fast. Non-determinism, different data orderings and the less-restrictive handling of density constraints in iterative partitioning lead to slightly different results. It should be noted that the additional speedup is obtained with the same number of CPUs.

8.3.2 Flow-based Partitioning

In our experiments with the flow-based partitioning above, we see that the results with 3×3 repartitioning are better in most cases, as already observed by Brenner [2005]. However, in the old implementation of BonnPlace, 3×3 repartitioning was restricted to "high" effort placement runs due to its high runtime. With the new implementation of BonnPlace, 3×3 repartitioning becomes affordable and is thus used as default. For low effort placement, we maintain the option of 2×2 repartitioning.
In the new version, beyond the flow-based partitioning, possibly clustered netlists and 3×3 repartitioning, we use the local placement method (see Section 6.2.9), which yields about 0.5%-1% of netlength improvement. This method is also used by default and is contained in the results, unless otherwise noted.
Again, we first want to avoid density effects and compute placements with a 97% target density. The results are summarized in Table 8.6. The leftmost columns show the old version of BonnPlace with iterative and recursive partitioning respectively, with standard settings. The other 5 columns show the result of the new version of BonnPlace with flow-based partitioning and clustering, using *BestChoice* and different clustering ratios. Except for Felix, where the old version is slightly better, the new BonnPlace produces much shorter netlengths for all test cases. Without clustering and with moderate cluster ratios ($\alpha \leq 5$),

	BonnPlace					
	old				new	
	recursive		iterative		recursive	
Chip	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ssR
Lucius	4.54 104.3%	0:04:06 316.7%	4.56 104.8%	0:02:58 245.4%	4.35 100.0%	0:01:18 100.0%
Elisa	3.54 101.1%	0:02:09 279.4%	3.47 99.1%	0:01:22 205.9%	3.50 100.0%	0:00:46 100.0%
Felix	7.71 100.0%	0:01:51 269.5%	7.59 98.5%	0:01:26 269.4%	7.71 100.0%	0:00:41 100.0%
Rabe	16.12 102.1%	0:06:30 239.1%	15.82 100.2%	0:06:54 314.3%	15.79 100.0%	0:02:43 100.0%
Julia	13.31 100.7%	0:08:54 243.6%	13.48 102.0%	0:07:19 239.0%	13.21 100.0%	0:03:39 100.0%
Max	23.81 102.1%	0:17:03 308.4%	22.03 94.5%	0:18:35 385.7%	23.31 100.0%	0:05:32 100.0%
Roger	30.63 98.2%	0:27:54 302.2%	30.26 97.0%	0:28:42 360.2%	31.18 100.0%	0:09:14 100.0%
Patrick	53.14 100.5%	0:40:27 202.7%	51.65 97.7%	0:43:57 251.9%	52.90 100.0%	0:19:57 100.0%
Ashraf	68.33 99.6%	0:33:16 252.6%	67.32 98.1%	0:28:58 333.8%	68.61 100.0%	0:13:10 100.0%
Erhard	463.27 100.0%	2:40:30 325.6%	459.28 99.2%	1:26:28 201.5%	463.10 100.0%	0:49:18 100.0%
Arijan	552.65 99.1%	4:13:27 225.8%	550.12 98.6%	3:14:04 251.9%	557.81 100.0%	1:52:16 100.0%
Tomoku	384.56 99.6%	4:27:45 228.1%	378.87 98.2%	3:27:49 238.6%	385.91 100.0%	1:57:22 100.0%
Trips	663.69 98.7%	6:49:16 262.6%	658.87 98.0%	5:05:11 226.7%	672.38 100.0%	2:35:51 100.0%
Andre	508.40 98.9%	7:10:27 226.7%	500.36 97.3%	6:00:28 246.9%	514.13 100.0%	3:09:52 100.0%
Ludwig	651.06 101.0%	6:19:01 246.4%	654.67 101.6%	6:10:23 343.8%	644.37 100.0%	2:33:50 100.0%
Leyla	767.23 100.8%	8:54:31 248.0%	772.52 101.5%	6:54:23 268.9%	761.28 100.0%	3:35:32 100.0%
Erik	620.95 100.9%	8:04:21 194.1%	611.09 99.3%	8:46:26 276.1%	615.22 100.0%	4:09:31 100.0%
Avg.	100.4%		99.2%		100.0%	
Total		51:01:28 235.4%		43:25:22 200.3%		21:40:32 100.0%

Table 8.5: Comparison of new and old BonnPlace's global placement. BBox is the bounding box netlength in meters of the legal placement, the runtimes reflect the wall clock runtimes of global placement. All runs were performed on Intel XEON with 2.7 GHz and 64 GB and used up to 4 CPUs simultaneously with standard parameters.

the mid-size designs with large blockages on the chip area dominate the image again: on Julia, Rabe, and Roger, the relative improvement is the highest one, but also on Trips the netlength can be improved by about 10%. The scenario of $\alpha = 20$ is also interesting: even with this extreme clustering ratio, the results are comparable to the old placement, however with some significant degradations for the largest test cases. The runtimes are also convincing, even in this artificial scenario of high densities. Moderate clusterings allow a speedup of 2.5–3.8 over the old BonnPlace with an average improvement of 8.4%–6.4%.
The scenario above was made with high density targets. In real-word instances, usually much lower densities and even equal distribution are used. To this end, we performed more comparisons to address these issues. As there is no guarantee on the density compliance of the old iterative partitioning, we test the new placement against the recursive partitioning method of the old version of BonnPlace. In the old version of BonnPlace, the same scenario was used for placement with movebounds. For these tests, target densities were computed by the formula:

$$\mathcal{D}_{\text{target}} := \frac{\sum_{c \in \mathcal{C}} \text{size}(c)}{\text{free\_area}(\mathcal{A})} + 1\% \cdot \text{number of levels}. \tag{8.1}$$

Recursive partitioning accumulates rounding decisions, so we run our tool with and without an additional 2% target density to compensate for such effects. The evaluation is presented in Table 8.7 for the 2×2 repartitioning and in Table 8.8 for the 3×3 case. The runs were made on an Intel XEON with 3.3 GHz and 64 GB machine, which provides better speedups in parallel processing than the older Intel computers and used the *BestChoice* algorithm with cluster ratio 5. In addition, lower densities mean more cell movement, which leads to higher repartitioning effort and longer runtimes. The results show legalized netlength without local placement.
In the first setup (Table 8.7), the fast mode, the new BonnPlace achieves similar results to the old version, but is 8.7 times faster. On several of the larger designs, the speedup even exceeds a factor of 10. Results in the new standard mode, the second case, are shown in Table 8.8. Here, the results are significantly better while the speedup over the old version of BonnPlace still sums up to a factor of 5.4.

chip	BonnPlace													
	old				new flow-based									
	iterative		recursive		no clustering		Clustering using *BestChoice* with cluster ratio α							
							$\alpha = 2$		$\alpha = 5$		$\alpha = 10$		$\alpha = 20$	
Elisa	3.07	0:01:14	3.07	0:01:58	2.84	0:01:05	2.75	0:00:42	2.86	0:00:28	2.82	0:00:25	2.81	0:00:24
	100.0%	1.0	100.0%	0.6	92.3%	1.1	89.4%	1.8	93.0%	2.7	91.7%	3.0	91.3%	3.2
Lucius	3.91	0:02:24	4.13	0:03:00	3.56	0:01:45	3.63	0:01:03	3.73	0:00:41	3.75	0:00:37	3.83	0:00:34
	100.0%	1.0	105.5%	0.8	91.0%	1.4	92.8%	2.3	95.5%	3.5	95.8%	3.9	98.0%	4.2
Felix	7.53	0:01:21	7.55	0:01:34	7.66	0:00:52	7.6	0:00:38	7.68	0:00:25	7.7	0:00:25	7.79	0:00:25
	100.0%	1.0	100.3%	0.9	101.7%	1.6	101.0%	2.1	102.0%	3.2	102.3%	3.2	103.4%	3.3
Rabe	16.84	0:03:55	16.34	0:06:04	12.5	0:03:30	12.1	0:02:09	12.36	0:01:17	12.52	0:01:06	12.5	0:01:00
	100.0%	1.0	97.0%	0.6	74.2%	1.1	71.8%	1.8	73.4%	3.1	74.3%	3.6	74.2%	4.0
Julia	11.77	0:04:21	11.78	0:06:56	10.84	0:04:28	10.6	0:02:38	10.84	0:01:39	10.96	0:01:19	10.81	0:01:06
	100.0%	1.0	100.1%	0.6	92.1%	1.0	90.1%	1.7	92.1%	2.6	93.1%	3.3	91.8%	4.0
Max	20.8	0:06:25	19.82	0:08:40	18.98	0:06:53	18.39	0:04:10	17.99	0:02:44	18.08	0:02:24	18.1	0:02:05
	100.0%	1.0	95.3%	0.7	91.3%	0.9	88.4%	1.5	86.5%	2.3	86.9%	2.7	87.0%	3.1
Roger	30.77	0:11:00	31.94	0:16:33	27.75	0:11:27	27.63	0:06:49	27.74	0:04:07	28.02	0:03:21	28.72	0:02:48
	100.0%	1.0	103.8%	0.7	90.2%	1.0	89.8%	1.6	90.2%	2.7	91.1%	3.3	93.3%	3.9
Ashraf	63.3	0:21:40	63.73	0:23:09	58.32	0:12:43	57.97	0:08:24	61.53	0:05:22	65.96	0:04:22	70.19	0:04:06
	100.0%	1.0	100.7%	0.9	92.1%	1.7	91.6%	2.6	97.2%	4.0	104.2%	5.0	110.9%	5.3
Patrick	47.57	0:27:45	48.06	0:32:41	44.08	0:25:03	43.92	0:14:47	44.64	0:09:01	46.24	0:06:56	46.89	0:05:51
	100.0%	1.0	101.0%	0.8	92.7%	1.1	92.3%	1.9	93.8%	3.1	97.2%	4.0	98.6%	4.7
Erhard	425.92	1:10:57	421.72	1:34:14	394.57	1:26:29	398.24	0:42:17	413.64	0:25:31	436.01	0:18:19	456.07	0:14:47
	100.0%	1.0	99.0%	0.8	92.6%	0.8	93.5%	1.7	97.1%	2.8	102.4%	3.9	107.1%	4.8
Arijan	509.81	2:06:25	504.52	3:00:28	479.56	2:14:57	471.64	1:18:18	484.29	0:45:53	509.09	0:35:36	535.75	0:28:13
	100.0%	1.0	99.0%	0.7	94.1%	0.9	92.5%	1.6	95.0%	2.8	99.9%	3.6	105.1%	4.5
Philipp	363.72	3:12:04	363.72	3:12:04	343.5	2:04:36	340.15	1:11:05	342.37	0:40:38	357.49	0:31:57	376.6	0:28:12
	100.0%	1.0	100.0%	1.0	94.4%	1.5	93.5%	2.7	94.1%	4.7	98.3%	6.0	103.5%	6.8
Tomoku	360.41	3:39:38	366.99	2:46:47	346.14	1:48:27	344.6	1:10:35	354.37	0:48:57	383.06	0:43:15	423.99	0:36:54
	100.0%	1.0	101.8%	1.3	96.0%	2.0	95.6%	3.1	98.3%	4.5	106.3%	5.1	117.6%	6.0
Trips	631.72	4:15:25	631.9	5:09:00	574.65	2:48:48	573.89	1:39:42	589.69	1:04:16	604.01	0:56:30	622.35	0:45:05
	100.0%	1.0	100.0%	0.8	91.0%	1.5	90.8%	2.6	93.3%	4.0	95.6%	4.5	98.5%	5.7
Andre	472.96	4:16:01	474.91	5:05:45	442.29	3:06:08	438.4	1:55:19	448.74	1:12:02	473.78	0:56:54	519.58	0:47:20
	100.0%	1.0	100.4%	0.8	93.5%	1.4	92.7%	2.2	94.9%	3.6	100.2%	4.5	109.9%	5.4
Ludwig	616.58	4:10:02	615.69	4:23:15	599.16	1:52:16	590.55	1:20:40	603.38	0:55:29	648.04	0:48:24	725.82	0:44:10
	100.0%	1.0	99.9%	0.9	97.2%	2.2	95.8%	3.1	97.9%	4.5	105.1%	5.2	117.7%	5.7
Leyla	728.71	5:07:12	722.91	6:26:51	687.61	3:32:11	688.51	2:11:25	718.22	1:33:42	766.47	1:13:22	844.22	1:03:56
	100.0%	1.0	99.2%	0.8	94.4%	1.4	94.5%	2.3	98.6%	3.3	105.2%	4.2	115.9%	4.8
Erik	568.14	6:13:18	571.44	6:30:04	545.53	3:24:23	547.43	2:17:54	547.43	1:29:08	605.05	1:09:01	674.29	1:01:56
	100.0%	1.0	100.6%	1.0	96.0%	1.8	96.4%	2.7	96.4%	4.2	106.5%	5.4	118.7%	6.0
	100.0%		100.2%		92.4%		91.6%		93.6%		97.2%		101.6%	
		35:31:08		39:49:04		23:26:01		14:28:35		9:21:20		7:34:13		6:28:52
		1.0		0.9		1.5		2.5		3.8		4.7		5.5

Table 8.6: Comparison between old BonnPlace with different partitioning algorithms and the new BonnPlace with the flow-based partitioning and different cluster ratios using *BestChoice*. The placements were done with high density of 97% to fade out density differences. All runtimes are global placement runtimes, the legalized netlengths are provided. The tests ran on Intel XEON with 3.0 GHz and 64 GB in parallel mode.

8.4 New BonnPlace with Movebounds: Recursive vs. Flow-based

In this section we compare the flow-based partitioning and the recursive partitioning method (Algorithm 4) on instances with movebounds using the new BonnPlace. As soft movebounds are interpreted as hard ones (with possibly enlarged area) we restrict ourselves to runs with hard movebounds.

The test suite for movebounds contains 11 instances with various numbers of constraints. Several instances have been created from flattening existing hierarchies, where for each flattened RLM, its area became a movebound and the cells within the flattened RLM were associated to this movebound. Others were defined by experienced designers for different purposes. The test suite is summarized in Table 8.9. All these tests were performed on an Intel XEON with 3.0 GHz and 64 GB in parallel mode and used up to 8 CPUs.

To fade-out any density issues, we again set the target density to 97%. The results on instances with movebounds are presented in Table 8.10 for the inclusive hard movebounds and in Table 8.11 for the exclusive hard ones. The recursive partitioning used 2×2 repartitioning, and the new flow-based partitioning used 3×3 by default.

In the case of inclusive hard movebounds, the new flow-based partitioning shows a remarkable netlength improvement of more than 10% on average. On Ashraf, Valentin and Trips the improvements exceed even 10%, on Erik 20% and on Rabe more than 25%.

The price for this is a runtime loss of about one third, which can easily be compensated by the moderate application of clustering, as shown in the rightmost column. With this strategy, the overall gain over the recursive partitioning can almost be kept, but clustering with *BestChoice* provides and additional speedup of 1.7 over a standard recursive partitioning setup.

A similar situation shows up in the case of exclusive movebounds. Again, dominated by the results on Rabe with an improvement of more than 25%, flow-based partitioning performs better in terms of netlength by more than 13%. The runtime loss is again moderate and can be fully compensated by clustering. In total, on exclusive hard movebound instances, the new default algorithm is twice as fast as the old one and produces placements with netlengths improved by 11.4% on average.

Finally, we show that the legalization routine by Brenner and Vygen [2004] has a significant part in total placement runtime. In the past, legalization runtime contributed with usually less than 5% to the total placement runtime. This is no longer the case in the presence of movebounds. Table 8.12 shows the runtimes on instances with movebounds. Legalization becomes a relatively expensive routine in the new version of BonnPlace and takes almost 50% of total placement runtime. On Ashraf and Andre it is even the dominating algorithm with 72% and 58% of total placement runtime respectively.

Table 8.13 shows the runtimes on instances with exclusive hard movebounds. Here, legalization contributes more than 30% to the total placement runtime. This number is smaller compared to the inclusive hard case because Multisection is not needed to assign cells

chip	target density	BonnPlace					
		old recursive 2x2 repartitioning		new fast 2x2 repartitioning			
						+2% target density	
		BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss
Elisa		3.71	0:01:47	3.95	0:00:21	3.68	0:00:20
	63.46%	100.0%	1.0	106.4%	5.1	99.3%	5.4
Lucius		6.46	0:03:29	5.81	0:00:34	5.42	0:00:33
	55.10%	100.0%	1.0	89.9%	6.1	83.9%	6.3
Felix		7.87	0:01:43	8	0:00:18	7.98	0:00:16
	82.29%	100.0%	1.0	101.6%	5.7	101.4%	6.4
Rabe		17.13	0:05:49	17.51	0:00:55	16.26	0:00:47
	61.24%	100.0%	1.0	102.2%	6.3	94.9%	7.4
Julia		18.32	0:07:08	15.89	0:00:52	16.16	0:00:48
	58.26%	100.0%	1.0	86.7%	8.2	88.2%	8.9
Max		23.81	0:11:07	23.59	0:01:18	22.32	0:01:13
	65.05%	100.0%	1.0	99.1%	8.6	93.7%	9.1
Roger		32.45	0:16:51	33.11	0:01:51	32.45	0:01:51
	58.33%	100.0%	1.0	102.0%	9.1	100.0%	9.1
Ashraf		79.86	0:24:12	81.56	0:02:32	78.57	0:02:31
	45.77%	100.0%	1.0	102.1%	9.6	98.4%	9.6
Patrick		54.56	0:31:37	59.00	0:09:00	58.53	0:08:20
	61.56%	100.0%	1.0	108.1%	3.5	107.3%	3.8
Erhard		537.64	1:22:41	579.14	0:08:28	575.81	0:08:20
	48.63%	100.0%	1.0	107.7%	9.8	107.1%	9.9
Arijan		640.22	2:06:18	658.82	0:14:21	640.96	0:14:57
	43.58%	100.0%	1.0	102.9%	8.8	100.1%	8.4
Philipp		422.96	2:10:46	435.98	0:15:49	437.26	0:15:06
	62.32%	100.0%	1.0	103.1%	8.3	103.4%	8.7
Tomoku		490.28	2:45:23	476.14	0:21:30	466.05	0:21:55
	45.24%	100.0%	1.0	97.1%	7.7	95.1%	7.5
Valentin		869.51	4:17:31	864.02	0:25:30	836.42	0:26:09
	65.37%	100.0%	1.0	99.4%	10.1	96.2%	9.8
Trips		702.46	3:47:06	760.6	0:21:53	744.12	0:22:04
	56.56%	100.0%	1.0	108.3%	10.4	105.9%	10.3
Andre		691.52	4:50:57	667.52	0:29:08	653.27	0:28:13
	37.99%	100.0%	1.0	96.5%	10.0	94.5%	10.3
Ludwig		758.79	3:47:31	736.42	0:26:47	750.44	0:26:04
	52.57%	100.0%	1.0	97.1%	8.5	98.9%	8.7
Leyla		925.70	5:33:21	971.86	0:38:31	957.01	0:39:00
	44.31%	100.0%	1.0	105.0%	8.7	103.4%	8.5
Erik		734.02	4:56:50	810.62	0:37:00	721.04	0:38:33
	56.67%	100.0%	1.0	110.4%	8.0	98.2%	7.7
Average		100.0%		101.2%		98.2%	
Total			37:22:07		4:16:38		4:17:00
			1.0		8.7		8.7

Table 8.7: Old BonnPlace with recursive partitioning compared to new FlowBased-Partitioning. In magenta the speedups over old recursive are provided. Runtimes are global placement wall-clock runtimes, BBox shows the legalized netlength. Runs were made on Intel XEON with 3.3 GHz and 64 GB.

chip	target density	BonnPlace					
		old recursive 2x2 repartitioning		new standard 3x3 repartitioning			
						+2% target density	
		BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss
Elisa		3.71	0:01:47	3.57	0:00:27	3.28	0:00:25
	63.46%	100.0%	1.0	96.2%	4.0	88.5%	4.3
Lucius		6.46	0:03:29	5.25	0:00:29	5.21	0:00:30
	55.10%	100.0%	1.0	81.2%	7.2	80.6%	7.0
Felix		7.87	0:01:43	7.78	0:00:21	7.68	0:00:22
	82.29%	100.0%	1.0	98.8%	4.9	97.5%	4.7
Rabe		17.13	0:05:49	16.39	0:01:01	15.83	0:01:01
	61.24%	100.0%	1.0	95.7%	5.7	92.4%	5.7
Julia		18.32	0:07:08	14.81	0:01:23	13.42	0:01:21
	58.26%	100.0%	1.0	80.8%	5.2	73.3%	5.3
Max		23.81	0:11:07	21.08	0:01:50	21.13	0:01:56
	65.05%	100.0%	1.0	88.5%	6.1	88.7%	5.8
Roger		32.45	0:16:51	30.42	0:03:48	29.90	0:03:48
	58.33%	100.0%	1.0	93.7%	4.4	92.1%	4.4
Ashraf		79.86	0:24:12	79.17	0:05:01	75.63	0:04:54
	45.77%	100.0%	1.0	99.1%	4.8	94.7%	4.9
Patrick		54.56	0:31:37	54.89	0:05:38	53.65	0:05:43
	61.56%	100.0%	1.0	100.6%	5.6	98.3%	5.5
Erhard		537.64	1:22:41	546.80	0:14:10	536.58	0:14:28
	48.63%	100.0%	1.0	101.7%	5.8	99.8%	5.7
Arijan		640.22	2:06:18	633.18	0:25:02	607.82	0:25:08
	43.58%	100.0%	1.0	98.9%	5.0	94.9%	5.0
Philipp		422.96	2:10:46	411.68	0:26:18	406.82	0:27:25
	62.32%	100.0%	1.0	97.3%	5.0	96.2%	4.8
Tomoku		490.28	2:45:23	451.79	0:37:53	438.88	0:37:03
	45.24%	100.0%	1.0	92.1%	4.4	89.5%	4.5
Valentin		869.51	4:17:31	823.60	0:43:45	799.24	0:42:26
	65.37%	100.0%	1.0	94.7%	5.9	91.9%	6.1
Trips		702.46	3:47:06	694.85	0:37:51	689.23	0:39:35
	56.56%	100.0%	1.0	98.9%	6.0	98.1%	5.7
Andre		691.52	4:50:57	624.58	0:48:53	605.86	0:49:04
	37.99%	100.0%	1.0	90.3%	6.0	87.6%	5.9
Ludwig		758.79	3:47:31	722.51	0:38:52	706.13	0:41:57
	52.57%	100.0%	1.0	95.2%	5.9	93.1%	5.4
Leyla		925.70	5:33:21	921.73	0:57:48	900.72	0:58:12
	44.31%	100.0%	1.0	99.6%	5.8	97.3%	5.7
Erik		734.02	4:56:50	692.19	1:07:28	680.23	0:57:15
	56.67%	100.0%	1.0	94.3%	4.4	92.7%	5.2
Average		100.0%		94.4%		91.7%	
Total			37:22:07		6:57:58		6:52:33
			1.0		5.4		5.4

Table 8.8: Old BonnPlace with recursive partitioning compared to new FlowBased-Partitioning. In magenta the speedups over old recursive are provided. Runtimes are global placement wall-clock runtimes, BBox shows the legalized netlength. Runs were made on Intel XEON with 3.3 GHz and 64 GB.

Chip	$\lvert\mathcal{M}\rvert$	$\lvert\mathcal{C}\rvert$	% cells with movebounds	max cell density in a movebound	remarks
Lucius	1	77 679	88.3%	52%	
Felix	3	84 966	63.0%	70%	
Rabe	2	175 646	4.3%	67%	
Ashraf	206	866 777	22.0%	92%	flattened hierarchy
Erhard	43	2 578 246	97.8%	74%	
Philipp	2	3 945 833	16.3%	50%	non-rectangular
Tomoku	85	5 296 120	1.2%	74%	overlapping, flattened hierarchy
Trips	114	5 747 007	99.4%	81%	overlapping
Andre	43	6 794 323	3.8%	73%	flattened hierarchy
Ludwig	33	7 500 446	2.7%	70%	overlapping, flattened hierarchy
Erik	39	9 316 938	84.6%	85%	flattened hierarchy

Table 8.9: Summary of instances with movebounds.

Inclusive Hard Movebound Tests						
chip	new BONNPLACE Partitining Method:					
	recursive 2 × 2 repartitioning		flow-based 3 × 3 repartitioning			
					BestChoice $\alpha = 5$	
	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss
Lucius	3.95	0:00:46	3.71	0:01:42	3.66	0:00:50
	100.0%	1.0	94.0%	0.5	92.9%	0.9
Felix	8.14	0:00:30	7.54	0:01:06	7.54	0:00:43
	100.0%	1.0	92.6%	0.5	92.6%	0.7
Rabe	16.95	0:02:02	12.59	0:03:56	12.44	0:01:20
	100.0%	1.0	74.3%	0.5	73.4%	1.5
Ashraf	69.27	0:12:05	59.71	0:19:41	61.59	0:09:32
	100.0%	1.0	86.2%	0.6	88.9%	1.3
Erhard	509.74	0:36:31	480.01	0:52:17	499.31	0:24:52
	100.0%	1.0	94.2%	1.0	98.0%	1.5
Philipp	384.39	1:33:56	345.63	2:16:17	355.32	0:46:47
	100.0%	1.0	89.9%	0.7	92.4%	2.0
Tomoku	382.55	1:22:13	360.56	2:15:23	367.00	0:47:44
	100.0%	1.0	94.3%	0.6	95.9%	1.7
Trips	688.24	2:11:16	619.40	3:04:19	611.58	1:14:51
	100.0%	1.0	90.0%	0.7	88.9%	1.8
Andre	504.81	2:14:03	467.38	3:28:28	462.23	1:35:41
	100.0%	1.0	92.6%	0.6	91.6%	1.4
Ludwig	642.28	2:05:23	613.23	1:59:50	624.09	1:12:23
	100.0%	1.0	95.5%	1.0	97.2%	1.7
Erik	720.54	2:56:28	582.17	3:45:54	598.15	1:30:24
	100.0%	1.0	80.8%	0.8	83.0%	2.0
Average	100.0%		89.2%		90.2%	
Total		13:15:15		18:08:54		07:45:08
		1.0		0.7		1.7

Table 8.10: Comparison between different partitioning methods in new BONNPLACE on instances with inclusive hard movebounds and 97% target density. The runtimes are global placement wall clock runtimes, the legalized bounding box netlength is provided.

Exclusive Hard Movebound Tests						
chip	new BonnPlace Partitining Method:					
	recursive 2 × 2 repartitioning		flow-based 3 × 3 repartitioning			
					BestChoice $\alpha = 5$	
	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss
Rabe	17.26	0:03:02	12.83	0:03:03	12.85	0:01:18
	100.0%	1.0	74.3%	1.0	74.5%	2.3
Ashraf	71.54	0:08:43	61.73	0:13:25	64.44	0:06:13
	100.0%	1.0	86.3%	0.7	90.1%	1.6
Erhard	534.42	0:47:12	505.69	0:58:37	518.64	0:20:03
	100.0%	1.0	94.6%	0.8	97.0%	2.3
Andre	516.40	2:21:58	484.39	3:04:24	494.04	1:11:54
	100.0%	1.0	93.8%	0.8	95.7%	2.0
Erik	713.52	2:52:34	600.04	3:19:13	624.82	1:24:00
	100.0%	1.0	84.1%	0.9	87.6%	2.1
Average	100.0%		86.3%		88.6%	
Total		6:14:28		7:38:41		3:03:28
		1.0		0.8		2.0

Table 8.11: Comparison between different partitioning methods in new BonnPlace on instances with exclusive hard movebounds and 97% target density. The runtimes are global placement wall clock runtimes, the legalized bounding box netlength is provided.

to legalization regions and the backup strategy for legalization with lower densities does not apply.
However, further runtime improvement for the overall placement is possible by accelerating the legalization part, for example by processing regions in parallel.

8.5 Comparison to an Industrial Tool

We tested the outcome of our tool also against the results produced by a state-of-the-art industrial placer. The implementation of the industrial tool follows Viswanathan et al. [2007]. There have been huge differences in terms of density, so once again, to avoid density effects we only compare placements with 97% target density. In our tests, to obtain a better comparison, both the industrial tool as well as our placer used *BestChoice* clustering with a cluster ratio of $\alpha = 5$ (Alpert et al. [2005]). BonnPlace runs in default settings and use the local placement. The results are presented in Table 8.14 on instances without movebounds.

In total, the tools produce comparable results on average in terms of netlength, but BonnPlace is 5.5 times faster. When considering particular results, however, the deviations of both tools are remarkable. On the small test cases Lucius and Elisa, the netlengths computed by the industrial tool are more than 9.2% and 9.8% shorter. On the small chip Dagmar and the large test cases Arijan and Valentin, BonnPlace performs significantly better, with an improvement of 16.8%, 10.8% and 9.1% respectively.
The results with movebounds produced by our tool significantly outperform the industrial

New BonnPlace: Inclusive Hard Movebound Tests

Chip	Runtime (hh:mm:ss)			Relative legalization runtime
	Global	Detailed	Total	50.7 %
Lucius	0:00:50	0:00:52	0:01:42	24.4 %
Felix	0:00:43	0:00:14	0:00:57	32.2 %
Rabe	0:01:20	0:00:38	0:01:58	72.1 %
Ashraf	0:09:32	0:24:40	0:34:12	36.5 %
Erhard	0:24:52	0:14:19	0:39:11	23.5 %
Philipp	0:46:47	0:14:23	1:01:11	48.2 %
Tomoku	0:47:44	0:44:29	1:32:12	49.4 %
Trips	1:14:51	1:13:08	2:27:59	49.3 %
Andre	1:35:41	1:33:09	3:08:50	58.3 %
Ludwig	1:12:23	1:41:08	2:53:31	48.6 %
Erik	1:30:24	1:25:26	2:55:50	49.3 %
Total	7:45:08	7:32:26	15:17:34	49.3 %

Table 8.12: Runtime distributions on runs with new BonnPlace with inclusive hard movebounds. Runs were made on an Intel XEON with 3.0 GHz and 64 GB using up to 8 CPUs. Runtimes are wall clock runtimes.

New BonnPlace: Exclusive Hard Movebound Tests

Chip	Runtime (hh:mm:ss)			Relative legalization runtime
	Global	Detailed	Total	
Rabe	0:01:18	0:00:23	0:01:41	22.8 %
Ashraf	0:06:13	0:09:04	0:15:17	59.3 %
Erhard	0:20:03	0:09:58	0:30:01	33.2 %
Andre	1:11:54	0:27:21	1:39:15	27.6 %
Erik	1:33:48	0:36:43	2:00:43	30.4 %
Total	3:03:28	1:23:29	4:26:57	31.3%

Table 8.13: Runtime distributions on runs with new BonnPlace with exclusive hard movebounds. Runs were made on an Intel XEON with 3.0 GHz and 64 GB using up to 8 CPUs. Runtimes are wall clock runtimes.

tool, as shown in Tables 8.15 and 8.16. For tests with inclusive hard movebounds, the placement produced by BonnPlace is 32% shorter, and the tool is 9.5 times faster. For the exclusive movebounds, the gap is 32.9% in favor of BonnPlace and our tool is more than 20 times faster. The reliability of BonnPlace is also convincing: while the industrial tool failed twice completely (in case of Philipp and Ashraf) on instances with inclusive hard movebounds and produced several violations on Tomoku, Andre and Erik (361, 62 and 4655 respectively). The placements produced by BonnPlace were legal in each of the tests. On instances with exclusive hard movebounds the industrial tool produced illegal outcomes on Andre and Erik showing 463 and 5630 violations, respectively.

8.6 Benchmarks

Apart from industrial instances, we have also tested our tool on the latest international benchmarks, as proposed in ISPD 20005 (Nam et al. [2005]) and ISPD 2006 (Nam et al. [2006]) and compared the results to several of the placers presented in Chapter 3. We cite the results of other placers from the papers of Spindler, Schlichtmann and Johannes [2008] and Agnihotri and Madden [2007].

On ISPD 2005 benchmarks (see Table 8.17), where only bounding box netlength is evaluated, we run BonnPlace in high-effort mode with *BestChoice* cluster ratio 2 and applied local placement. BonnPlace is the fourth best out of twelve. Only NTU3 (Chen et al. [2008]), Aplace2 (Kahng and Wang [2006]) and RQL (Viswanathan et al. [2007]) produce shorter netlengths.

In the ISPD 2006 benchmarks (Nam et al. [2006]), a combined score function is evaluated. In this score function, bounding box netlength is measured. Then, density violations are counted, where the density is measured by a sliding window of 10×10 standard-cell heights. The relative scaled number of density violations is then added to the objective. Finally, runtime is evaluated and set in relation to the median of all placement runtimes: the runtime penalty (bonus) x is computed by the formula $\mathrm{x} = 0.04 \log_2(\frac{\text{tool cpu}}{\text{cpu median over all tools}})$, which roughly increases the netlength/density score by 4% if the tool is twice as slow as the median. On the other hand, the netlength/density score can be decreased by 4% if the placement is twice as fast as the median of others. The runtime penalties are bounded by $\pm 10\%$. For details we refer to Nam et al. [2006].
To obtain comparable runtimes, we also used an Opteron 852 machine. We applied *BestChoice* clustering with a ratio of $\alpha = 2$ and allowed parallel processing with 8 CPUs. In Table 8.18 we show detailed results of our tool and of the leading tool on this benchmark set, Kraftwerk2 (Spindler, Schlichtmann and Johannes [2008]).
All these test cases, except newblue1, only consist of standard cells. In the case of newblue1, which contains one large block and several macros, we ran macro placement to place the largest block (Phase 1 of MacroPlacement in Brenner [2005]), fixed the largest instance and then ran global placement with legalization with floating large blocks. An explicit

chip	industrial tool		new BonnPlace	
	BBox(m)	hh:mm:ss	Box(m)	hh:mm:ss
Dagmar	0.95 100.0%	0:02:13 1.0	0.80 83.2%	0:00:40 3.3
Elisa	2.60 100.0%	0:02:35 1.0	2.86 109.8%	0:00:36 4.4
Lucius	3.42 100.0%	0:03:30 1.0	3.73 109.2%	0:01:48 1.9
Felix	8.17 100.0%	0:03:05 1.0	7.68 94.0%	0:00:36 5.2
Paula	3.14 100.0%	0:06:30 1.0	3.21 102.3%	0:01:41 3.9
Rabe	12.42 100.0%	0:08:09 1.0	12.36 99.6%	0:01:44 4.7
Julia	10.65 100.0%	0:08:39 1.0	10.84 101.8%	0:02:15 3.9
Max	17.22 100.0%	0:18:00 1.0	17.99 104.5%	0:06:25 2.8
Roger	27.42 100.0%	0:22:21 1.0	27.74 101.2%	0:10:37 2.1
Ashraf	61.05 100.0%	0:49:03 1.0	61.53 100.8%	0:09:54 5.0
Patrick	43.84 100.0%	1:00:13 1.0	44.64 101.8%	0:12:24 4.9
Erhard	463.76 100.0%	2:24:24 1.0	413.64 89.2%	0:32:45 4.4
Arijan	485.04 100.0%	3:30:35 1.0	484.29 99.8%	0:59:36 3.5
Philipp	358.91 100.0%	4:22:42 1.0	342.37 95.4%	0:54:52 4.8
Tomoku	356.44 100.0%	7:51:32 1.0	354.37 99.4%	1:10:10 6.7
Trips	616.05 100.0%	6:31:12 1.0	589.69 95.7%	1:25:13 4.6
Valentin	671.49 100.0%	8:15:01 1.0	610.40 90.9%	1:36:21 5.1
Andre	437.01 100.0%	9:16:52 1.0	448.74 102.7%	1:38:04 5.7
Ludwig	598.40 100.0%	9:19:11 1.0	603.38 100.8%	1:30:30 6.2
Leyla	711.90 100.0%	13:43:02 1.0	718.22 100.9%	2:09:18 6.4
Erik	559.34 100.0%	13:25:50 1.0	547.43 97.9%	2:07:02 6.3
Average	100.0%		99.3%	
Total		81:44:09 1.0		14:52:29 5.5

Table 8.14: Comparison of the industrial tool to the new BonnPlace. Total wall clock runtimes and final bounding box netlengths are provided. The runs were made on Intel XEON with 3.0 GHz and 64 GB in parallel mode with 97% target density and standard settings.

chip	Inclusive Hard Movebounds industrial tool BBox(m)	hh:mm:ss	new BonnPlace Box(m)	hh:mm:ss
Lucius	4.55	00:03:30	3.66	00:01:42
	100.0%	1.0	80.5%	2.1
Felix	9.03	00:05:38	7.54	00:00:57
	100.0%	1.0	83.5%	5.9
Rabe	16.68	00:10:22	12.44	00:01:58
	100.0%	1.0	74.6%	5.3
Erhard	549.62	04:49:16	499.31	00:39:11
	100.0%	1.0	90.8%	7.4
Tomoku	*737.34	13:20:44	367	01:32:13
	100.0%	1.0	49.8%	8.7
Trips	703.88	11:37:42	611.58	02:27:59
	100.0%	1.0	86.9%	4.7
Andre	*1023.65	37:36:58	462.23	03:08:50
	100.0%	1.0	45.2%	12.0
Ludwig	1207.63	26:22:04	624.09	02:53:31
	100.0%	1.0	51.7%	9.1
Erik	*879.12	36:44:33	598.15	02:55:50
	100.0%	1.0	68.0%	12.5
Average	100.0%		68.0%	
Total		130:50:47		13:42:11
		1.0		9.5

Table 8.15: Comparison of the industrial tool to the new BonnPlace. Total wall-clock runtimes and final bounding box netlengths are provided. The runs were made on Intel XEON with 3.0 GHz and 64 GB in parallel mode with 97% target density and standard settings. Instances marked with (*) contain violations, see Section 8.4.

chip	Exclusive Hard Movebounds industrial tool BBox(m)	hh:mm:ss	new BonnPlace Box(m)	hh:mm:ss
Rabe	16.73	00:11:18	12.85	00:01:41
	100.0%	1.0	76.8%	6.7
Ashraf	93.21	02:09:14	64.44	00:15:17
	100.0%	1.0	69.1%	8.5
Erhard	633.64	04:25:35	518.64	00:30:01
	100.0%	1.0	81.9%	8.8
Andre	*1144.21	47:08:58	494.04	01:41:37
	100.0%	1.0	43.2%	27.8
Erik	*864.74	**43:20:47	624.82	02:10:34
	100.0%	1.0	72.3%	19.9
Average	100.0%		67.1%	
Total		97:15:52		04:39:10
		1.0		20.9

Table 8.16: Comparison of the industrial tool to the new BonnPlace. Total wall-clock runtimes and final bounding box netlengths are provided. The runs were made on Intel XEON with 3.0 GHz and 64 GB in parallel mode with 97% target density and standard settings. Instances marked with (*) contain violations, see Section 8.4. (**) The run was made in sequential mode, as the parallel one faild as it required more than 220 GB memory.

macro placement was too time consuming. To all other test cases, standard BonnPlace was applied. When considering netlength and density violation only, BonnPlace is slightly better than Kraftwerk, which produces better results on newblue1 and newblue4. Our tool performs better on the larger instances, such as adaptec5 and newblue5-7. Both tools, Kraftwerk2 and BonnPlace, are fast and obtain their rankings by a significant speedup over the other tools and reach or exceed the maximal runtime bonuses in several cases. Without bounds, BonnPlace's real runtime bonuses would be -11.1% (newblue1), -10.5% (newblue4) and -11.0% (newblue5).
The results on these benchmarks produced by other placement tools are summarized in Table 8.19, showing that BonnPlace is currently the best placement tool in this objective.

ISPD 2005 Benchmarks

	RQL		Aplace2		NTU3		BonnPlace		Kraftwerk2		Vasstu(II)	
adaptec2	88.51	0.97	87.31	0.95	89.85	0.98	91.61	1.00	92.85	1.01	92.97	1.01
adaptec4	188.86	0.99	187.65	0.98	193.74	1.01	191.19	1.00	199.43	1.04	192.06	1.00
bigblue1	94.98	0.93	94.64	0.93	97.28	0.96	101.66	1.00	97.67	0.96	98.09	0.96
bigblue2	150.03	0.96	143.82	0.92	152.20	0.98	155.63	1.00	154.74	0.99	153.43	0.99
bigblue3	323.09	0.94	357.89	1.04	348.48	1.02	343.14	1.00	343.32	1.00	370.72	1.08
bigblue4	797.66	0.95	833.21	0.99	829.16	0.99	841.69	1.00	852.4	1.01	828.25	0.98
		0.957		0.970		0.988		1.000		1.004		1.005

	FastPlace		mFAR		Dragon		mPL6		Capo		FengShui	
adaptec2	93.08	1.02	91.53	1.00	94.72	1.03	97.11	1.06	99.71	1.09	122.99	1.34
adaptec4	201.36	1.05	190.84	1.00	200.88	1.05	200.94	1.05	211.25	1.1	337.22	1.76
bigblue1	95.68	0.94	97.7	0.96	102.39	1.01	98.31	0.97	108.21	1.06	114.57	1.13
bigblue2	155.1	1.00	168.7	1.08	159.71	1.03	173.22	1.11	172.3	1.11	285.43	1.83
bigblue3	379.88	1.11	379.95	1.11	380.45	1.11	369.66	1.08	382.63	1.12	471.15	1.37
bigblue4	832.88	0.99	876.29	1.04	903.96	1.07	904.19	1.07	1098.76	1.31	1040.05	1.24
		1.016		1.031		1.050		1.056		1.128		1.423

Table 8.17: Results on ISPD 2005 benchmarks (Nam et al. [2005]). Comparison between legal bounding-box netlengths of different placement tools from academia.

ISPD 2006 Benchmarks												
	BonnPlace					Kraftwerk2					ratio	
	BBOX (B)	DENS (D)	CPU (C)	(B+D)	(B+C+D)	BBOX (B)	DENS (D)	CPU (C)	B+D	B+C+D	B+D	B+C+D
adaptec5	430.43	1.81	-9.52%	438.22	396.5	433.84	3.61	-9.35%	449.48	407.46	102.6 %	102.8%
newblue1	69.05	2.04	-10.00%	70.46	63.42	65.92	0.45	-8.38%	66.22	60.67	94.0 %	95.7%
newblue2	200.77	1.92	-8.16%	204.62	187.92	203.91	1.29	-10.00%	206.53	185.88	100.9 %	98.9%
newblue3	273.48	1.15	-8.25%	276.63	253.82	278.51	0.38	-10.00%	279.57	251.62	101.1 %	99.1%
newblue4	313.72	2.27	-10.00%	320.83	288.75	304.24	1.71	-8.63%	309.44	282.74	96.5 %	97.9%
newblue5	545.82	1.31	-10.00%	552.96	497.66	548.38	2.69	-9.50%	563.15	509.65	101.8 %	102.4%
newblue6	520.19	1.42	-9.42%	527.59	477.92	528.59	1.70	-9.89%	537.59	484.42	101.9 %	101.4%
newblue7	1075.98	0.97	-8.35%	1086.4	995.7	1126.58	3.16	-9.06%	1162.12	1056.84	107.0 %	106.1%
											100.6 %	100.5%

Table 8.18: Results on ISPD 2006 benchmarks (see Nam et al. [2006]). Detailed comparison between BONNPLACE and until now the best placement tool w.r.t. ISPD2006 score function, Kraftwerk2 (Spindler, Schlichtmann and Johannes [2008]). The benchmark score function consists of Bounding Box wirelength (B), density penalty (D) and runtime (C).

ISPD 2006 Benchmarks		
	(B+D)	(B+D+C)
BONNPLACE	100.0%	100.0 %
Kraftwerk2 (Spindler, Schlichtmann and Johannes [2008])	100.6%	100.5 %
NTUPlace3 (Chen et al. [2008])	96.5%	104.4 %
RQL (Viswanathan et al. [2007])	97.5%	n.a
FastPlace (Viswanathan and Chu [2005])	n.a.	109.7 %
mPL6 (Chan et al. [2006])	97.7%	109.7 %
mFAR (Hu, Zeng and Marek-Sadowska [2005])	106.1%	116.8 %
Aplace (Kahng and Wang [2006])	106.1%	122.9 %
Dragon (Wang, Yang and Sarrafzadeh [2000])	124.6%	129.9 %
Dplace (Luo and Pan [2008])	135.5%	143.8 %
Capo (Roy et al. [2005])	128.8%	146.1 %

Table 8.19: Results on ISPD 2006 benchmarks (see Nam et al. [2006]). Comparison of the average benchmark scores involving Bounding Box wirelength (B), density penalty (D) and runtime (C).

8.7 Clustering

By default, we use the *BestChoice* algorithm for clustering, as long as the random walk based clustering is in the preliminary stage. Table 8.20 shows the comparison between different clustering ratios of *BestChoice* with BonnPlace in standard settings, with a target density of 70%. Unlike the results on the placement proposed by Nam et al. [2006], clustering reduces netlengths in BonnPlace only with small clustering ratios. A clustering ratio of $\alpha = 5$ already leads to inferior results. On the other hand, we can confirm the achievable speedups with our tool.

8.8 Random Walk Clustering

Table 8.21 summarizes the runtimes of parallel hitting time computations in relation to the global placement runtimes (including hitting time computation and clustering) with cluster ratio $\alpha = 10$. The parallel random walk computation to 16 targets takes about 1 minute per million cells for large designs on average (on an Intel XEON with 3.3 GHz and 64 GB). For the small and mid-size designs, the cache allows faster random walk computation on the one hand, on the other hand the clustering has a lower impact on total runtime in global placement. Consequently, for small and mid-size designs, the random walk computation takes less than 5% of the global placement runtime, for the largest designs the ratio varies between 8.7% and 13.9%, which is still acceptable. The parallel computations with 8 CPUs yield speedups of 5 and better, when computing 16 targets. After the hitting times have been computed, a modified *BestChoice* algorithm is used for bottom-up clustering. The rank function (7.20) used here leads to situations, where for a cell c and its best neighbor c', the $\text{rank}_{RW'} = -\infty$. In this case, c, c' are not clustered and the clustering stops before reaching the desired cluster ratio. The combination of global random walk information in form of hitting times and bottom-up clustering shows very promising results. Despite the simple target choice and the preliminary implementation, the presented algorithm significantly outperforms *BestChoice* in terms of netlength (see Tab. 8.22). In all cases, BonnPlace with random walk clustering produces significantly better results than with *BestChoice* in all cases. The gap to *BestChoice* is 6.5% in favor of our routine on average, maximal deviation even exceeded 12%. Interesting is also the comparison to the unclustered runs: on average, the random walk clustering improves the placement netlength by 1.5%, where the major improvement was obtained on the small and mid-size designs, where the improvement exceeded even 10% in several cases. For the large designs, it seems that the random walk information helps already to prevent several bad clustering decisions, but it is insufficient to obtain an overall netlength improvement. When considering the overall placement runtimes, one can see that placement with *BestChoice* achieves a speedup of 3.5 on average, comparing to the unclustered run. Random walk clustering is 2.1 times faster than the unclustered run.

	BonnPlace (target density 70%)									
	no clustering		Clustering using the *BestChoice* algorithm (Alpert et al. [2005]) with cluster ratio α							
			$\alpha = 2$		$\alpha = 5$		$\alpha = 10$		$\alpha = 20$	
chip	Netl	Runtime	Netl	Runtime	Netl	Runtime	Netl	Runtime	Netl	Runtime
Elisa	3.45	0:01:01	3.35	0:00:40	3.39	0:00:28	3.34	0:00:24	3.34	0:00:21
	100.0%	1.0	97.2%	1.5	98.3%	2.2	96.8%	2.5	96.9%	2.9
Lucius	3.97	0:01:24	4.07	0:00:55	4.05	0:00:41	4.22	0:00:33	4.26	0:00:31
	100.0%	1.0	102.5%	1.5	102.2%	2.0	106.4%	2.5	107.4%	2.7
Felix	7.92	0:00:54	7.93	0:00:36	8.25	0:00:30	8.21	0:00:27	8.37	0:00:27
	100.0%	1.0	100.1%	1.5	104.1%	1.8	103.6%	2.0	105.6%	2.0
Rabe	14.47	0:03:14	14.08	0:01:53	14.29	0:01:21	14.34	0:01:02	14.55	0:00:58
	100.0%	1.0	97.3%	1.7	98.7%	2.4	99.1%	3.1	100.6%	3.3
Julia	13.36	0:03:56	12.77	0:02:27	13.06	0:01:43	13.28	0:01:12	13.95	0:01:02
	100.0%	1.0	95.6%	1.6	97.7%	2.3	99.4%	3.3	104.4%	3.8
Max	22.64	0:06:16	21.69	0:03:57	22.42	0:02:44	21.82	0:02:13	22.25	0:02:02
	100.0%	1.0	95.8%	1.6	99.0%	2.3	96.4%	2.8	98.3%	3.1
Roger	31.76	0:10:40	31.55	0:06:13	31.18	0:04:15	32.33	0:03:09	32.38	0:02:46
	100.0%	1.0	99.3%	1.7	98.2%	2.5	101.8%	3.4	102.0%	3.9
Ashraf	67.5	0:13:07	66.48	0:09:06	70.89	0:05:36	76.16	0:04:35	80.66	0:04:16
	100.0%	1.0	98.5%	1.4	105.0%	2.3	112.8%	2.9	119.5%	3.1
Patrick	50.13	0:23:01	50.18	0:13:21	51.35	0:08:33	54.72	0:06:40	55.58	0:05:27
	100.0%	1.0	100.1%	1.7	102.4%	2.7	109.2%	3.5	110.9%	4.2
Erhard	438.93	1:19:43	443.96	0:43:22	462.91	0:24:23	489.88	0:18:20	516.2	0:15:14
	100.0%	1.0	101.1%	1.8	105.5%	3.3	111.6%	4.3	117.6%	5.2
Arijan	520.85	2:15:49	511.51	1:18:58	528.88	0:45:24	565.58	0:34:59	596.85	0:27:34
	100.0%	1.0	98.2%	1.7	101.5%	3.0	108.6%	3.9	114.6%	4.9
Philipp	382.22	2:23:38	381.41	1:22:40	384.8	0:44:52	398.81	0:32:43	428.24	0:27:48
	100.0%	1.0	99.8%	1.7	100.7%	3.2	104.3%	4.4	112.0%	5.2
Tomoku	364.96	2:14:34	367.75	1:28:08	381.3	0:54:58	412.45	0:44:22	456.13	0:38:31
	100.0%	1.0	100.8%	1.5	104.5%	2.4	113.0%	3.0	125.0%	3.5
Valentin	730.04	3:46:22	731.23	2:09:31	757.39	1:17:10	780.51	0:59:52	821.52	0:50:11
	100.0%	1.0	100.2%	1.7	103.7%	2.9	106.9%	3.8	112.5%	4.5
Trips	640.05	3:10:28	643.04	1:41:00	667.65	1:03:45	677.33	0:50:31	703.9	0:43:36
	100.0%	1.0	100.5%	1.9	104.3%	3.0	105.8%	3.8	110.0%	4.4
Andre	479.84	3:22:19	476.28	2:04:17	491.3	1:16:23	523.28	0:56:58	581.53	0:53:48
	100.0%	1.0	99.3%	1.6	102.4%	2.6	109.1%	3.6	121.2%	3.8
Ludwig	636.58	2:02:37	630.02	1:28:03	646.86	0:59:13	696.3	0:49:55	782.27	0:45:16
	100.0%	1.0	99.0%	1.4	101.6%	2.1	109.4%	2.5	122.9%	2.7
Leyla	730.73	3:56:32	732.26	2:30:08	767.33	1:31:15	830.19	1:19:07	915.49	1:08:30
	100.0%	1.0	100.2%	1.6	105.0%	2.6	113.6%	3.0	125.3%	3.5
Erik	590.23	4:32:18	594.02	2:43:10	616.05	1:37:17	656.88	1:13:38	736.8	1:05:38
	100.0%	1.0	100.6%	1.7	104.4%	2.8	111.3%	3.7	124.8%	4.1
	100.0%		99.3%		102.0%		106.1%		111.8%	
		30:07:53		18:08:25		11:00:31		08:40:40		07:33:56
		1.0		1.6		2.7		3.5		4.0

Table 8.20: Results of BonnPlace with and without *BestChoice* for different cluster ratios with target density of 70%. Runtimes are global placement wall clock runtimes, the legalized bounding box netlength is shown. The runs were performed on Intel XEON with 3.0 GHz and 64 GB and used up to 8 CPUs simultaneously.

BonnPlace random walk clustering Hitting time computation runtimes (16 targets)				
chip	num cells	Runtime hh:mm:ss		
		hitting time	global placement	ratio
Dagmar	50 534	0:00:00	0:00:21	0.00%
Elisa	67 723	0:00:00	0:00:23	0.00%
Lucius	77 679	0:00:01	0:00:31	3.23%
Felix	84 966	0:00:01	0:00:28	3.57%
Paula	129 026	0:00:02	0:00:54	3.70%
Rabe	175 646	0:00:03	0:01:02	4.84%
Julia	190 554	0:00:03	0:01:04	4.69%
Max	328 789	0:00:05	0:02:01	4.13%
Roger	456 700	0:00:13	0:03:20	6.50%
Ashraf	866 777	0:00:43	0:05:26	13.19%
Patrick	1 052 709	0:00:47	0:07:13	10.85%
Erhard	2 578 246	0:01:57	0:16:16	11.99%
Arijan	3 753 151	0:03:50	0:27:37	13.88%
Philipp	3 945 833	0:03:44	0:29:23	12.71%
Tomoku	5 296 120	0:05:25	0:43:36	12.42%
Trips	5 747 007	0:04:43	0:42:12	11.18%
Valentin	5 838 457	0:04:07	0:47:08	8.73%
Andre	6 794 323	0:07:09	0:57:20	12.47%
Ludwig	7 500 446	0:07:22	0:57:29	12.82%

Table 8.21: Hitting time computations (wall clock runtimes) in relation to the global placement runtime of the following run in default settings and target density of 70%. Clustering $\alpha = 10$ on Intel XEON with 3.3 GHz and 64 GB, using 8 CPUs

8.9 Summary and Outlook

The presented algorithm is a significant improvement of the old BonnPlace global placement. On both the real-world instances from industry and international benchmarks, we were able to improve our placement in terms of netlength and runtime. Several designs saw improvements of 10% and beyond in terms of netlength. At the same time, the new implementation, despite its generalized data structures, is extremely fast: more than 5 times faster than the old version of BonnPlace.
Another significant issue is placement with movebounds. Here, our modified recursive partitioning already produces reasonable placements. With the new flow-based approach, we can improve our results again by about 10% on average for both the inclusive and the exclusive hard movebounds.

BonnPlace also shows its efficiency in direct comparison to the industrial tool, where we are able to close the netlength gap on instances without movebounds, and accelerate our tool to be more than 5.5 times faster w.r.t the industrial placer. With movebounds, the results are unambiguously clear: BonnPlace's placements are more than 30% shorter on average and BonnPlace is 9.5 times faster with inclusive movebounds and almost 21 times faster with exclusive movebounds.

The international benchmarks show that BonnPlace is competitive in terms of netlength, but not the best tool if netlength is the only objective. On the other hand, when combining

	BonnPlace								
chip	no clustering		*BestChoice* $\alpha = 10$		RandomWalk $\alpha = 10$				
	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss	BBox(m)	hh:mm:ss			
			ratio vs no cluster		ratio vs no cluster		ratio vs *BestChoice*		achieved α
Dagmar	0.92	00:00:48	0.91	0:00:22	0.9	0:00:39			6.6
	100.0%	1.0	98.9%	2.2	97.7%	1.2	98.8%	0.56	
Elisa	3.45	0:01:01	3.34	0:00:24	3.03	0:00:40			
	100.0%	1.0	96.8%	2.5	87.9%	1.5	90.8%	0.60	
Lucius	3.97	0:01:24	4.22	0:00:33	4.07	0:00:56			
	100.0%	1.0	106.4%	2.5	102.7%	1.5	96.5%	0.59	
Felix	7.92	0:00:54	8.21	0:00:27	7.85	0:00:48			
	100.0%	1.0	103.6%	2.0	99.1%	1.1	95.6%	0.56	
Paula	3.77	0:02:03	3.87	0:00:49	3.86	0:01:28			
	100.0%	1.0	102.8%	2.5	102.4%	1.4	99.6%	0.56	
Rabe	14.47	0:03:14	14.34	0:01:02	13.78	0:01:45			
	100.0%	1.0	99.1%	3.1	95.2%	1.8	96.1%	0.59	
Julia	13.36	0:03:56	13.28	0:01:12	12.1	0:01:58			
	100.0%	1.0	99.4%	3.3	90.6%	2.0	91.1%	0.61	
Max	22.64	0:06:16	21.82	0:02:13	20.13	0:03:37			
	100.0%	1.0	96.4%	2.8	88.9%	1.7	92.3%	0.61	
Roger	31.76	0:10:40	32.33	0:03:09	28.66	0:04:50			
	100.0%	1.0	101.8%	3.4	90.2%	2.2	88.7%	0.65	
Ashraf	67.5	0:13:07	76.16	0:04:35	66.89	0:09:39			
	100.0%	1.0	112.8%	2.9	99.1%	1.4	87.8%	0.47	
Patrick	50.13	0:23:01	54.72	0:06:40	52.6	0:13:52			7.0
	100.0%	1.0	109.2%	3.5	104.9%	1.7	96.1%	0.48	
Erhard	438.93	1:19:43	489.88	0:18:20	445.68	0:28:37			
	100.0%	1.0	111.6%	4.3	101.5%	2.8	91.0%	0.64	
Arijan	520.85	2:15:49	565.58	0:34:59	516.7	0:51:04			
	100.0%	1.0	108.6%	3.9	99.2%	2.7	91.4%	0.69	
Philipp	382.22	2:23:38	398.81	0:32:43	385.08	0:52:48			8.0
	100.0%	1.0	104.3%	4.4	100.7%	2.7	96.6%	0.62	
Tomoku	364.96	2:14:34	412.45	0:44:22	397.93	1:18:16			7.8
	100.0%	1.0	113.0%	3.0	109.0%	1.7	96.5%	0.57	
Valentin	730.04	03:46:22	780.51	0:59:52	747.46	1:34:02			
	100.0%	1.0	106.9%	3.8	102.4%	2.4	95.8%	0.64	
Trips	640.05	03:10:28	677.33	0:50:31	634.57	1:21:10			
	100.0%	1.0	105.8%	3.8	99.1%	2.3	93.7%	0.62	
Andre	479.84	03:22:19	523.28	0:56:58	489.43	1:37:47			9.4
	100.0%	1.0	109.1%	3.6	102.0%	2.1	93.5%	0.58	
Ludwig	636.58	2:02:37	696.3	0:49:55	664.51	1:48:08			5.4
	100.0%	1.0	109.4%	2.5	104.4%	1.1	95.4%	0.46	
Average			105.4%		98.5%		93.4%		
Total		21:41:54		6:09:06		10:32:04			
				3.5		2.1			

Table 8.22: Different clustering algorithms in BonnPlace compared to the run without clustering. The legalized netlength and the global placement wall clock runtimes (incl. clustering) are shown. The clustering stopped in several cases before the clustering ratio was achieved, as there were neighbors with rank$=-\infty$ only. The runs were made on Intel XEON with 3.0 GHz and 64 GB in standard settings.

netlength with density and runtime, BonnPlace indeed shows the best results on ISPD 2006 benchmarks, even with cautious clusterings.

As clustering method, we applied the *BestChoice* algorithm in most cases, which is shown to be inferior to our new, even preliminary, random walk based clustering scheme. The new random walk based clustering outperforms *BestChoice* by 6.6% on average, but in several cases the differences were more than 10%. On large instances, where clustering is desirable for performance reasons, the information from the small number of different hitting time computations seems to be insufficient. We believe that improving clustering can again increase the quality and decrease the runtime of BonnPlace.

The presented results are based on the replacement of the global placement by the new routines. For legalization, we used the algorithm by Brenner and Vygen [2004]. In the old version of BonnPlace, legalization runtime did not play a crucial role, but by the acceleration of global placement the relative runtime of legalization comes to the fore, especially in combination with movebounds (cf. Tables 8.12 and 8.13). In several runs, legalization takes more than 50% of the entire placement run. The total placement runtime of our tool can again be reduced by speeding up the legalization, which could for example use multithreaded algorithms in a similar framework as presented in this thesis.

Bibliography

Ababei, C., Selvakkumaran, N., Bazargan, K., and Karypis, G. [2002] Multi-objective circuit partitioning for cutsize and path-based delay minimization. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (2002), 181–185.

Adya, S.N. and Markov, I.L. [2002] Consistent placement of macro-blocks using floorplanning and standard-cell placement. In Proceedings of the International Symposium on Physical Design (2002), 12–17.

Agnihotri, A.R., and Madden, P.H. [2007]: Fast Analytic Placement using Minimum Cost Flow. Proceedings of the Asia and South Pacific Design Automation Conference (2007), 128–134.

Agnihotri, A.R., Ono, S., and Madden, P.H. [2005]: Recursive bisection placement: feng shui 5.0 implementation details. In Proceedings of the International Symposium on Physical Design (2005), 230–232.

Ahuja, R.K, Batra, J.L., and Gupta S.K. [1984]: A parametric algorithm for convex cost network flow and related problems. European Journal of Operational Research 16 (2), (1984), 222–235.

Ahuja, R.K., Orlin, J.B. and Stein, C. [1994]: Improved Algorithms for Bipartite Network Flow. SIAM Journal on Computing 23 (5) (1994), 906–933.

Alpert, C.J., Chan, T.F., Huang, D.J., Kahng, A.B., Markov, I.L., Mulet, P., and Yan, K. [1997]: Faster minimization of linear wirelength for global placement. In Proceedings of the International Symposium on Physical Design (1997), 4–11

Alpert, C.J., Huang, J.-H. and Kahng, A.B. [1997]: Multilevel Circuit Partitioning In Proceedings of the 34th Annual Design Automation Conference (1997). 530–533.

Alpert, C., Kahng, A.B., Nam, G.-J., Reda, S., and Villarrubia, P. [2005]: A semi-persistent clustering technique for VLSI circuit placement. Proceedings of the ACM International Symposium on Physical Design (2005), 200–207.

Alpert, C.J., Mehta, D.P., and Sapatnekar, S.S. (Eds.) [2009]: Handbook of Algorithms for VLSI Physical Design Automation. Taylor and Francis, Boca Raton (2009)

Barnett, S. [1990]: Matrices: methods and applications. Oxford University Press (1990).

Bednar, T.R., Buffet, P.H., Darden, R.J., Gould, S.W. and Zuchowski, P.S [2002] Issues and Strategies for the physical design of system-on-chip ASICs. IBM Journal of Research and Developement 46 (6) (2002), 661–673.

Bock, A. [2010], Postoptimierung durch Umplatzierung und Gate Sizing. Diploma thesis, University of Bonn (2010).

Brenner, U. [2000], Plazierung im VLSI-Design. Diploma thesis, University of Bonn (2000).

Brenner, U. [2005], Theory and Practice of VLSI Placement. Ph.D. thesis, University of Bonn (2005).

Brenner, U., and Rohe, A. [2002]: An effective congestion driven placement framework. Proceedings of the ACM International Symposium on Physical Design (2002), 6–11.

Brenner, U. and Struzyna, M. [2005]: Faster and better global placement by a new transportation algorithm. In Proceedings of the 42nd Annual Design Automation Conference (2005). 591–596.

Brenner, U., Struzyna, M., and Vygen, J. [2008]: BonnPlace: Placement of leading-edge chips by advanced combinatorial algorithms. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 27 (2008), 1607–1620.

Brenner, U., and Vygen, J. [2001]: Worst-case ratios of networks in the rectilinear plane. Networks 38 (2001), 126–139.

Brenner, U., and Vygen, J. [2004]: Legalizing a placement with minimum total movement. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 23 (2004), 1597–1613.

Brenner, U., and Vygen, J. [2009]: Analytical methods in VLSI placement. In: Alpert, Mehta and Sapatnekar [2009], pp. 327–346

Buttlar, D. and Farrell, J. and Nichols B. [1996]: PThreads Programming A POSIX Standard for Better Multiprocessing. O'Reilly, (1996).

Caldwell A., Kahng, A.B. and Markov I. Placement Formats, rev. 1.2 (1999) *http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/Placement/plFormats.html*

Chan, T.F., Cong, J., Shinnerl, J.R., Sze, K., and Xie, M. [2006]: mPL6: enhanced multilevel mixed-size placement. In Proceedings of the International Symposium on Physical Design (2006), 212–214.

Chandra, A.K., Raghavan, P. Ruzzo, W.L., Smolensky, R. and Tiwari. P. The Electrical Resistance Of A Graph Captures Its Commute And Cover Times. Proceedings of the 21st Annual ACM Symposium on Theory of Computing, (1989) 574–586.

Chang, Y.-W., Jiang Z.-W. and Chen, T.C. [2009]: Essential Issues in Analytical Placement Algorithms. Information Processing Society of Japan Transactions on System LSI Design Methodology 2(0) (2009), 145-166 2009.

Charikar, M., Makarychev, K., and Makarychev, Y. [2007]: A Divide and Conquer Algorithm for d-Dimensional Arrangement. Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (2007), 541–546.

Chen, T.C., Hsu, T.C., Jiang, Z.W., and Chang, Y.W. [2005]: NTUplace: a ratio partitioning based placement algorithm for large-scale mixed-size designs. In Proceedings of the International Symposium on Physical Design (2005), 236–238.

Chen, T.C., Jiang, Z.W., Hsu, T.C, Chen, H.C. and Chang, Y.W. [2008]: NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27 (7) (2008), 1228–1240.

Chen, M., Liu, J., and Tang, X. [2008]: Clustering via random walk hitting time on directed graphs. In Proceedings of the 23rd National Conference on Artificial intelligence 2 (2008), 616–621.

Cheng, C.K., Yao, S.Z., and Hu, T.C. [1991]: The orientation of modules based on graph decomposition. IEEE Transactions on Computers 40 (1991), 774–780.

Chong, P. and Szegedy, C. [2007]: A morphing approach to address placement stability. In Proceedings of the 2007 International Symposium on Physical Design (2007): 95–102

Cong, J., Hagen, L. and Kahng, A.B. [1991]: Random walks for circuit clustering. Proceedings of the IEEE Conference on ASIC, (1991), 14.2.1-14.2.4

Cong, J. and Lim, S.K. [2004]: Edge Separability-Based Circuit Clustering With Application to Multi-level Circuit Partitioning IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23 (3), (2004), 346–357.

Cong, J. and Luo, G. [2008]: Highly efficient gradient computation for density-constrained analytical placement methods. In Proceedings of the International Symposium on Physical Design (2008), 39–46.

Cuthill, E. and McKee, J. [1969]: Reducing the bandwidth of sparse symmetric matrices. In Proceedings of the ACM 24th National Conference (1969), 157–172.

Doyle, P.G. and Snell, J.L [1984]: Random Walks and electric Networks.
The Mathematical Association of America, Washington D.C. (1984).

Driesen, K. and Hölzle, U. [1996]: The direct cost of virtual function calls in C++. ACM Special Interest Group on Programming Languages Notes 31 (10) (1996), 306–323.

Eisenmann, H., and Johannes, F.M. [1998]: Generic global placement and floorplanning. Proceedings of the 35th IEEE/ACM Design Automation Conference (1998), 269–274.

Eisenmann, H. and Johannes, F.M. [1998]: Generic global placement and floorplanning. In Proceedings of the 35th Annual Design Automation Conference (1998), 269–274.

Elmore, W.C. [1984]: The transient response of damped linear networks with particular regard to wideband amplifiers. Journal of applied physics, 19 (1) (1948) 55–63.

Fiduccia, C.M., and Mattheyses, R.M. [1982]: A linear-time heuristic for improving network partitions. Proceedings of the 19th IEEE/ACM Design Automation Conference (1982), 175–181.

Firat, A., Chatterjee, S., and Yilmaz, M. [2007]: Genetic clustering of social networks using random walks. Computational Statistics & Data Analysis 51 (12), (2007), 6285–6294.

Fouss, F., Pirotte, A., Renders, J., and Saerens, M. [2007]: Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation. IEEE Transactions on Knowlege and Data Engineering 19 (3), (2007), 355–369.

Gangbo, W. and McCann, R.J. [1995]: Optimal maps in Monge's mass transport problem. Comptes Rendus de l'Académie des Sciences. Série I. Mathématique. 321 (12) (1995), 1653–1658.

Garey, M.R., Graham, R.L., and Johnson, D.S. [1977]: The complexity of computing Steiner minimal trees. SIAM Journal of Applied Mathematics 32 (1977), 835–859.

Garey, M.R. and Johnson, D.S. [1979], Computers and Intractability; A Guide to the Theory of NP-Completeness. (1979), W.H. Freeman & Co., New York, NY, USA.

Gusfield, D., Martel, C. and Fernández-Baca, C. [1987]: Fast algorithms for bipartite network flow. SIAM Journal on Computing 16 (2) (1987) 237–251.

Hagen, L. and Kahng, A.B. [1992]: A new approach to effective circuit clustering. In Proceedings of the International Conference on Computer-Aided Design, (1992), 422–427.

Hall, K.M. [1970]: An r-Dimensional Quadratic Placement Algorithm, Management Science, 17 (3), Theory Series (1970), 219–229

Harel, D. and Koren, Y. [2001]: On Clustering Using Random Walks. In Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science 2245, (2001), 18–41.

Held, S. [2009]: Timing Closure in Chip Design. Ph.D. thesis, University of Bonn (2008).

Hoffmann, S., Lienhart, R. [2008]: Eine Einführung in die parallele Programmierung mit C/C++ Springer, 1st edition, (2008).

Hu, B. and Marek-Sadowska, M. [2003]: Fine granularity clustering for large scale placement problems. In Proceedings of the International Symposium on Physical Design (2003), 67–74.

Hu, J., Shin, Y., Dhanwada, N.R. and Marculescu, R. [2004], Architecting voltage islands in core-based system-on-a-chip designs. International Symposium on Low Power Electronics and Design (2004), 180–185.

Hu, B., Zeng, Y., and Marek-Sadowska, M. [2005]: mFAR: fixed-points-addition-based VLSI placement algorithm. In Proceedings of the International Symposium on Physical Design (2005), 239–241

Intel Thread Building Blocks 3.0 [2010]:
available online
http://software.intel.com/sites/products/collateral/hpc/tbb/Intel_TBB3_InDepth.pdf

Kantorovitch, C.V. [1960]: Mathematical Methods of Organizing and Planning Production. Management Science 6 (4), (1960), 363–422.

Kahng, A.B and Wang, Q. [2006]: A faster implementation of APlace. Proceedings of the ACM International Symposium on Physical Design (2006), 218–220.

Karp, R.M. [1972]: Reducibility among combinatorial problems. In: Complexity of Computer Computations (R.E. Miller, J.W. Thatcher, eds.), Plenum Press, New York 1972, 85–103.

Karypis, G., Aggarwal, R., Kumar, V., and Shekhar, S. [1999]: Multilevel hypergraph partitioning: applications in VLSI domain. IEEE Transactions on VLSI Systems 7(1), (1999), 69–79.

Karypis, G. Kumar, V. [1999]: A fast and high quality multilevel scheme for partitioning irregular graphs SIAM Journal on Scientific Computing 20 (1), (1999) 359

Karypis, G. Kumar, V. [1999]: Multilevel k-way Hypergraph Partitioning VLSI Design, Vol. 11, No. 3, pp. 285 - 300, 2000

Khatkhate, A., Li, C., Agnihotri, A.R., Yildiz, M.C., Ono, S., Koh, C., and Madden, P.H. [2004]: Recursive bisection based mixed block placement. In Proceedings of the International Symposium on Physical Design (2004), 84–89.

Klein, D.J. and Randic, M. [1993]: Resistance Distance, Journal of Mathematical Chemistry 12, (1993) 81–95.

Kleinberg. J. [2002]: An impossibility theorem for clustering Proceedings of the 2002 Conference on Advances in Neural Information Processing Systems 15, (2002) 463–470.

Kleinhans, J.M., Sigl, G., Johannes, F.M., and Antreich, K.J. [1991]: GORDIAN: VLSI placement by quadratic programming and slicing optimization, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10 (1991), 356–365.

Korte, B., and Vygen, J. [2008]: Combinatorial Optimization: Theory and Algorithms. Fourth edition. Springer, Berlin (2008).

Kucar, D. and Vannelli, A. [2006]: Using Eigenvectors to Partition Circuits, INFORMS Journal on Computing, 18(2), (2006), 197–208.

Lauff C. [2010]: Clusteringverfahren in der Platzierung im physikalischen Entwicklungsprozess von Logikchips Diploma thesis, University of Bonn (2010).

Li, J. and Behjat, L. [2006]: Net cluster: a net-reduction based clustering preprocessing algorithm. In Proceedings of the International Symposium on Physical Design (2006), 200–205.

Li, C. and Koh, C. [2007]: Recursive Function Smoothing of Half-Perimeter Wirelength for Analytical Placement. In Proceedings of the International Symposium on Quality Electronic Design (2007), 829–834.

Lovász, L. [1993]: Random walks on graphs: a survey, Combinatorics, Paul Erdős is Eighty, vol. 2, Bolyai Society of Mathematical Studies, 2 (1993), 1–46.

Lua, Y., Hong, X., Zhuoa, Q., Caia, Y. and Gub J. [2007]: An efficient quadratic placement based on search space traversing technology. Integration, the VLSI Journal 40 (3) (2007), 253–260.

Luo, T. and Pan, D.Z. [2008]: DPlace2.0: a stable and efficient analytical placement based on diffusion. Proceedings of the 2008 Asia and South Pacific Design Automation Conference (2008), 346–351.

Maßberg, J. [2009]: Facility Location and Clock Tree Synthesis. Ph.D. thesis, University of Bonn (2009).

METIS - Family of Multilevel Partitioning Algorithms [1998]: available online: *http://glaros.dtc.umn.edu/gkhome/views/metis/*

Monge, G [1781]: Sur la théorie des déblais et des remblais. Mémoires de l'académie de Paris, (1781).

Müller, D. [2006]: Optimizing Yield in Global Routing. Proceedings of the IEEE International Conference on Computer-Aided Design (2006), 480–486.

Müller, D. [2009]: Fast Ressource Sharing in VLSI Routing. Ph.D. thesis, University of Bonn (2009).

Muuss, K. [1996]: Xi Data Model, mauscript, University of Bonn (1996).

Nam, G.-J.,Reda, S. Alpert, C.J., Villarrubia, P.G. and Kahng, A.B. [2006] A fast hierarchical quadratic placement algorithm. IEEE Transactions on Computer-Aided Design of Circuits and Systems, 25(4) (2006) 678–691.

Nam, G.J. and Villarubia, P.G. [2009] Placement: Introduction/Problem Formulation. in Handbook of Algorithms for Physical Design Automation (2009), 277–287.

Nam, G.-J., Alpert, C.J., Villarubbia, P., Winter, B. and Yildiz, M. [2005]: The ISPD 2005 Placement Contest and Benchmark Suite. Proceedings of the ACM International Symposium on Physical Design (2005), 216–220. *http://www.sigda.org/ispd2005/contest.htm*

Nam, G.-J. [2006]: The ISPD 2006 Placement Contest and Benchmark Suite. Proceedings of the ACM International Symposium on Physical Design (2006), 167. *http://www.sigda.org/ispd2006/contest.html*

Onodera, H., Taniguchi, Y., and Tamaru K. [1991]: Branch-and-bound placement for building block layout. Proceedings of the 28th ACM/IEEE Design Automation Conference (1991), 433–439.

Orlin, J.B. [1993]: A faster strongly polynomial minimum cost flow algorithm. Operations Research 41 (1993), 338–350.

Osipov, V. and Sanders, P. [2010]: n-Level Graph Partitioning, available online *http://arxiv.org/abs/1004.4024*

Ossler, P.J. [2004], Placement Driven Synthesis Case Studies on Two Sets of Two Chips: Hierarchical and Flat. Proceedings of the International Symposium on Physical Design (2004), 190–196.

Ossenberg-Engels, D. [2010]: Timingoptimierung kleiner Fanin Cones durch Logikresynthetisierung im VLSI Design. Diploma thesis, University of Bonn (2010).

Queyranne, M. [1986]: Performance ratio of polynomial heuristics for triangle inequality quadratic assignment problems. Operations Research Letters 4 (1986), 231–234.

Rao, C.R. and Mitra C.K. [1972]: Generalized inverse of a matrix and its applications Proceedings of the Sixth Berkeley Symp. on Math. Statist. and Prob. 1 (1972), 601–620.

Rao, V.B. [1995]: Delay Analysis of the Distributed RC Line. Proceedings of the 32nd ACM/IEEE Design Automation Conference (1995), 370–375.

Ren, H., Pan, D.Z., Alpert, C.J., and Villarrubia, P. [2005]: Diffusion-based placement migration. In Proceedings of the 42nd Annual Design Automation Conference (2005), 515–520.

Ren, H., Pan, D.Z., Alpert, C.J., Villarrubia, P. and Nam G.-J.[2007]: Diffusion-Based Placement Migration With Application on Legalization IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26(12) (2007): 2158–2172

Roy, J.A., Papa, D.A., Adya, S.N., Chan, H.H., Ng, A.N., Lu, J.F., and Markov, I.L. [2005]: Capo: robust and scalable open-source min-cut floorplacer. In Proceedings of the International Symposium on Physical Design (2005). 224–226.

Salz, P. [2009]: personal communication. IBM (2009).

Samet, H. [1984]: The Quadtree and Related Hierarchical Data Structures. ACM Computing Surveys 16 (2), (1984),

Schneider, J. [2009]: Macro Placement in VLSI Design. Diploma thesis, University of Bonn (2009).

Sechen, C. and Sangiovanni-Vincentelli, A. [1986]: TimberWolf3.2: a new standard cell placement and global routing package. Proceedings of the 23rd ACM/IEEE Design Automation Conference (1986), 432–439.

Si2- Silicon Intgration Initiative [2009]: The Chip Hierarchical Design System Technical Data Standard. Silicon Intgration Initiative (2009), *http://www.si2.org* .

Sigl, G., Doll, K. and Johannes, F.M. [1991] Analytical placement: a linear or a quadratic objective function? Proceedings of the 28th ACM/IEEE Design Automation Conference (1991), 427–432.

Spielman, D.A. and Srivastava, N. [2008]: Graph sparsification by effective resistances. Proceedings of the 40th Annual ACM Symposium on Theory of Computing (2008). 563–568.

Spindler, P., Schlichtmann, U. and Johannes F.M. [2008]: Kraftwerk2 - A Fast Force-Directed Quadratic Placement Approach Using an Accurate Net Model. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(8) (2008): 1398–1411.

Stoer, J. and Bulirsch, R. [2002]: Introduction to Numerical Analysis 3rd Edition, Springer (2000).

Stroustrup, B. [2000]: The C++ Programming Language Third Special Edition, Addison-Wesley Professional (2000)

Struzyna, M. [2004]: Alanytisches Placement im VLSI Design. Diploma thesis, University of Bonn, (2004).

Tamir, A [1991]: Obnoxious facility location on graphs. SIAM Journal on Discrete Mathematics 4 (1991), 550–567.

TRIPS (The Tera-op, Reliable, Intelligently adaptive Processing System) [2006]: *http://userweb.cs.utexas.edu/users/cart/trips/index.html*

Tsay, R.-S., Kuh, Kuh E.S. and Hsu, C.P [1988]: PROUD: A Sea-Of-Gates Placement Algorithm. IEEE Design & Test 5 (6), (1988), 44–56.

Tsota, K., Koh, C.-K. and Balakrishnan, V. [2009]: A study of routability estimation and clustering in placement In Proceedings of the IEEE International Conference on Computer-Aided Design (2009), 363–366.

Villani, C. [2009]: Optimal Transport old and new. Springer (2009)

Viswanathan, N., and Chu, C.C-N. [2005]: FastPlace: Efficient Analytical Placement using Cell Shifting IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24 (5) (2005), 722-733.

Viswanathan, N.,Pan, M. and Chu, C.C-N. [2007]: FastPlace 3.0: A Fast Multilevel Quadratic Placement Algorithm with Placement Congestion Control. Proceedings of the Asia and South Pacific Design Automation Conference, (2007), 135–140

Viswanathan, N., Nam, G., Alpert, C. J., Villarrubia, P., Ren, H., and Chu, C. [2007]: RQL: global placement via relaxed quadratic spreading and linearization. In Proceedings of the 44th ACM/IEEE Design Automation Conference (2007), 453–458

Vorwerk, K., Kennings, A., and Vannelli, A. [2004]: Engineering details of a stable force-directed placer. In Proceedings of the IEEE International Conference on Computer-Aided Design (2004), 573–580.

Vygen, J. [1997]: Algorithms for large-scale flat placement. Proceedings of the 34th IEEE/ACM Design Automation Conference (1997), 746–751.

Vygen, J. [2005]: Geometric quadrisection in linear time, with application to VLSI placement. Discrete Optimization 2 (2005), 362–390.

Vygen, J. [2007]: New theoretical results on quadratic placement. Integration, the VLSI Journal 40 (2007), 305–314.

Vygen, J. [2008]: On the complexity of bin packing, scheduling, and placement. Report No. 08990, Research Institute for Discrete Mathematics, University of Bonn, (2008)

Wang, M., Yang, X., and Sarrafzadeh, M. [2000]. Dragon2000: standard-cell placement tool for large industry circuits. In Proceedings of the IEEE International Conference on Computer-Aided Design (2000), 260–263.

Warme, D.M., Winter, P., and Zachariasen, M. [2000]: Exact algorithms for plane Steiner tree problems: a computational study. In: Advances in Steiner Trees (D.Z. Du, J.M. Smith, J.H. Rubinstein, eds.), Kluwer Academic Publishers, Boston 2000, pp. 81–116.

Wong, D.F., Leong, H.W., Liu, C.L. [1988]: Simulated Annealing for VLSI Design, Springer (1988).

Xiang, H., Ren, H., Trevillyan, L., Reddy, L., Puri, R., and Cho, M. [2010]: Logical and physical restructuring of fan-in trees. In Proceedings of the 19th International Symposium on Physical Design (2010), 67–74

Xiong, J., Wong, Y.-C., Sarto, E. and He, L. [2006]: Constraint driven I/O planning and placement for chip-package co-design. Proceedings of the Asia and South Pacific Design Automation Conference, (2006), 207–212.

Xiu, Z. and Rutenbar, R.A. [2007]: Mixed-size placement with fixed macrocells using grid-warping. In Proceedings of the International Symposium on Physical Design (2007), 103–110.

Yan, J.Z., Chu, C. and Mak, W. [2010]: SafeChoice: a novel clustering algorithm for wirelength-driven placement. In Proceedings of the International Symposium on Physical Design (2010). 185–192.

Yao, B., Chen, H., Cheng, C., Chou, N., Liu, L., and Suaris, P. [2005]: Unified quadratic programming approach for mixed mode placement. In Proceedings of the International Symposium on Physical Design (2005), 193–199.

Yen, L., Vanvyve, D., Wouters, F., Fouss, F.,Verleysen, M. and Saerens, M. [2005] Clustering using a random walk-based distance measure. Proceedings of the 13th European Symposium on Artificial Neural Networks (2005) 317–324.

Summary

Placement is one of the key steps in chip design. The major contributions of this thesis deal with global placement, a common relaxation of the placement problem. Here, we look for rough positions of the circuits, minimizing weighted wirelength. Beyond this classical objective, several other placement aspects of particular interest are addressed in this thesis: explicit position constraints in placement (movebounds) to address timing and routability as well as heterogeneous placeable objects and connection types. We put a particular emphasis on the runtime of global placement as it significantly contributes to the overall turnaround time.

Based on the idea of subsequent quadratic netlength minimization and top-down partitioning as in BonnPlace (Brenner, Struzyna and Vygen [2008]), we present several new algorithms, generalized data structures, and a completely new implementation.

The generalized concept of movebound constraints on subsets of circuits — formalized in Chapter 4 — allows inclusive and exclusive movebounds. Both restrict the positions of associated circuits to their area. Exclusive movebounds additionally serve as blockages to the non-associated circuits. Movebound areas may overlap. Using a decomposition of the chip area into homogeneous parts with respect to movebounds, the regions, we show that in $\mathcal{O}(n + rm^2)$ time (with n circuits, m movebounds and r regions) one can decide whether a fractional partitioning with movebounds exists. We prove that an optimal (with respect to some modular cost function) fractional partitioning can be obtained in $\mathcal{O}\big(r^2 n\big(\log(n) + r\log(r)\big)\big)$ time. Making use of this result, we generalize the top-down partitioning scheme of BonnPlace's global placement and extend the legalization algorithm of Brenner and Vygen [2004] to handle movebounds efficiently. Interactions of movebounds and density constraints are addressed and a modified algorithm for partitioning with movebounds is presented, which significantly improves the partitioning performance in practice.

In Chapter 5 we present a novel flow-based partitioning algorithm for global placement, which combines a new MinCostFlow model for computing directions with extremely fast and highly parallelizable local realization steps. Unlike the iterative and recursive approaches in the previous top-down partitionings, our algorithm provides a global view of the problem. The size of the MinCostFlow instance does not depend on the number of circuits and, despite its global view, is only linear in the number of regions. Therefore, an

optimal solution can be computed in $\mathcal{O}\big(r^2\log^2(r)\big)$ time. This approach is hence applicable to huge designs, as even on these instances the flow computation can be done within a few seconds.
Our new flow-based partitioning can also address density targets much more accurately and lowers the risk of density violations.
Another significant advantage of this algorithm is the fact that it can be applied to *any* initial placement. We prove that flow-based partitioning guarantees a feasible (fractional) partitioning solution (if one exists), even with movebounds and starting from placements with high density violations. Using this new partitioning routine, we can extend the congestion-driven placement approach of Brenner and Rohe [2002] to optimize movement, density adjustment and circuit size inflation even in the presence of movebounds. As a feasible fractional partitioning is guaranteed at any stage, flow-based partitioning can be used for incremental placement, and also creates the opportunity of applying local, density unaware optimization steps (such as buffering) within global placement.
We propose to realize the optimal flow by a sequence of combined local quadratic netlength minimization and local partitioning steps. Our realization scheme can be performed extremely quickly and efficiently parallelized.

In Chapter 6 we generalize the placement data structures and their implementation. Our new global placement can handle circuits, clusters, and fragments of circuits as well as different connection and attraction types. We also show how a clustering hierarchy can efficiently be handled in global placement. The classical net models clique and star are generalized to clustered netlists. We discuss several other implementation aspects and present a new parallelization scheme. The proposed shared-memory parallelization framework allows us to efficiently use dozens of processors for several core routines of BonnPlace and guarantees repeatability. Despite the new generalized data structures, our new implementation makes the new version of BonnPlace twice as fast as the previous one, even without using the new algorithmic contributions.

In Chapter 7 we focus on clustering in global placement. We use a random walk approach to compute global connectivity information. We show how hitting times of random walks in hypergraphs can be obtained from solving a sparse linear equation system. We present a method to compute hitting times to several targets simultaneously in parallel by using a single shared and sparse matrix. They are then used in a bottom-up clustering scheme in order to avoid wrong clustering decisions. Our approach, though still in a preliminary state, significantly outperforms the popular *BestChoice* algorithm (Nam et al. [2006]). It leads to much shorter netlengths (by more than 10% in several cases and 6.6% on average) in a reasonable runtime. Even clusterings reducing the number of circuits by a factor of 10 lead to better netlengths than placing unclustered netlists.

Chapter 8 concludes this work by providing several experimental results on a large testbed of real-world chips and benchmarks. Comparing our new global placement to the old version of BonnPlace, we observe significant improvements in terms of netlength. Our new

BonnPlace produces 8.4% shorter netlength on average. In several cases, we are able to improve the netlength by more than 10%, in one case even by more than 28%. These improvements are mostly due to the new flow-based partitioning. The new BonnPlace is more than 5.4 times faster. In standard settings our new global placement takes less than one hour for every chip in our testbed, even for an ASIC with 9.3 million circuits. In the fast mode, comparable results to the old version of BonnPlace can be obtained more than 8.7 times faster.

We also compared our new BonnPlace to a state-of-the-art industrial placer. Without movebounds, our tool produces similar results but is more than 5.5 times faster. On instances with movebounds, our placements have more than 32% shorter netlength. Moreover, our running time is better by a factor of more than 9 with inclusive and almost 21 with exclusive movebounds.

Our tool also shows competitive results in terms of netlength on recent benchmarks. On the latest ISPD 2006 benchmarks (Nam et al. [2006]), the new version of BonnPlace currently produces the best results among all placement tools.