Theory and Practice of VLSI Placement

Dissertation

zur Erlangung des Doktorgrades

der Mathematisch-Naturwissenschaftlichen Fakultät

der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Ulrich Brenner

aus Siegburg

im September 2005

Diese Dissertation ist auf dem Hochschulschriftenserver der ULB Bonn
http://hss.ulb.uni-bonn.de/diss_online elektronisch publiziert.

Erscheinungsjahr: 2006

Tag der Prüfung: 31. 3. 2006

Erstgutachter: Prof. Dr. B. Korte
Zweitgutachter: Prof. Dr. J. Vygen

Contents

Chapter 1

Introduction

The design of VLSI (= **v**ery-**l**arge **s**cale **i**ntegrated) chips is one of the most important and inspiring applications of mathematical optimization. The design process opens a wide spectrum of tasks that can be modeled very accurately as optimization problems and can be solved by efficient algorithms. Moreover, the growing complexity of VLSI chips makes the help of such algorithms absolutely mandatory.

VLSI chips consist of a huge number of tiny modules, called *circuits*, which are arranged in a rectangular area. The circuits either compute Boolean functions or store bits, and they communicate with each other by sending signals via electrical connections (*wires*). A crucial step for the whole design process of a chip is the computation of the circuit positions (*placement*). When placing the circuits, we have to ensure that further design steps are possible, for example, the routing of the wires between the circuits under a number of technological constraints. In addition, the way we place the circuits has a great influence on the optimization of the most important objective functions in chip design, e.g. the minimization of cycle time, size, and power dissipation of the chip. Thus, placement has to take care of several different goals. Furthermore, any tool for placement that is intended for the optimization of recent industrial chips must be very fast, as state-of-the-art chips may consist of several millions of circuits. Hence, only linear or almost linear running times are acceptable.

In this work, we examine the problem of placing the circuits of a VLSI chip under theoretical as well as practical aspects. We divide the placement problem appropriately into subproblems and propose new algorithms to solve them. In the coarsest subdivision of the placement problem, we distinguish between *global placement* in which the circuits are spread out roughly over the chip area and *detailed placement* (or *legalization*) in which the circuits are moved to their final positions. Though the placement problem as a whole is absolutely intractable from a theoretical point of view, some of the subproblems can be solved provably optimal or almost optimal. In other cases, we propose algorithms whose quality is shown at least a posteriori by experiments. Together, the algorithms for the subproblems are combined in the placement tool BonnPlace. The tool is based on an earlier version proposed by Vygen [1996]. It has been developed at the Research Institute for Discrete Mathematics of the University of Bonn in cooperation with IBM Microelectronics and has been used by IBM for the design of many challenging logic chips.

This thesis is organized as follows:

In Chapter 2, we summarize briefly our notation, especially concerning graph theory, and recall some well-known facts on efficient data structures that are used in the rest of the work. In addition, we state the placement problem formally, discuss different objective functions and give an overview of the complexity results on placement. At the end of the chapter, we present the chip data sets used for the experiments in the following chapters.

Chapter 3 contains an overview of previous approaches to global placement. In particular, the former version of BonnPlace is described in detail as our algorithm is based on it.

In Chaper 4, we present one of the most important contributions of this work. We propose a new algorithm for the Transportation Problem, a classical optimization task. In the Transportation Problem, we are given a set A of warehouses and a set B of customers. Each warehouse stores a certain amount of a product (its supply), and each customer has a certain demand of the product. In addition, we know for each warehouse $a \in A$ and each customer $b \in B$ the costs $c(a, b)$ for transporting one unit of the product from a to b. We assume that the costs scale linearly, so transporting γ units from a to b costs $\gamma \cdot c(a, b)$ (for $\gamma \in \mathbb{R}_{\geq 0}$). The task is to find a cheapest shipment from the warehouses to the customers respecting all supply and demand values. If the number of warehouses is fixed, our algorithm solves the problem to optimality in time $O(n \log n)$ (where $n := |B|$) improving the fastest previously known algorithm that had a running time of $O(n \log^2 n)$ on such instances.

The algorithm for the Transportation Problem is the main new ingredient of our global placement approach that is described in Chapter 5. We demonstrate how the transportation algorithm can be used to spread the circuits over the chip area by assigning them to subregions of the chip. The formulation as a Transportation Problem allows to handle more subregions in a single step and to apply more realistic objective functions for this assignment compared to the previous BonnPlace version. We propose several ways to make use of this flexibility. We conclude Chapter 5 by presenting experiments that compare these new methods to earlier approaches.

After global placement, the circuits are distributed over the chip area but their placement is not yet legal; especially, there will be many overlaps. In Chapter 6, we present an algorithm that legalizes a given placement, i.e., it moves the circuits in such a way that all overlaps are removed and all other technological constraints are met. Since we assume that the input of the legalization step is an optimized placement, our goal is to change the placement as less as possible. Hence, we want to minimize the total movement of the circuits during legalization. The first part of our legalizer is based on a new minimum cost flow formulation, and we show that this formulation is best possible in a natural, well-defined way. By comparing the results to lower bounds computed by a relaxation of the legalization problem, we can demonstrate that the legalization results are only a few percent away from the optimum on all tested chips. It should be noted that legalization can be applied not only as a second step after global placement but also after all kinds of optimization steps performed after placement that create new overlaps between circuits.

Not only the variety of different objective functions that have to be optimized makes placement a challenging problem, but also the fact that a placer has to be flexible enough to

handle very different types of instances. It is obvious that an algorithm that performs well on large sets of circuits with similar sizes and shapes may fail if the task is to place only a few hundreds of circuits with arbitrary sizes and shapes. Though the global placer proposed in Chapter 5 can, in principle, handle circuits of any size, it is mainly designed for the placement of many circuits whose sizes differ not too much. Therefore, we present an algorithm especially for the placement of a small number of larger objects (*macro placement*) in Chapter 7. The algorithm divides the circuits to be placed into three groups according to their sizes and places each group with a different strategy. We show the effectiveness of our macro placer mainly by comparing its results to macro placements that were found manually by an experienced designer.

Sets of circuits that communicate with each other are connected by so-called *nets*, and the main optimization goal for placement in literature is the minimization of the sum the lengths of these nets. Short connections between the circuits are useful for all further design steps after placement and are mandatory for a short cycle time of the chip. However, as the nets have to be realized by wires that have to meet certain minimum-distance constraints (routing), very dense arrangements of the circuits may make routing impossible. To avoid such issues, we propose in Chapter 8 a method to detect routing-critical areas early in the placement process. As soon as a critical area is found, the placement density is reduced in this area (*congestion-driven placement*). Experiments show that this method allows to place the circuits with higher density in uncritical areas and therefore helps to improve the quality of result.

While most of the chapters contain experiments that analyze single features of our placer, we present in Chapter 9 experiments that examine our placement tool as a whole. We test BonnPlace with different experiments both on artificial benchmarks and on real-world chips showing that BonnPlace is arguably one of the most effective and efficient existing placement tools.

At this point, I would like to take the opportunity to express my gratitude to my supervisors Professor Bernhard Korte and Professor Jens Vygen. Without their ideas, help and guidance this thesis would not have been possible. I would like to thank Professor Dieter Rautenbach for carefully proof-reading this thesis. I am grateful to all colleagues of the VLSI project, especially to my collaborators in the placement team, Anna Pauli, Dieta Schülter, Markus Struzyna, and Katrin Weidenbach.

Chapter 2

Preliminaries

In this chapter, we will introduce some basic terms and conventions that will be used in this work and we will give an overview of the problems that occur in VLSI placement. For the general notation see also the notation index in the appendix.

2.1 Basic Definitions and Results

As many problems that will be discussed in this work are formulated as graph problems or minimum cost flow problems, we will shortly summarize the basic definitions on these topics that we will use. We will mostly follow the standard notation as it has been introduced, e.g., in the textbook by Korte and Vygen [2002].

An *undirected graph* G is a triple $(V(G), E(G), \phi_G)$ of two finite sets $V(G)$ and $E(G)$, and a mapping $\phi_G : E(G) \to \binom{V(G)}{2}$. A *directed graph* G is a triple $(V(G), E(G), \phi_G)$ of two finite sets $V(G)$ and $E(G)$, and a mapping $\phi_G : E(G) \to (V(G) \times V(G)) \setminus \{(v, v) \mid v \in V(G)\}$. Let $(V(G), E(G), \phi_G)$ be a (directed or undirected) graph. The elements of $V(G)$ are called *vertices* or *nodes*, and the elements of $E(G)$ are called *edges*. Two edges e and e' with $\phi_G(e) = \phi_G(e')$ are called *parallel*. G is called *simple* if ϕ_G is injective. For simple graphs, we often identify an edge $e \in E(G)$ with $\phi(e)$ and denote the graph G by the pair $(V(G), E(G))$.

Let G be a directed graph. A *(directed) path* of length k in G is a graph P with $V(P) = \{v_1, \ldots, v_{k+1}\} \subseteq V(G)$, $E(P) = \{e_1, \ldots, e_k\} \subseteq E(G)$, such that $v_i \neq v_j$ for $i \neq j$, $i, j \in \{1, \ldots, k+1\}$, and $\phi_G(e_i) = (v_i, v_{i+1})$ for $i \in \{1, \ldots, k\}$. The path P is called *v_1-v_{k+1}-path*. A *(directed) cycle* of length k in G is a graph C with $V(C) = \{v_1, \ldots, v_{k+1}\} \subseteq V(G)$, $E(C) = \{e_1, \ldots, e_k\} \subseteq E(G)$, such that $v_i \neq v_j$ for $i \neq j$, $i, j \in \{1, \ldots, k\}$, $\phi_G(e_i) = (v_i, v_{i+1})$ for $i \in \{1, \ldots, k\}$, and $v_1 = v_{k+1}$.

In an undirected graph G with $v \in V(G)$, $\delta_G(v)$ is the set of edges in $E(G)$ incident to v, so $\delta_G(v) := \{e \in E(G) \mid v \in \phi_G(e)\}$. The number $|\delta_G(v)|$ is called *degree of v in G*.

In a directed graph G with $v \in V(G)$, $\delta_G^+(v)$ ($\delta_G^-(v)$) denotes the set of edges in $E(G)$ leaving (entering) v, so $\delta_G^+(v) = \{e \in E(G) \mid \exists w \in V(G) : \phi_G(e) = (v, w)\}$, and $\delta_G^-(v) =$

$\{e \in E(G) \mid \exists w \in V(G) : \phi_G(e) = (w, v)\}$. The number $|\delta_G^+(v)|$ is called *out-degree of v in G*, and $|\delta_G^-(v)|$ is called *in-degree of v in G*.

A *flow network* is a quadruple (G, b, u, cost) where G is a directed graph, $b : V(G) \to \mathbb{R}$ is a mapping with $\sum_{v \in V(G)} b(v) = 0$, $u : E(G) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ defines *edge capacities*, and $\text{cost} : E(G) \to \mathbb{R}$ defines *edge costs*. The number $b(v)$ is called *supply* value of v if $b(v) > 0$, and it is called *demand* value if v if $b(v) < 0$. The edge costs are *conservative* if there is no directed cycle C in G with $\sum_{e \in E(C)} \text{cost}(e) < 0$. For a flow network (G, b, u, cost), a *flow* f is a mapping $f : E(G) \to \mathbb{R}_{\geq 0}$ such that $f(e) \leq u(e)$ for all $e \in E(G)$, $\sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) = b(v)$ for all $v \in V(G)$. The *cost* of a flow f is defined as $\sum_{e \in E(G)} \text{cost}(e) \cdot f(e)$. $G_{f,u}$ is the *residual graph* of f, i.e., $V(G_{f,u}) := V(G)$ and $E(G_{f,u}) := \{e \in E(G) \mid f(e) < u(e)\} \dot\cup \{\overleftarrow{e} \mid e \in E(G) \text{ and } f(e) > 0\}$ where the edges $\overleftarrow{e}$ are additional edges with $\overleftarrow{e_1} \neq \overleftarrow{e_2}$ for $e_1 \neq e_2$, $e_1, e_2 \in E(G)$. The edge $\overleftarrow{e}$ is called *reverse edge* of e. For each edge $e \in E(G)$, we define $\phi_{G_{f,u}}(e) := \phi_G(e)$, and for each $\overleftarrow{e} \in E(G_{f,u}) \setminus E(G)$ with $\phi_G(e) = (v, w)$ we define $\phi_{G_{f,u}}(e) := (w, v)$. For a flow f on the edges of a directed graph G with edge capacities u, u_f is the residual capacity on the edges of the residual graph, i.e., it is $u(e) - f(e)$ on edges in $E(G)$ and $f(e)$ on the reverse edges. The function cost_f describes the residual edges costs, so we have $\text{cost}_f(e) := \text{cost}(e)$ on edges of $e \in E(G) \cap E(G_{f,u})$ and $\text{cost}_f(\overleftarrow{e}) := -\text{cost}(e)$ on reverse edges.

For terms and definitions on computational complexity, we refer to the book by Cormen, Leiserson, and Rivest [1990].

For efficient implementations of our algorithms, we make several times use of *heaps*, data structures that store elements labelled with certain *keys* (see Cormen, Leiserson, and Rivest [1990] for an overview). Heaps support at least the following operations:

- inserting an element,
- deleting an element,
- finding an element with smallest key,
- decreasing the key of an element.

Binomial heaps allow to perform all these functions in time $O(\log n)$ where n is the number of elements stored in the heap. With *Binary heaps*, finding an element with smallest key can be done in time $O(1)$, the other functions mentioned above take time $O(\log n)$. *Fibonacci heaps* (see Fredman and Tarjan [1984]) need amortized time of $O(\log n)$ for deleting an element, the other functions can be done in amortized time $O(1)$. Here "amortized time" means that a single call of the function may take a longer time but that, e.g., inserting k elements into an empty heap takes only time $O(k)$.

2.2 The Placement Problem

When we talk about VLSI chips, we always mean logic chips, as their placement is much more complicated than the placement of the regular structure of a pure memory chip.

In VLSI placement, we are given a set of rectangular objects (called *cells*, *circuits* or *modules*) that have to be placed within a rectangular chip area $r_0 = [0, W] \times [0, H]$. All circuits have to be placed in such a way that their edges are parallel to the x- or y-axis, so when we talk about rectangles in the plane, we always mean axis-parallel rectangles. The circuits either compute boolean functions (AND, OR etc.) or store bits. They communicate with each other via connections that are given by so-called *nets*. Formally, the instance of the PLACEMENT PROBLEM can be described as follows:

PLACEMENT PROBLEM

Instance:
- A rectangular *chip area* $r_0 = [0, W] \times [0, H]$;
- a grid $x_\delta\mathbb{N} \times y_\delta\mathbb{N}$;
- a set $C = C_{row} \dot\cup C_{macro}$ of rectangular *circuits* with widths $w : C \to x_\delta(\mathbb{N} \setminus \{0\})$ and heights $h : C \to y_\delta(\mathbb{N} \setminus \{0\})$ where $h(c) = y_\delta$ for all $c \in C_{row}$;
- a set B of rectangular *blockages*;
- a set P of *pins*;
- *pin offsets* $(x_{offset}, y_{offset}) : P \to \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}$;
- a function $ckt : P \to C \cup \{\emptyset\}$;
- a set $\mathcal{N} \subseteq 2^P$ of pairwise disjoint *nets* such that $P = \bigcup_{N \in \mathcal{N}} N$;
- a subset $C_f \subseteq C$ of preplaced circuits with coordinates $x_f : C_f \to x_\delta\mathbb{N}$ and $y_f : C_f \to y_\delta\mathbb{N}$;
- netweights $\omega : N \to \mathbb{R}_{\geq 0}$.

We ask for an extension of x_f and y_f to functions $x : C \to x_\delta\mathbb{N}$ and $y : C \to y_\delta\mathbb{N}$ (i.e., $x(c) = x_f(c)$ and $y(c) = y_f(c)$ for $c \in C_f$) such that the open rectangles in $\{]x(c), x(c) + w(c)[\times]y(c), y(c) + h(c)[\mid c \in C\}$ are pairwise disjoint subsets of r_0 and do not intersection any blockage. In other words, $(x(c), y(c))$ is the lower left corner of the rectangle of width $w(c)$ and heigh $h(c)$ that is covered by c, and the areas covered by different circuits may have at most a line in common.

The elements of C_{row} are called *row circuits* or *standard circuits*, their height y_δ is called *standard height*. The elements of C_{macro} are called *macros*. We may assume that C_{macro} contains much less elements than C_{row}. In real-world instances there may be several millions of row circuits but at most a few thousand macros. The y-coordinates of the circuits have to be integer multiples of the standard height y_δ, so they have to be placed in so-called *circuit rows*. This constraint is motivated by the fact that the circuits have to be connected to a power supply net. The row-based design makes the power supply much easier, since the power wires can be routed along the circuit rows. Without the regular row structure, it would hardly be possible to provide power supply to all standard circuits. The x-coordinate of the circuits must be an integer multiple of x_δ to facilitate the *routing* step (to be explained later) but it should be mentioned that x_δ is normally much smaller than y_δ and also smaller than most of the circuit widths.

To decide if a feasible solution (x, y) to an instance of the PLACEMENT PROBLEM exists, is an NP-complete problem as it contains the PARTITION PROBLEM (see Karp [1972]). However, since the total size of circuits is in all practical instances significantly smaller than the size of the chip (see the testsuite at the end of this chapter) and most of the

circuits are standard circuits, it is, in practice, quite easy to find a feasible solution: even simple greedy heuristics will work.

On the other hand, we are not just interested in *any* feasible solution, but we have to take into consideration that circuits communicate with each other and with the input- and output-connection points of the chip. The connections are described by the sets of nets and pins. Each pin either belongs to a circuit $ckt(p)$ (if $ckt(p) \in C$) or is an input- or output-pin (IO-pin) of the chip (if $ckt(p) = \emptyset$). If a pin p belongs to a circuit $ckt(p)$, then its offsets $(x_{offset}(p), y_{offset}(p))$ describe its relative position to the position of $ckt(p)$. So, if $ckt(p)$ is placed at $(x(ckt(p)), y(ckt(p)))$, then the position of p is $(x(p), y(p)) = (x(ckt(p)) + x_{offset}(p), y(ckt(p)) + y_{offset}(p))$. For an IO-Pin p, the position is directly given by the offsets, so $(x(p), y(p)) = (x_{offset}(p), y_{offset}(p))$. For a given placement $(x, y) : C_f \to x_\delta\mathbb{N} \times y_\delta\mathbb{N}$, the pins have fixed positions in the plane, and sometimes we will identify a pin p with its position $(x(p), y(p))$.

Each net N contains a set of pins (or *terminals*) that has to be connected by wires (*routing*). These wires may use several wiring planes. Wires on different planes can be connected by *vias*. Of course, the wires of different nets have to be disjoint, and a number of other technological constraints (especially defining lower bounds for distances between wires) have to be met. In practice, it is easier to handle such constraints if the router is restricted to use only vertical or horizontal wires in each plane. Typically, routers even have to embed the wires into a three-dimensional grid (*grid-based routing*), where the extension of the grid in z-direction is determined by the number of planes. Since we only consider designs with grid-based routing in this work, we restrict the feasible circuit coordinates to integer multiples of (x_δ, y_δ) to make sure that the pins are placed at routing grid coordinates. In grid-based routing, the task is to pack a set of vertex-disjoint *Steiner trees* (one for each net) into a grid graph. Since this graph can contain several billions of nodes, most routers work in two steps. First, the nets are connected in a very coarse version of the routing grid. The result of this *global routing* gives a quite accurate estimation of the routability of the chip; it also serves as a guideline for the *local router* which generates a detailed list of wires for each net.

Placement has to take care of the routability of the chip: obviously, a placement that cannot be routed is useless.

In addition to routability, *timing optimization* is a second goal placement has to consider. The nets connecting the pins are used to transfer signals. These signals start at input pins or at circuits that store bits (*latches*) and end at output pins or latches. The time that the signals need to pass through the paths on the chip determines the cycle time of the chip, i.e., it determines how fast the chip may work. In a simple model where all latches send and receive their signals at the same time (which is not necessary, see Held et al. [2003]), the minimum possible cycle time of the chip is just the maximum time a signal needs to pass one of these paths.

A common approach to take routability and timing optimization into account in placement is to minimize the amount of wire that is needed to connect the pins of each net. In this approach, one ignores disjointess and distance constraints of the wires and computes for each net (an estimation of) the length of a shortest L_1-Steiner tree. Then, the task is to minimize the sum of all these lengths. Since experiments show that (at least for

chips that are not too routing-critical) the sum of the Steiner tree lengths is a quite good estimation for the total amount of wire needed to connect all pins, minimizing the total netlength improves routability, at least from a global point of view. However, placements with short netlengths tend to be quite dense in certain regions, and a high density may lead to local routing problems. Therefore, we can compare netlengths only if we control the placement density. We can incorporate timing into the minimization of netlengths by defining netweights for nets on timing-critical paths. Then, our goal is to minimize the total weighted netlength.

By minimizing the total (weighted) netlength in placement, we also reduce the power consumption of the chip because power consumption depends on the total length of wire used in routing.

As placement is an early step in the physical design process, the quality of its result can hardly be estimated without performing the following design steps. On the other hand, for experimental and theoretical comparisons of different placement algorithms, we are interested in measurements that do not depend on the behaviour of a certain routing or timing optimization method. For this purpose, minimization of netlengths is the most important optimization goal that is used in almost all placement comparisons in literature.

2.3 Net Models

We have seen that the total weighted wire length is a reasonable objective function in placement. The question is how we can measure and optimize the netlength efficiently without actually routing the chip. There is a number of different estimations for the wiring length of a net. We assume that we are given a placement (x, y) of the circuits and therefore also positions $(x(p), y(p))$ for the pins. So, an estimation of the wirelength of a net is a function $s : \{N \subset \mathbb{R}^2 \mid \ N \text{ finite}\} \to \mathbb{R}$ that assigns to each net (identified with the set of its pin positions) a real number. Let $N \subset \mathbb{R}^2$ be a set with $n := |N| < \infty$. For a graph G with $V(G) \subset \mathbb{R}^2$, let L_1-length(G) be the sum of all L_1-distances between the endpoints of edges.

Steiner tree:

$$\mathrm{SMT}(N) := \min\{L_1\text{-length}(T) \mid T \text{ is a connected graph with } N \subseteq V(T) \subset \mathbb{R}^2\}.$$

A tree T with $N \subseteq V(G) \subset \mathbb{R}^2$ and L_1-length$(T) = \mathrm{SMT}(N)$ is called *L_1-minimum Steiner tree for N*. The minimum Steiner tree netlength is the most accurate net model. Since the distances between the planes of the routing grid are very small, one can ignore them without changing the netlength significantly. In that case $\mathrm{SMT}(N)$ is exactly the length of a shortest possible wiring for N (ignoring all other nets). It is NP-hard to compute a shortest Steiner tree (see Garey, Johnson [1977]), but the instances that occur in VLSI-design are small (most of the nets contain less than, say, 10 pins), and for these instance there are efficient algorithms that solve them very fast (see Warme, Winter, and Zachariasen [2000]).

Minimum spanning tree:

$$\mathrm{MST}(N) := \min\{L_1\text{-length}(T) \mid T \text{ is a connected graph with } N = V(T)\}.$$

A tree T with $N = V(T)$ and L_1-length(T) = MST(N) is called *L_1-minimum spanning tree for N*. The minimum spanning tree netlength is less accurate than the Steiner tree netlength but, e.g., with Kruskal's algorithm (see Kruskal [1956]), implemented with a Delauney triangulation, it can be computed in time $O(n \log n)$ (see textbooks, e.g., Korte and Vygen [2002]). Of course, we have $\mathrm{SMT}(N) \leq \mathrm{MST}(N)$. On the other hand, a famous result by Hwang [1976] yields the inequality $\mathrm{MST}(N) \leq \frac{3}{2}\mathrm{SMT}(N)$, so the minimum spanning tree netlength approximates the minimum Steiner tree netlength within a factor of $\frac{3}{2}$.

Clique:

$$\textsc{Clique}(N) := \frac{1}{n-1} \sum_{p,q \in N} \mathrm{dist}_1(p,q).$$

The clique netlength can easily be computed in time $O(n \log n)$. The sum of all distances is weighted by $\frac{1}{n-1}$, since otherwise single nets with many terminals would be too dominant. As the number of summands in the clique model is $\frac{n(n-1)}{2}$ and the number of edges in a minimum spanning tree is $n-1$, some authors use a factor of $\frac{2}{n}$ instead of $\frac{1}{n-1}$.

Star:

$$\textsc{Star}(N) := \min \left\{ \sum_{p \in N} \mathrm{dist}_1(p,q) \;\middle|\; q \in \mathbb{R}^2 \right\}.$$

The additional node q that is connected to all pins of the net is called *Steiner node*. The star netlength can be computed in time $O(n)$ (using a linear-time algorithm to compute a median, see Blum et al. [1973]).

Bounding box:

$$\begin{aligned} \mathrm{BB}(N) &:= \max\{x(p) \mid p \in N\} - \min\{x(p) \mid p \in N\} \\ &+ \max\{y(p) \mid p \in N\} - \min\{y(p) \mid p \in N\}. \end{aligned}$$

This is the simplest net model since the structure of the pin set N is not considered. One may move single pins, but, as long as one does not change the maximum or minimum coordinates, this will not change the bounding-box netlength. Obviously, it is less accurate than the Steiner tree or the spanning tree model, but at least for nets with only a few pins, BB(N) is a surprisingly good estimation of the actual wirelength. We have $\mathrm{BB}(N) \leq \mathrm{SMT}(N)$ for all nets, and Chung and Hwang [1979] show that for nets N with $|N| \leq 10$ (which are by far the majority of all nets), $\mathrm{SMT}(N) \leq 2\mathrm{BB}(N)$. Experiments on real-world instances (see Brenner [2000]), show that for most nets with up to 10 pins, the ratio $\mathrm{SMT}(N)/\mathrm{BB}(N)$ is much smaller than 2; even among the nets with exactly 10 pins, the ratio was smaller than 1.5 for more than 95 % of these instances.

The first four net models defined above replace the nets by graphs (where the net pins are contained in the vertex set), place the additional vertices optimally (if there are any) and use the total edge length of the graph as an estimation for the wirelength. The drawback of the first two models is that the topology of the graph is not known in advance. Therefore, it is quite difficult to use these models as objective functions in an algorithm. Among all net models that replace each net by a graph whose topology is defined before placement

(i.e., the graph does not depend on the pin positions), clique is the best, as the following theorem shows that has been proved in a paper by Brenner and Vygen [2001]:

Theorem 2.1 (Brenner and Vygen [2001]) *Let N be a finite set with $|N| > 1$. For an undirected graph G with $N \subseteq V(G)$, edge costs $c : E(G) \to \mathbb{R}_{>0}$, and a mapping $\sigma : N \to \mathbb{R}^2$, we define*

$$M_{(G,c)}(\sigma) := \min \left\{ \sum_{\{v,w\} \in E(G)} c(\{v,w\}) \cdot dist_1(\sigma'(v), \sigma'(w)) \;\middle|\; \sigma' : V(G) \to \mathbb{R}^2 \text{ with } \sigma'|_N = \sigma \right\},$$

and

$$r(G,c) := \sup \left\{ \frac{M_{(G,c)}(\sigma_1)}{M_{(G,c)}(\sigma_2)} \;\middle|\; \sigma_1 : N \to \mathbb{R}^2, \sigma_2 : N \to \mathbb{R}^2, \mathrm{SMT}(\{\sigma_1(v) \mid v \in N\}) = \mathrm{SMT}(\{\sigma_2(v) \mid v \in N\}) = 1 \right\}$$

Then, the function $r(G,c)$ is minimum if G is the clique on N and $c(e) = 1$ for all $e \in E(G)$. The minimum is $\frac{3}{2}$ if $|N| = 4$ and $\frac{\left\lceil \frac{|N|}{2} \right\rceil \left\lfloor \frac{|N|}{2} \right\rfloor}{|N|-1}$ for $|N| \neq 4$.

□

For a study on the worst-case ratios between the different net models, we refer to Brenner and Vygen [2001].

In literature on VLSI placement, bounding-box netlength is by far the most important measurement to compare the quality of different placements though it is even worse in reflecting the optimization goals for placement than, e.g., Steiner tree netlength. In this work, we will also mainly consider bounding-box netlength when we test our placer, but, as mentioned above, comparing any kind of netlength estimation computed with one of the net models described above only makes sense if also the density constraints of the corresponding placements are comparable. If we compare placements with different density constraints, we rather check the wirelength computed by a state-of-the-art (global) router because the router wirelength also reflects detours that are necessary due to dense placement regions.

2.4 Linear and Quadratic Netlength

In the net models that represent the net by a graph, we consider the sum of all L_1-lengths of the graph edges as an estimation of the wirelength. Choosing the L_1-distance is motivated by the fact that we have only horizontal or vertical wires in the routing planes. A different, widely used approach consists of minimizing *quadratic netlength*, i.e., replacing each net by a graph and using the sum of all squared Euclidean edge lengths as an objective function. So, the quadratic clique length of a net N is

$$\textsc{Clique}^2(N) := \frac{1}{n-1} \sum_{p,q \in N} \mathrm{dist}_2^2(p,q),$$

and the quadratic star length of a net N is

$$\textsc{Star}^2(N) := \min \left\{ \sum_{p \in N} \text{dist}_2^2(p, q) \;\middle|\; q \in \mathbb{R}^2 \right\}.$$

It is a well-known fact that for quadratic netlength the clique and the star net model are equivalent because it is easy the check that we have $\textsc{Star}^2(N) = \frac{n-1}{n}\textsc{Clique}^2(N)$. The star net model needs an additional node but produces a graph with only n edges while the clique net model needs $\frac{n(n-1)}{2}$ edges. Hence, often clique is used for nets with only a few terminals while star is applied to bigger nets.

Although quadratic netlength is not an accurate estimation for the routing wirelength, it has three important advantages:

1. Quadratic netlength reflects the fact that for timing optimization we want to avoid single large connections rather than to improve the average connection length.

2. Quadratic netlength is easy to optimize, if we ignore disjointness: To simplify the notation, we assume that all offsets of pins p with $c(p) \neq \emptyset$ are zero which does not change the complexity of the problem. Then, minimizing quadratic netlength can be regarded as finding positions $(x(v), y(v)) \in \mathbb{R}^2$ for the nodes v of an undirected graph G minimizing

$$\sum_{\{v,w\} \in E} \omega'(\{v, w\}) \cdot \Big((x(v) - x(w))^2 + (y(v) - y(w))^2 \Big).$$

   The elements of $V(G)$ correspond to the circuits, to IO-pins, or to additional points in the subgraphs that replace the nets. The mapping $\omega' : E(G) \to \mathbb{R}_{\geq 0}$ reflects the given netweights $\omega(N)$ and the weights due to the net model that is used. Let $V(G) = V_1 \dot\cup V_2$ such that V_1 contains the movable vertices and V_2 the preplaced vertices. Obviously, the x- and y-coordinates can be computed separately. For the x-coordinate, we have to minimize $x^t Ax - 2bx$ where $A = (a_{v,w})_{v,w \in V_1}$ and $b = (b_v)_{v \in V_1}$ with

$$a_{v,w} := \begin{cases} \sum\limits_{\substack{v' \in V(G): \\ \{v,v'\} \in E(G)}} \omega'(\{v, v'\}) & : \; v = w \\ -\omega'(\{v, w\}) & : \; v \neq w,\ \{v, w\} \in E \\ 0 & : \; \text{otherwise} \end{cases}$$

   and $b_v := \sum_{w \in V_2, \{v,w\} \in E(G)} \omega'(\{v, w\}) x(w)$. If each connected component of G contains an element of V_2, then matrix A is positive definite, and the quadratic program (QP) "min $x^t Ax - 2bx$" can be solved by computing a vector x with $Ax = b$ (see, e.g., Vygen [1996]).

   It should be noted that for linear netlength, it is also possible to compute an optimum placement in polynomial time (without meeting the disjointness constraints) since the problem can easily be formulated as a linear program. In fact, it can even be formulated as a minimum cost flow problem. But in practice, the quadratic netlength can be optimized faster.

3. Quadratic objective functions lead to more "stable" algorithms and better reproducable results. If each connected component of G (in the above notation) contains an element of V_2 (which is the case for real-world instances), the quadratic function is strictly convex, and hence there is only one optimal solution. For linear netlength, many optimal solutions can exist. In addition, the solution of the quadratic netlength minimization is not changed "too much" if we change the netlist "slightly" (see Vygen [2002a] where this statement is quantified) while even small changes of the instance can lead to a completely different placement if we minimize linear netlength. Since we have to rerun placement several times during the process of physical design, each time with slightly different netlists, this stability aspect is very important for the applicability of placement algorithms.

2.5 Complexity of the Placement Problem

We have already mentioned that it is NP-complete to decide if a given instance of the Placement Problem can be placed legally but that practical instances, that consist mostly of standard circuits and are not completely full, are easier to handle. Therefore, the following relaxation seems to be reasonable: We assume that *all* circuits have unit height and unit width and must be placed at integer coordinates. Moreover, we assume that all pin offsets are 0 and all nets have exactly two terminals. The task is to find a legal placement such that the weighted sum of the L_1-lengths of the nets is minimized (Unit-Square Placement Problem). Obviously, it is trivial to find a legal placement on these instances (if one exists). On the other hand, Queyranne [1986] has proved that for this relaxation no polynomial-time constant-factor approximation algorithm is possible if $P \neq NP$ even if we restrict ourselves to instances where the chip area has height 1.

An even more restricted version of the Unit-Square Placement Problem is the Linear Arrangement Problem where, again, the chip area has height 1, and no preplaced circuits or pins and no blockages are allowed. This problem is NP-hard even if all netweights are 1 (Garey, Johnson, and Stockmeyer [1976]), but there is a polynomial-time algorithm with an approximation factor of $O(\log(|C|))$ (Rao, Richa [1998]). For the special case of the Unit-Square Placement Problem where no preplaced objects and no blockages are allowed and all netweights are 1 but where the chip area is $[0, |C|] \times [0, |C|]$, Vempala [1998] proposed a randomized algorithm that computes with high probability an $O(\log^{3.5}(|C|))$-factor approximation. Vempala even claims an approximation factor of $O(\log^3(|C|))$ but there is a gap in his proof (see Brenner [2000]). If we restrict this special case further by demanding $|V| = \Theta(|C|^2)$, there is a polynomial-time approximation scheme (see Arora, Frieze, and Kaplan [1996]). However, these dense instances are not relevant for the practice as we have $|V| = \Theta(|C|)$ in all realistic instances.

Hansen[1989] considers another special case of the Placement Problem: Given an undirected graph G with edge weights, we ask for integer coordinates of the vertices of G such that the sum of the weighted edge lengths is minimized. Hansen proposes an approximation algorithm for this problem with an approximation factor of $O(\log^2(|V(G)|))$.

Even, Guha, and Schieber [2000] try find a routable placement with smallest-possible place-

ment area. Given a simple undirected graph G with maximum degree at most 4, the task is to find an $l \times l$-grid graph H, an injective mapping $\psi : V(G) \to V(H)$ and for each edge $\{v, w\}$ in G a path in H connecting $\psi(v)$ and $\psi(w)$. The paths have to be edge-disjoint and may not form *knock-knees* (two bends at an internal node), while the crossing of two paths is allowed. The goal is to minimize the area l^2. For this problem, the authors give an $O(\log^4(|V(G)|))$-factor approximation algorithm.

2.6 Global and Detailed Placement

Following a widely used strategy, the algorithm that we propose works in two steps: In the first step, called *global placement*, circuits are spread over the chip area optimizing the objective functions described above but without meeting the disjointness constraints and without placing the standard circuits at the grid coordinates. Only the non-standard circuits will have legal positions after this step and will be fixed at their position. In the second step, called *detailed placement* or *legalization*, our goal is to meet all constraints while changing the placement as little as possible. Note that some authors use "detailed placement" for a placement phase that locally tries to improve a given placement, while in this work, we use "detailed placement" and "legalization" in the same meaning.

2.7 The Testsuite

The algorithms presented in this work have been tested on a set of recent ASICs from IBM Microelectronics. Table 2.1 gives an overview of the chips we used for experiments. The instance sizes range from about 72 000 to 3.6 millions of circuits. For each chip, the table reports the number of circuits, nets and pins, the global density (i.e., the total size of all circuits divided by the total free area of the chip) and the standard density (i.e., the size of all standard circuits divided by the total free area of the chip that will not be covered by a macro). The numbers in column seven are the width and the height of the chip area (in mm). The last column contains the width of a smallest rectangle that can be built in the chip's technology.

Chip	#Circuits	#Nets	#Pins	Global Density	Standard Density	Size (mm)	Lithography
Jens	72 496	73 273	260 054	86.01 %	76.97 %	2.3 × 2.0	0.25 μm
Hans	72 940	73 822	252 053	30.87 %	26.64 %	4.9 × 4.9	0.25 μm
Hartmut	158 802	166 558	609 151	41.66 %	28.29 %	17.0 × 17.1	0.25 μm
Klaus	214 988	219 975	682 416	63.86 %	19.15 %	13.3 × 13.3	0.35 μm
Christian	278 083	283 358	865 841	44.87 %	22.93 %	9.9 × 9.9	0.18 μm
Heidrun	282 677	300 600	1 112 112	88.08 %	56.04 %	6.5 × 6.3	0.13 μm
Paul	399 830	387 980	1 342 937	91.04 %	73.17 %	4.9 × 4.9	0.09 μm
James	412 050	425 464	1 450 744	53.95 %	13.73 %	15.0 × 15.0	0.18 μm
Max	521 375	528 953	1 512 022	74.37 %	53.72 %	9.4 × 9.4	0.18 μm
Aidan	681 987	706 499	2 494 195	56.91 %	40.71 %	11.4 × 11.6	0.18 μm
Katrin	763 484	838 095	2 881 265	80.29 %	55.62 %	10.1 × 10.1	0.13 μm
Hanno	779 033	790 735	2 449 976	69.26 %	34.25 %	10.2 × 10.3	0.13 μm
Sven	825 737	841 554	3 264 316	77.08 %	37.40 %	15.6 × 15.7	0.18 μm
Dagmar	904 756	930 848	3 249 493	70.10 %	61.20 %	12.5 × 12.5	0.25 μm
Yvonne	915 086	973 324	3 660 175	84.24 %	50.97 %	11.1 × 11.2	0.13 μm
Dieta	923 472	934 276	3 786 401	44.40 %	31.34 %	12.6 × 12.6	0.18 μm
Alex	971 113	1 011 808	3 561 893	78.67 %	57.82 %	9.3 × 9.4	0.13 μm
Sandra	1 317 488	1 387 358	5 532 226	52.94 %	36.21 %	13.4 × 13.4	0.18 μm
Josef	1 349 390	1 378 303	4 514 036	59.21 %	35.14 %	15.6 × 15.7	0.18 μm
Kevin	1 497 709	1 555 652	5 036 256	87.32 %	71.19 %	10.7 × 10.7	0.13 μm
Reinhardt	1 513 386	1 528 361	5 274 635	74.33 %	56.74 %	13.4 × 13.4	0.18 μm
Nadine	1 654 756	1 688 719	6 419 832	71.07 %	40.01 %	18.2 × 18.2	0.18 μm
Hardy	2 056 728	2 074 182	6 343 151	64.66 %	50.84 %	14.2 × 14.2	0.18 μm
Wolf	2 396 642	2 410 881	7 490 008	51.42 %	27.08 %	16.5 × 16.6	0.13 μm
Ulrich	2 601 521	2 662 828	10 571 925	52.83 %	25.52 %	16.5 × 16.6	0.13 μm
Fermi	3 605 741	3 677 412	15 776 217	73.81 %	54.21 %	15.6 × 16.6	0.13 μm

Table 2.1: The chips used for experiments.

Chapter 3

Previous Global Placement Algorithms

This chapter gives an overview of previous approaches to placement. We focus on global placement algorithms but it should be noted that some of the methods not only compute global placements but also include legalization.

3.1 Simulated Annealing

Simulated annealing is a general scheme that can be applied to a wide variety of optimization problems. Starting with any feasible solution, simulated annealing algorithms apply iteratively local changes to the solution. The changing steps are chosen randomly and steps that make the solution worse are allowed, so it is possible to leave local optima. In early steps, bigger worsening changes are applied ("higher temperature"), while in later steps only small worsening changes are allowed ("lower temperature"). The slower the temperature is decreased in this process the better are the results. Simulated annealing approaches have been used in VLSI design in several ways (see Sechen [1988], and Wong, Leong, and Liu [1988]). A placement tool that is based mainly on simulated annealing is TimberWolf, one of the first automated placement tools at all (Sechen and Sangiovanni-Vincentelli [1986], Sechen and Lee [1987], Sun and Sechen [1995, 1997]). TimberWolf does not only apply to global placement as it computes legal placements. The TimberWolf tool package also contains a simulated annealing based solver for global routing.

The strength of simulated annealing is its flexibility. Any objective function can be used as long as it can be evaluated efficiently. So, not only netlength, but also, e.g., estimations of routability and timing can be combined in one optimization goal for placement. In addition, it is quite easy to implement. This may explain why it was used in the first automated VLSI tools. However, if one wants to get competetive results by simulated annealing, the temperature has to be lowered so slowly that running times get unacceptable even on medium-sized instances. Simulated annealing processes can be made to converge to an optimum solution, but only with extremely large running times. By parallelization (see,

for example, Sun and Sechen [1997]) one might slightly increase the instance sizes that can be handled by such approaches but the scaling of the running times is still too bad for recent real-world designs.

Although simulated annealing is too slow for a global optimization of a placement, it is still in use to solve subproblems or for local optimization. For example, Adya et al. [2004] use in some cases simulated annealing during global placement to compute locations for larger circuits. Wang, Yang, and Sarrafzadeh [2000b] apply low-temperature simulated annealing during a partitioning-based placement run to improve the placement locally.

3.2 Minimum-Cut Placers

Top-down recursive partitioning is used in many placement algorithms. The main idea consists of recursively dividing both the chip area and the set of circuits into subsets and to assign each circuit subset to a subarea of sufficient capacity. The step is repeated until the regions are small enough to run legalization.

The main question in all partitioning-based placement approaches is how the assignment of the circuits to the subregions is computed. An objective function that is widely used in literature is the minimization of the *cut-size* of the partition, i.e., the number of nets with pins in different subsets. Many placement tools follow this strategy or modifications of it, for example QUAD (Huang and Kahng [1997]), Capo (Caldwell, Kahng, and Markov [2000], Adya et al. [2004], and Adya and Markov [2005]), Dragon (Wang, Yang, and Sarrafzadeh [2000b], Sarrafzadeh, Wang, and Yang [2003]), Feng Shui (Yildiz and Madden [2001], Khatkate et al. [2004]), and NTUplace (Chen et al. [2005]). Since computing a balanced partition of the circuit set into, say, two subsets with minimum cut size is NP-hard (Bui and Jones [1992]), minimum-cut-based algorithms apply heuristics. Mainly the linear-time approach described by Fiduccia and Mattheyses [1982] is either applied directly or used as a subroutine in more sophisticated multilevel partitioning methods (see Alpert, Huang, and Kahng [1997] and Karypis et al. [1997]). Empirically, these multilevel partitioners produce significantly better cuts than the method of Fiduccia and Mattheyses but the improvements of the result are paid by larger running times. Nevertheless, minimum-cut placers are in general much faster than algorithms based on simulated annealing and can produce reasonable results. Groups of strongly connected circuits will not be split in the first partitioning steps and thus will probably be placed in the same area. On the other hand, it is obviously not necessary to keep these groups together in the early steps if one can guarantee that the circuits of each group will be placed in regions that are close enough to each other. It may even happen that excellent cuts in early partitioning steps make large cuts in later steps necessary. Under practical aspects, a main drawback of minimum-cut-based partitioning strategies is their instability: small changes in the netlist can lead to completely different placements. Since (as mentioned above) a designer typically has to repeat the placement step during the physical design process several times with slightly different inputs, such instable algorithms are hard to use without additional (artficial) constraints like movebounds for certain circuits. With movebounds, users prescribe areas (mostly rectangles or sets of rectangles) for groups of circuits in which these circuits have to be placed.

It should be noted that dividing the set of circuits into subsets with small cuts is also sometimes used to accelerate other placement methods. To this end, clusters of circuits instead of single circuits are placed in the early phase of the placement. Then, the locations of the clusters are used to compute a placement for the circuits within the clusters. Some of the placers described in the next section make use of such a clustering.

3.3 Pure Analytic Placers

Analytic placers start with a placement that minimizes netlength but ignores disjointness (and other constraints) completely. Then, the placement is modified in order to reduce the overlaps of the circuits. In a pure analytic approach this is done by changing the objective function to be optimized.

As it can be optimized efficiently (if disjointness is ignored), quadratic netlength is a commonly used objective function for analytic placers. The solution of the equation system $Ax = b$, as described in Section 2.4, would be an optimal placement (in terms of squared netlength) if circuits did not overlap. The idea of the *force-directed placement* approach by Eisenmann and Johannes [1998] (see also Eisenmann [1999]) is to start with a solution of $Ax = b$ and to work towards disjointness by changing iteratively the right-hand side b. To this end, they define repulsing forces between the circuits and compute for each circuit the sum of all other circuits' forces on it. There is also an attracting force of the chip area, and the forces are scaled in such a way that they would be zero in an exactly equally distributed placement. In the beginning, the vector entry $b(c)$ is the sum of the coordinates of the preplaced pins circuit c is connected to, and in each step of the algorithm, the forces at the current position of c are added to $b(c)$. This can be regarded as changing the positions of the preplaced vertices connected to c. After changing b to b', the equation system $Ax = b'$ is solved, new forces are computed according to the new circuit positions, and so on, until the circuit distribution over the chip area is good enough to start legalization. The whole algorithm can be seen as a discrete simulation of a continuous physical repulsion process.

A great advantage of the force-directed placement approach is its stability: if b is changed in sufficiently small steps, the movement of the circuits is almost continuous. The algorithm does not need any artificial cutlines on the chip area that would enforce discrete (and therefore instable) decisions. The approach follows most consequently the strategy of reducing the overlaps of the circuits while changing the QP solution as little as possible.

On the other hand, the relative positions of circuits can hardly be changed during a placement run based on this approach. Therefore, circuits tend to stay in their relative positions they got after a few iterations even if swapping some of them could reduce the netlength significantly. Additional swapping steps are necessary to compute placements with short netlength. Vorwerk, Kennings, and Vannelli [2004] propose an optimization method that they call "BoxPlace". For each circuit, they compute the area where the circuit can be placed such that the bounding-box length of the nets connected to it is minimized. Then, they choose one position in this area that creates the least new overlap as new location for the circuit. The new placement of the circuits is accepted if the total overlap has not increased too much. BoxPlace is called (with different orderings of the circuits) each time

after some iterations of the force-directed placement when the overlap has been reduced by 3 %. A different local optimization step for force-directed placement that greedily moves single circuits is described by Viswanathan and Chu [2004].

A drawback of force-directed placement is its running time. For n circuits, a sufficiently precise approximation of the forces can be computed in time $O(n \log n)$ (see Eisenmann [1999]), but there is a big number of iterations necessary, and in each iteration a QP has to be solved. The experiments reported by Eisenmann [1999] indicate a running time of about $O(n^{1.5})$ for complete placement runs which can be too much for large instances. Viswanathen and Chu [2004] (FastPlace) propose an idea for a fast force-directed placement. Instead of computing two-dimensional force vectors, they iteratively spread circuits only in horizontal or vertical direction. The authors divide the chip area with a regular grid into rectangular bins. To compute the repulsing forces in x-direction, they consider sets of bins that form a row. In each bin of a row, they compute the actual density. Then, if there are bins with overloads, they compute how the circuits in the row can be moved horizontally such that the overload is reduced. Correspondingly, vertical forces are computed using columns of bins. Finally, circuits are moved in the direction of the forces by artificial connections to fixed pins at appropriate positions. Although the combination of the horizontal and vertical spreading forces is only a rough approximation of real physical repulsing forces, the algorithm (combined with local optimization step, see above) seems to produce very efficiently quite acceptable results at least on dense benchmarks that do not contain macros.

The most important disadvantage of the force-directed placement approach is the fact that it computes a (nearly) equal distribution of the circuits on the chip area. If the chip density is high, i.e., if the total size of circuits is almost as big as the free chip area, then such a placement is desirable, but for designs with lower density such distributions will cause weak results. One could force the placer to use lower densities in some regions (and therefore higher densities in other regions) by defining areas with smaller capacity, but then it would be necessary to decide in advance which parts of the chip one wants to use for the placement of the circuits. Possibly, the problem can be fixed by introducing dummy circuits. In combination with effective swapping methods such dummy circuits can be placed together with the real logic and block the parts of the area.

Recently, Kahng and Wang [2004] (APlace) and Chan, Cong, and Sze [2005] (mPL) proposed an analytic placers that do not try to optimize quadratic netlength but use a differentiable approximation to the bounding-box netlength. For a net N for pins positions $(x(p), y(p))$ for each $p \in N$ and a parameter α, they define

$$WL_\alpha(N) := \alpha \left(\ln \left(\sum_{p \in N} e^{\frac{x(p)}{\alpha}} \right) + \ln \left(\sum_{p \in N} e^{\frac{-x(p)}{\alpha}} \right) + \ln \left(\sum_{p \in N} e^{\frac{y(p)}{\alpha}} \right) + \ln \left(\sum_{p \in N} e^{\frac{-y(p)}{\alpha}} \right) \right).$$

It is easy to see that $WL_\alpha(N) \to BB(N)$ for $\alpha \to 0$. Kahng and Wang [2004] combine this function with a smooth potential function that penalizes placement overloads to a differentiable objective function that they optimize by a conjugate gradient method. Chan, Cong, and Sze [2005] compute a discrete approximation of repulsing forces in order to reduce overlaps. Both placers apply a multilvel minimum-cut clustering of the circuits to improve the running time.

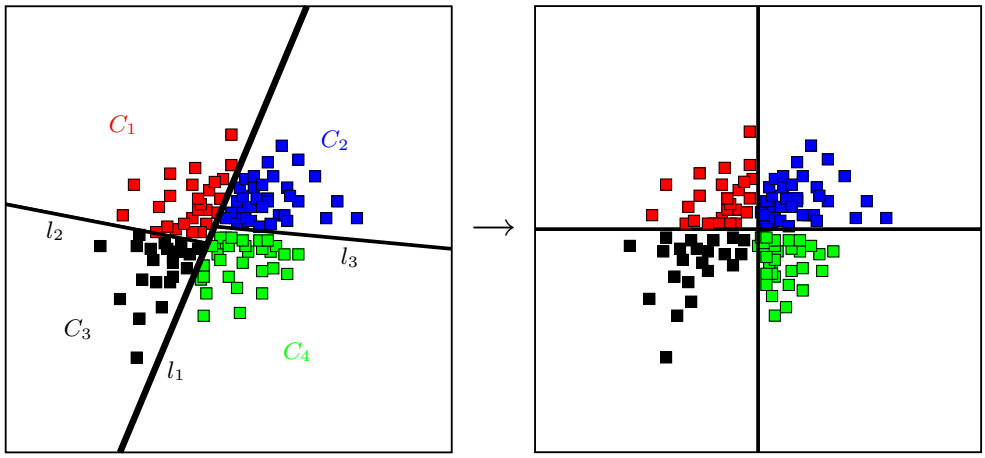


Figure 3.1: A "grid-warping" partitioning step.

3.4 Analytic Placer with Top-Down Partitioning

Several placers combine analytic placement ideas and recursive top-down partitioning. They start with a placement minimizing (quadratic) netlength and then partition the circuits according to their locations.

One of the first combinations of analytic placement and recursive partitioning was the tool PROUD (Tsay, Kuh, and Hsu [1988]). Starting with a QP solution, the circuits are partitioned by a vertical or horizontal cutline at the median of the circuits' coordinates. However, in their experiments the authors observe better results for a "simple heuristic" that divides the circuits and the area into four parts using a vertical and a horizontal cutline in one single step.

Kleinhans et al. [1991] (GORDIAN) partition the circuits by a vertical or horizontal cutline, too. Circuits that are placed close to the cutline are allowed to leave their subset in order to minimize the number of nets that are cut, so a min-cut heuristic is applied to these circuits. After a partitioning step, additional constraints on the center of gravities of the circuit sets move circuits towards the region they are assigned to. Local optimization steps that are able to correct wrong decisions in partitioning (similar to the routine REPARTITIONING in BONNPLACE, see the next section) help to improve the quality of result.

BONNPLACE (Vygen [1997]) will be explained in greater detail in the next section as our placer is based on this algorithm.

Xiu et al. [2004] (see also Xiu [2005]) also start with a placement that minimizes quadratic netlength but partition the set of circuits by cutlines that do not have to be horizontal or vertical. Assume, for example, that we want to partition the set of circuits (and the chip area) into four parts. The chip area is partitioned by a horizontal and a vertical cutline running through the whole chip area, thus forming four rectangular subregions. In order to partition the set of circuits, the authors compute a cutline l_1 connecting the upper borderline of the chip area to the lower borderline and two cutlines l_2 and l_3 connecting the left (right) borderline of the chip area to l_1 (see Figure 3.1). These three cutlines

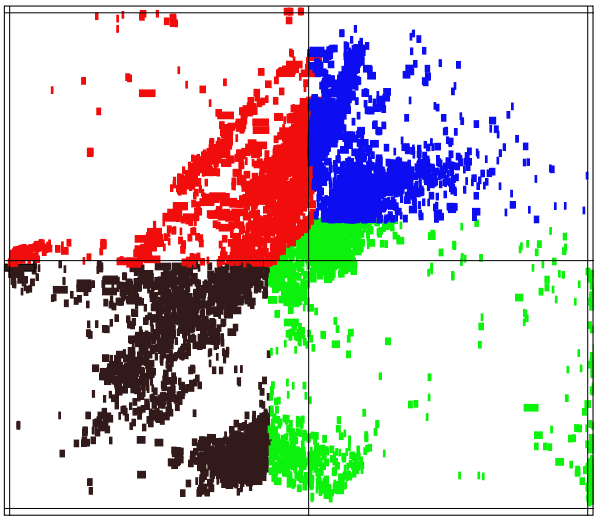


Figure 3.2: A set of circuits partitioned by an American Map.

partition the set of circuits into four subsets C_1, C_2, C_3, and C_4, and each subset is assigned in a canonical way to a subregion. The cutlines used to partition the set of circuits are chosen such that some capacity constraints are met for the subregions and such that routing congestion and netlength are minimized when the circuits are moved to their regions. As it seems to be hard to find optimal cutlines with these optimization goals, the authors apply a kind of local search to compute the cutlines. They argue that this is good enough as there is only a small number of variables (two variables for each cutline). As the algorithm does not only use vertical and horizontal cutlines for the partitioning of the circuits and distorts the placement of the circuits in a partitioning step, the authors call it "grid-warping" partitioning. Using some ideas taken from Vygen [1997] (terminal propagation, repartitioning; to be explained the next section), the partitioning step is repeated recursively until the regions are small enough.

3.5 BonnPlace

The main idea of the old BonnPlace, the approach described by Vygen [1997], is (similar to the force-directed placement approach) based on the observation that a placement computed by solving the QP $Ax = b$ (as stated above) would be optimal if it was legal. Consequently, the algorithm tries to partition the circuits in such a way that the placement is changed as little as possible, i.e., that the total movement that is needed to move the circuits to the subregions they are assigned to is minimized. Formally, the following problem is considered: Given a set of circuits placed in the plane and capacities for the four quadrants of the plane, the task is to move each circuit into one of the four quadrants of the plane such that the total size of circuits in each quadrant does not exceed its capacity. The objective function to be minimized is the total L_1-movement.

Using geometric structures called *American Maps*, at least a fractional solution (up to three circuits are distributed to different quadrants; all other circuits are assigned to only

one quadrant) can be computed in linear time (see Vygen [2000]). The structure of an American Map can be seen in Figure 3.2 that shows a part of a chip. The circuits to be placed are colored according to the region they are assigned to, i.e., the red circuits are assigned to the upper left region, the dark blue circuits to the upper right region, the dark grey ones to the lower left area, and the green ones to the lower right rectangle. The diagonal and the vertical and horizontal lines between the circuit sets form what is called an American Map. The partitioning strategy based on American Maps can be considered as a two-dimensional generalization of the one-dimensional median computation proposed by Tsay, Kuh, and Hsu [1988].

The partitioning of the regions is done by vertical and horizontal cutlines that cross the whole chip area. Between each pair of cutlines whose distance is still big enough a new cutline is inserted in each partitioning step, so, generally, each region is partitioned into four subregions. The horizontal cutlines are chosen such that they are placed at the border of two neighbouring circuit rows. Schematically, BONNPLACE can be described as follows:

BONNPLACE

Input: An instance of the placement problem.

Output: A placement.

① INITIALIZATION:
window_set := $\{r_0\}$;
$C(r_0) := C$;

② Solve a QP to minimize quadratic net-length;

③ WHILE (window size is big enough)
{
  FOR (each window r in window_set)
  {
    Solve constrained QP;
    PARTITIONING$(r, C(r))$;
    Solve a QP (with terminal propagation at the subwindow borders);
  }
  REPARTITIONING;
}

④ LEGALIZATION;

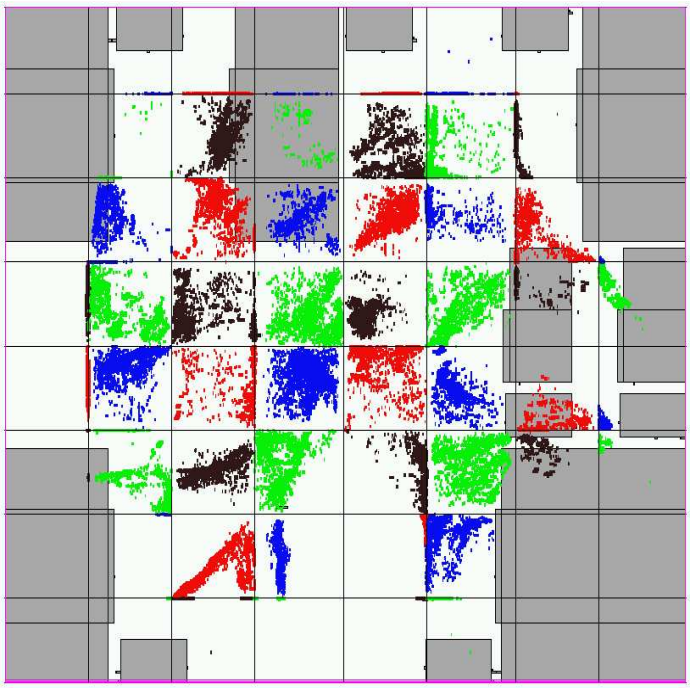
PARTITIONING$(r, C(r))$:

① Partition r into 4 subwindows $r_1, \ldots, r_4$;
② window_set := (window_set$\backslash\{r\}) \cup \{r_1, \ldots, r_4\}$;
③ Partition $C(r)$ into 4 subsets $C(r_1), \ldots, C(r_4)$ (using American Maps) and move cells into the corresponding window;

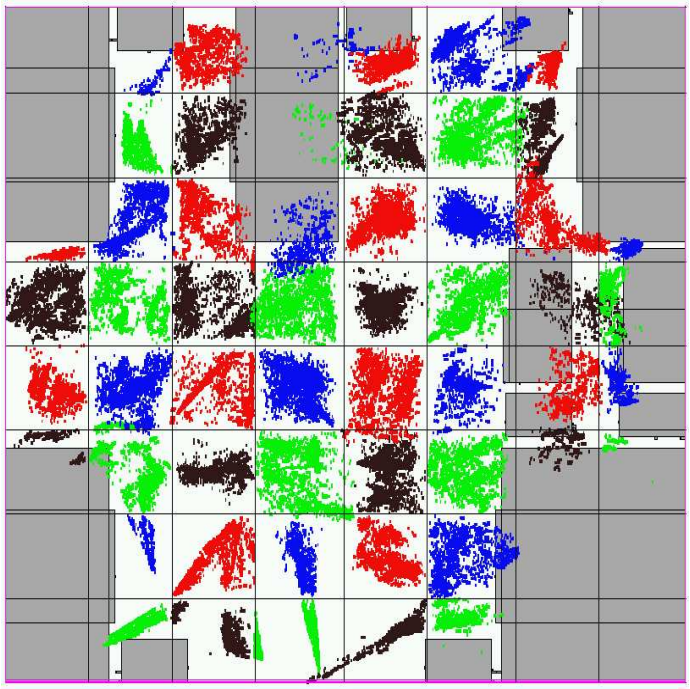
The algorithms starts with the complete list of circuits and the complete chip area. In step ② (level 0), a quadratic program (QP) is solved to compute locations for each circuit in r_0. For smaller nets, the clique net model is used while nets with many pins are replaced by a star (controlled by a user-defined parameter). Then, in each iteration of the loop in ③ (a level), the circuits are divided into subsets and assigned to subwindows by the function PARTITIONING. In order to ensure that in the following QP solutions each circuit is placed within its window, nets are split at the window borders (*terminal propagation*). Assume that we have bounds $a \leq x(c) \leq b$ for the x-coordinate of a circuit c. For each circuit c' that is connected to c but is placed, e.g., in a window to the right of b, we replace the connection to c' by an artificial connection between c and a fixed pin with x-coordinate b. Of course, connections to circuits c' which will be placed to the left of a are replaced by a connection to a fixed pin with x-coordinate a. For this construction, the additional Steiner nodes for the nets that are represented by a star also have to be assigned to certain windows and are placed in the same way as the circuits. Connections to fixed pins outside the bounds of a circuit are also split. Note that the net splitting is done for x- and y-coordinate independently, so for x-coordinate only the vertical cutlines and for the y-coordinates only the horizontal cutlines between the windows are considered. Especially, it is possible (and in fact will happen quite often) that a connection has to be split for the computation of the x-coordinate but not the y-coordinate, and vice versa. This splitting of the nets forces each circuit to be placed inside the window it is assigned to.

After all circuits are placed inside their window, all windows and circuit sets are partitioned. In BONNPLACE, this partitioning is, as described above, done in such a way that the capacity constraints are met (if possible) and that after moving each cell to its new window the total movement costs of the cells are minimized. If all circuits in a region are placed very close to the corner of their region, it is not reasonable to partititon them according to the small differences in their positions. An additional constrained QP before partitioning helps to solve this problem: if the weighted center of the circuits in a window is closer to one of the window corners than it could be in any disjoint placement of the circuits, an additional QP is solved with the constraint that the center of the circuits is placed at the closest possible position to the corner (so the center will be moved away from the corner). Figure 3.3 illustrates the effect of the additional constrained QP on the chip Jens. The first picture shows the placement at the end of level 3. The preplaced circuits are grey, for the other circuits the colors correspond to their windows. Especially in the outer parts of the chips, circuits tend to be placed as close to the chip's center as possible. Obviously, for these regions, a partitioning based on this placement would be useless. Figure 3.3 (b) shows how the QP solution with the additional constraints for some of the regions looks like. Using this placement we can decide which of circuits in the outer regions have the most important conections to the center. Note that due to the additional constraints it is possible that circuits are placed outside the window they are assigned to. Nevertheless, they will not be assigned to a different window in this step.

An important step in BONNPLACE is REPARTITIONING, a function that tries to find local improvements of the placement. It considers areas consisting of 2×2-windows (i.e., sets of four windows touching each other in one point) and computes a new partitioning for this region and new positions for the cells in the region. It accepts the new placement if the weighted netlength has decreased. This is done for all 2×2-windows. This loop is called



(a) Placement at the end of level 3



(b) Solution of the constrained QP before the partitioning in level 4

Figure 3.3: The effect of the constrained QP before the partitioning step.

repeatedly (with different orders of the 2×2-windows) as long as it yields a considerable improvement of the wire-length. Repartitioning enables the cells to leave the region they are currently placed in. It has also been used by Huang and Kahng [1997] in a minimum-cut-based placer.

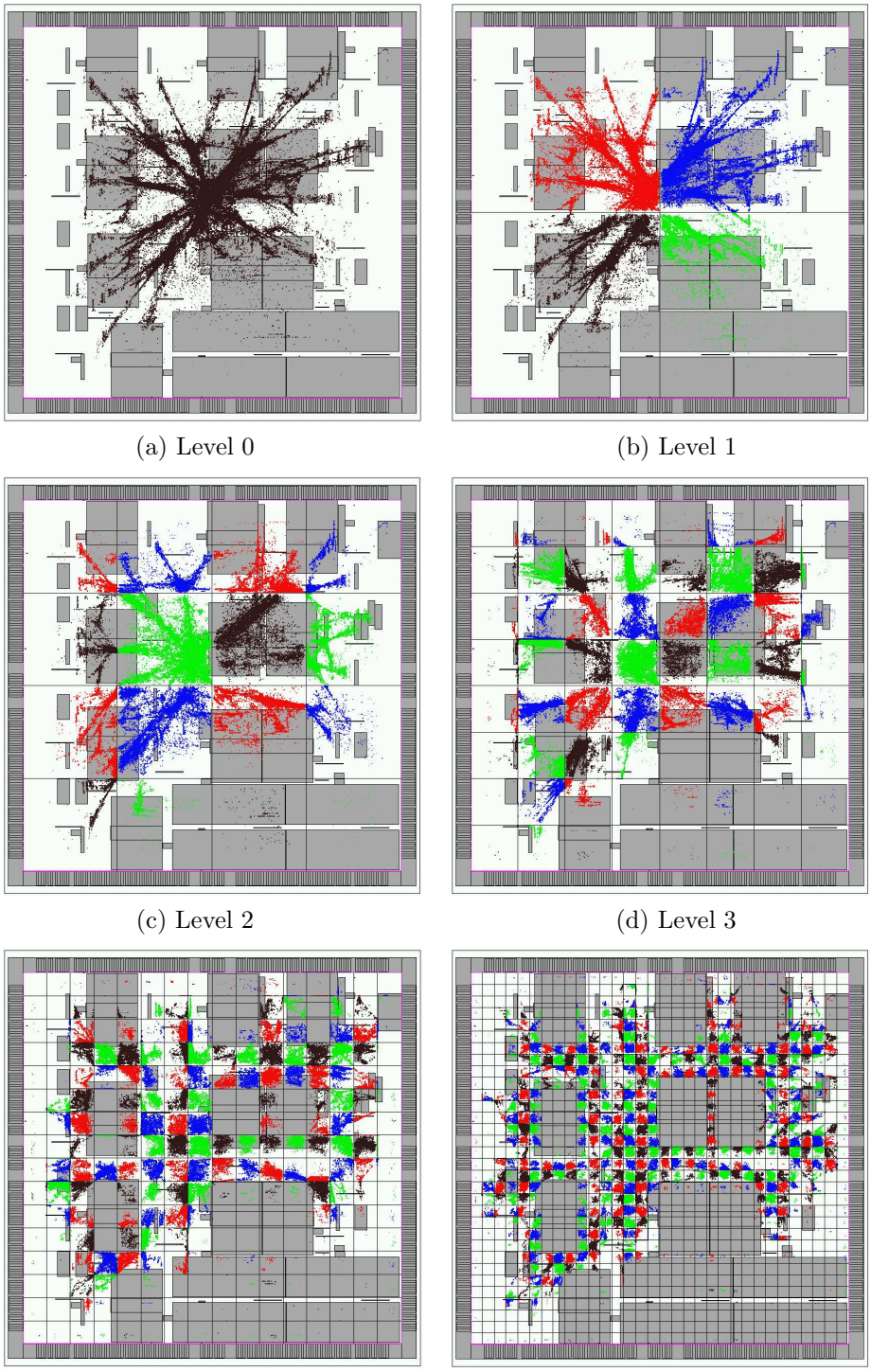
As the windows get smaller and smaller during a placement run, macros will sooner or later become too big to be handled reasonably in a partitioning step. Therefore, all macros that are too big compared to the window size will be fixed at the beginning of a level. The macros to be fixed in a level are legalized greedily, i.e., the algorithm places them one after the other to a free position that is as close as possible to its position at the end of the previous level.

The capacity of a window is not only determined by the unblocked area within the window but also by the maximum allowed density. The placement density is controlled by two user-defined parameters. The first parameter defines the maximum density at the beginning of the placement process, the second one is the increment of the allowed density in each level (typically 1%).

The loop in step ③ stops when the size of the windows is small enough. Especially, each window has to be contained in a single circuit row. Then, the cells are spread out quite well on the chip area and we can call the function Legalization that computes the detailed placement of the cells.

Figure 3.4 visualizes the first levels of a run of BonnPlace on the chip Max that consists of 512 375 circuits. The pictures show the placment at the end of the corresponding level. The grey objects are preplaced macros, the other circuits are colored according to the window they are placed in.

Our global placer is based on the same general strategy as the BonnPlace algorithm described above but improves on it with respect to several aspects. The most important contribution is a new partitioning method that is much more flexible than the partitioning based on American Maps as it can handle any number of subregions and any cost function. We will apply a new efficient algorithm for the Transportation Problem that will be described in the next chapter.



(a) Level 0 (b) Level 1

(c) Level 2 (d) Level 3

(e) Level 4 (f) Level 5

Figure 3.4: Levels 0 to 5 of a BonnPlace run.

Chapter 4

A Faster Transportation Algorithm

In this chapter, we will present a new efficient algorithm for the TRANSPORTATION PROBLEM. For unbalanced instances, this is the fastest known algorithm. The TRANSPORTATION PROBLEM is a classical, well-studied optimization problem and has an important application in our global placement algorithm.

4.1 Transportation Problems

A crucial step in our global placement algorithm consists of assigning circuits to parts of the chip area meeting some capacity constraints. Formally, we are interested in the solution of the following problem:

MULTISECTION PROBLEM

Instance:
- Finite sets C and R.
- *Sizes* $\text{size} : C \to \mathbb{R}_{\geq 0}$.
- *Capacities* $\text{cap} : R \to \mathbb{R}_{\geq 0}$.
- *Costs* $d : C \times R \to \mathbb{R}$.

Task: Find a mapping $g : C \to R$ with $\sum_{c \in C : g(c) = r} \text{size}(c) \leq \text{cap}(r)$ (for all $r \in R$) minimizing $\sum_{c \in C} d((c, g(c)))$.

We may assume that $\sum_{c \in C} \text{size}(c) \leq \sum_{r \in R} \text{cap}(r)$, since otherwise no solution could exist. The problem is a generalization of the ASSIGNMENT PROBLEM where the sizes and capacities are all 1. On the other hand, it is a special case of the so-called GENERALIZED ASSIGNMENT PROBLEM where the size of an element C also depends on the element R it is assigned to.

For the rest of this chapter, we define $n := |C|$ and $k := |R|$. We are interested in instances where the number n can be huge but where k is small, so we need an algorithm that handles efficiently unbalanced instances.

To decide if a solution of the MULTISECTION PROBLEM exists, is NP-complete since it contains the NP-complete PARTITIONING PROBLEM. We can relax the problem by skipping the restriction that each $c \in C$ has to be assigned completely to a single region. This leads to a fractional version of the MULTISECTION PROBLEM that is called TRANSPORTATION PROBLEM. Given an instance of the MULTISECTION PROBLEM, we formulate the TRANSPORTATION PROBLEM as a MINIMUM COST FLOW PROBLEM:

TRANSPORTATION PROBLEM

Instance: A flow network (G, b, u, cost) with

- a directed graph G with vertex set $V(G) = C \dot\cup R \dot\cup \{s, t\}$ and edge set $E(G) = (C \times R) \cup (\{s\} \times C) \cup (R \times \{t\})$,
- supply and demand values $b : V(G) \to \mathbb{R}$ with $b(s) = \sum_{c \in C} \text{size}(c) = -b(t)$ and $b(c) = b(r) = 0$ for $c \in C$ and $r \in R$,
- edge capacities $u : E(G) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ with $u((s, c)) := \text{size}(c)$, $u((r, t)) := \text{cap}(r)$, and $u((c, r)) = \infty$ for $c \in C$ and $r \in R$,
- edge costs $\text{cost} : E(G) \to \mathbb{R}_{\geq 0}$ with $\text{cost}((s, c)) = 0$, $\text{cost}((r, t)) = 0$ and $\text{cost}((c, r)) = d((c, r))$ for $c \in C$ and $r \in R$.

Task: Find a minimum cost flow f in the flow network (G, b, u, cost).

Note that we have capacities only on edges incident to the artificial nodes s and t, so our instances can be regarded as uncapacitated.

Let f be a solution of the TRANSPORTATION PROBLEM. For $c \in C$, we define $\tau_f(c) := |\{r \in R \mid f((c, r)) > 0\}|$. Let $F_f := \{c \in C \mid \tau_f(c) > 1\}$ be the set of elements $c \in C$ with outgoing flow into more than one edge. Obviously, each feasible solution f of the TRANSPORTATION PROBLEM with $F_f = \emptyset$ corresponds to a feasible solution of MULTISECTION PROBLEM of the same costs, and vice versa.

Vygen [1996] has already shown that an optimum solution f can be transformed efficiently to an optimum solution g with $|F_g| \leq k - 1$ (in fact, Vygen showed this for $k = 4$, but the construction is the same for any k).

For the analysis of our algorithm, we need a slightly stronger result:

Lemma 4.1 *Given an instance $(G, b, u, cost)$ of the* TRANSPORTATION PROBLEM, *an optimum solution f, and the set F_f, we can transform f in time $O(k^2 \cdot |F_f|)$ into an optimum solution g of $(G, b, u, cost)$ such that*

(i) $|F_g| \leq k - 1$, *and*

(ii) $\sum_{c \in F_g} \tau_g(c) \leq 2k - 2$.

Proof: We start by setting $g := f$. During the transformation, g will always stay an optimum solution. We construct an bipartite undirected graph H_g on the vertex set $V(H_g) := F_g \cup R$ and edge set $E(H_g) = \{\{c, r\} \mid g((c, r)) > 0\}$.

Claim: If H_g does not contain a cycle, g meets conditions (i) and (ii).

Proof of the claim: Assume that there is no cycle in H_g. The graph H_g can be constructed in the following way: we start with the set R as vertex set and no edges (thus, we have k connected components). Then, we traverse the elements of F_g in arbitrary order and add one after the other to the graph (with the incident edges). As we will not get a cycle, the number of connected components is reduced by $\tau_g(c) - 1$ when adding an element c, so we have $\sum_{c \in F_g}(\tau_g(c) - 1) \leq k - 1$. Since $\tau_g(c) > 1$ for all $c \in F_g$, this proves $|F_g| \leq k - 1$ and hence $\sum_{c \in F_g} \tau_g(c) \leq |F_g| + k - 1 \leq 2k - 2$. This proves the claim.

The claim shows that we are done if there is no cycle in H_g. Hence let us assume that H_g contains a cycle $(\{r_1, c_1, r_2, c_2, \ldots, c_j, r_{j+1} = r_1\}, \{\{c_i, r_i\} \mid i = 1, \ldots j\} \cup \{\{c_i, r_{i+1}\} \mid i = 1, \ldots j\})$. Then, for sufficiently small $\epsilon > 0$ also g' and g'' with $g'((c_i, r_i)) := g((c_i, r_i)) + \epsilon$, $g'((c_i, r_{i+1})) := g((c_i, r_{i+1})) - \epsilon$, $g''((c_i, r_i)) := g((c_i, r_i)) - \epsilon$, $g''((c_i, r_{i+1})) := g((c_i, r_{i+1})) + \epsilon$, for $i = \{1, \ldots, j\}$, and $f'((c, r)) := g''((c, r)) := g((c, r))$, for all other edges $(c, r) \in G$), are feasible solutions. Since g is optimum and the cost of g is the arithmetic mean of the costs of g' and g'', both g' and g'' are optimum, too. If we choose ϵ as big as possible, then we have $\sum_{c \in F_{g'}} \tau_{g'}(c) < \sum_{c \in F_g} \tau_g(c)$ or $\sum_{c \in F_{g''}} \tau_{g''}(c) < \sum_{c \in F_g} \tau_g(c)$

Then, we set $g := g'$ or $g := g''$, respectively, and iterate this method until $\sum_{c \in F_g} \tau_f(c) \leq 2k - 2$. As we have $\tau_g(c) \geq 2$ for each $c \in F_g$, then condition (i) will be met, too. Once we know the set F_g, we can transform g into g' or g'' (with the above properties) in time $O(k)$, and the whole computation can be done in time $O(k^2 \cdot |F_f|)$. □

4.2 Previous Approaches

The fastest known strongly polynomial algorithm for general uncapacitated Minimum Cost Flow Problems is described by Orlin [1993] and can be implemented to run in time $O(|V(G)|(\log |V(G)|)(|E(G)| + |V(G)| \log(|V(G)|)))$. Applied to the Transportation Problem, the algorithms have a running time of $O((n^2k + nk^2 + (n + k)^2 \log(n + k)) \log(n + k))$ which is $O(n \log n(nk + n \log n))$ for $k < n$ and $O(n^2 \log^2 n)$ if k is constant. For our instances, these algorithms are much too slow, since their running times grows more than quadratically with n.

There are other algorithms that exploit the special structure of the minimum cost flow instances in the Transportation Problem. Kleinschmidt and Schannath [1995] describe a strongly polynomial algorithm with running time $O(n^2k \log n)$ (for $k < n$) which is slightly better than the direct application of the general minimum cost flow algorithm mentioned above (if $k = o(\log n)$) but is still quadratic in n. Tokuyama and Nakano [1992, 1995] improve this result (for $k = o(n/\log n)$) by showing how Orlin's minimumcost flow algorithm can be implemented to run in $O(nk^2 \log^2 n)$ on transportation instances.

Matsui [1993] even claims to have an algorithm with running time $O(n(k!)^2)$ which would

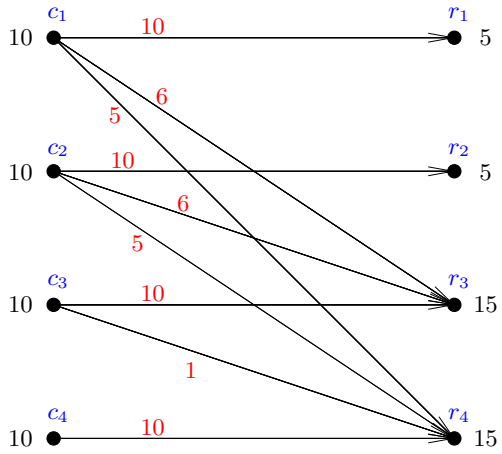


Figure 4.1: A counterexample to the correctness of the algorithm described by Matsui [1993].

be linear in n if k is a constant. He considers instances with $\sum_{c\in C} \text{size}(c) = \sum_{r\in R} \text{cap}(r)$ which, obviously, does not really change the complexity of the problem. However, the algorithm contains an error. The method works in two steps: First, an "almost optimal" mapping $h : C \times R \to \mathbb{R}_{\geq 0}$ is computed. Here, h is called "almost optimal", if the following conditions are fulfilled:

(1) $\sum_{r\in R} h((c,r)) = \text{size}(c)$ for all $c \in C$,

(2) h is an optimal solution for the TRANSPORTATION PROBLEM on the instance (G, b', u, cost) with $b'(c) := b(c)$ for $c \in C$ and $b'(r) := \sum_{c\in C} h((c,r))$, and

(3) $|\{r \in R \mid \sum_{c\in C} h((c,r)) < \text{cap}(r)\}| = 1$.

For an assignment $h : C \times R \to \mathbb{R}_{\geq 0}$, the elements of $\{r \in R \mid \sum_{c\in C} h((c,r)) < \text{cap}(r)\}$ are called *undersaturated*, so after the first part of the algorithms, there is only one undersaturated node. In a second step, the almost optimal solution is transformed into an optimal solution of the initial problem. This step is based on the observation that an almost optimal solution can be made to an optimal solution without decreasing the flow on incoming edges of the undersaturated node. However, the computation of the almost optimal solution in the first step turns out to be wrong. The step starts with an assignment h meeting the first two conditions described above. Then, all undersaturated nodes are contracted, so the second step can be applied to the contracted graph yielding an optimal assignment. However, when this optimal assignment in the contracted graph is used in order to compute a new assignment in the original graph, this new assignment does not have to meet condition (2). Figure 4.1 shows an instance where the algorithm will fail. In this instance, we have $C = \{c_1, c_2, c_3, c_4\}$, $R = \{r_1, r_2, r_3, r_4\}$, $\text{size}(c) = 10$ for all $c \in C$, $\text{cap}(r_1) = \text{cap}(r_2) = 5$, and $\text{cap}(r_3) = \text{cap}(r_4) = 15$. The red numbers are the edge costs. The edges which are not shown have costs 100, so they will not be used in an optimal solution. First, the algorithm would set $h((c_i, r_i)) = 10$ for $i \in \{1, \ldots, 4\}$ and $h((c_i, r_j)) = 0$ for $i \neq j$. Then, the two undersaturated nodes r_3 and r_4 would be contracted and flow from edges (c_1, r_1) and (c_2, r_2) would be rerouted to edges connecting

c_1 and c_2 to the contracted node. In the uncontracted graph, the result would be a mapping $h' : C \times R \to \mathbb{R}_{\geq 0}$ with $h'((c_1, r_1)) = h'((c_2, r_2)) = h'((c_1, r_4)) = h'((c_2, r_4)) = 5$ and $h'(e) = h(e)$ on all other edges. The residual graph corresponding to this flow contains a negative cycle $(c_3, r_4, c_2, r_3, c_3)$, so it is not optimal (as it has been claimed in the paper) and will not lead to an optimal solution.

As Matsui's [1993] algorithm turns out to be wrong, the fastest previously known algorithm for transportation instances with fixed k was the $O(n \log^2 n)$ algorithm by Tokuyama and Nakano [1992, 1995].

For some special cases of the TRANSPORTATION PROBLEM, faster algorithms are known. If $k = 2$, then the problem reduces to the FRACTIONAL KNAPSACK PROBLEM. The unweighted version of this problem (i.e., size$(c) = 1$ for all $c \in C$) has been solved by Blum et al. [1973] with a linear-time algorithm. Adolphson and Thomas [1977], Johnson and Mizoguchi [1978], and Balas and Zemel [1980] show how the algorithm for the unweighted version can be used as a subroutine for a linear time algorithm of the FRACTIONAL KNAPSACK PROBLEM with weights. The algorithm of Blum et al. can also be applied almost directly to weighted instances as shown by Vygen [1996] (see also Korte and Vygen [2002]). Tokuyama and Nakano [1991] describe an algorithm that solves the TRANSPORTATION PROBLEM in time $O(n(k!)^2)$ if size$(c) = 1$ for all $c \in C$ and cap$(r) \in \mathbb{N}$ for all $r \in R$. Ahuja et al. [1994] consider several flow algorithms and examine how fast they can solve flow problems on bipartite graphs. They show how the minimum cost flow algorithms presented by Goldberg and Tarjan [1990] can be modified for the TRANSPORTATION PROBLEM to run in time $O(nk^2 + k^3 \log(k \max\{d((c, r)) \mid c \in C,\, r \in R\}))$ or in time $O(nk^2 \log(k \max\{d((c, r)) \mid c \in C,\, r \in R\}))$ even if there are capacities on the graph edges. These algorithms are not strongly polynomial but if $\max\{d((c, r)) \mid c \in C,\, r \in R\}$ is fixed, we get an algorithm with running time $O(nk^2)$, so the running time grow linearly in n. We have already mentioned in the previous section the linear-time algorithm by Vygen [1997] that solves the TRANSPORTATION PROBLEM for instances where R consists of the four quadrands of the plane and where one can assign to each element of C a position in the plane such that $d((c, r))$ is the L_1-distance between the position of c and r. These are very strong restrictions of the problem, but, as we have seen, this special case has an important application in VLSI placement.

For the bottleneck version of the TRANSPORTATION PROBLEM where the objective function to be minimized is $\max\{d((c, r)) \mid g((c, r)) > 0\}$, there is a linear time algorithm for instances with constant k described by Hochbaum and Woeginger [1999].

4.3 Our Algorithm

We will show how the TRANSPORTATION PROBLEM can be solved in time $O(nk^2(\log n + k \log k))$, so for any fixed value of k, the problem can be solved in time $O(n \log(n))$. As mentioned above, the fastest previous algorithm (for constant k) had a running time of $O(n \log^2(n))$.

In our algorithm, n minimum cost flow problems on graphs whose size depends only on k have to be solved, and the running time depends on how fast we can compute min-

imum cost flows on these small instances. For uncapacitated instances (G, b, u, cost) (i.e., $u(e) = \infty$ for all $e \in E(G)$) the best known strongly polynomial algorithms have a running time of $(|V(G)| \log |V(G)|(|E(G)| + |V(G)| \log |V(G)|))$ (Orlin [1993]), while for the general problem, the best known strongly polynomial running time is $(|E(G)| \log |E(G)|(|E(G)| + |V(G)| \log |V(G)|))$ (Orlin [1993] and Vygen [2002b]). In our smaller instances, most of the edges are uncapacitated (or their capacity is at least as big as the total flow to be realized) but there are some edges with capacities. With the following trick that is due to Ford and Fulkerson [1962] minimum cost flow instances with capacities can be transformed equivalently into (larger) uncapacitated instances:

Lemma 4.2 (Ford and Fulkerson [1962]) *Let $(G, b, u, cost)$ be a minimum cost flow instance with finite capacities on σ edges. Then, there is an equivalent uncapacitated minimum cost flow instance with $|V(G)| + \sigma$ nodes and $|E(G)| + \sigma$ edges.*

Proof: Let E_{cap} be the set of edges with finite capacities (so we have $u(e) = \infty$ for $e \in E(G) \setminus E_{\text{cap}}$). We define a graph G' with $V(G') := V(G) \cup E_{\text{cap}}$ and

$$E(G') = (E(G) \setminus E_{\text{cap}}) \cup \Big\{((v,w),v) \mid (v,w) \in E_{\text{cap}}\Big\} \cup \Big\{((v,w),w) \mid (v,w) \in E_{\text{cap}}\Big\}.$$

For $(v,w) \in E_{\text{cap}}$, we set $\text{cost}'(((v,w),w)) := \text{cost}((v,w))$ and $\text{cost}'(((v,w),v)) := 0$. For $e \in E(G) \setminus E_{\text{cap}}$, we set $\text{cost}'(e) := \text{cost}(e)$. Of course, for all $e \in E(G')$, we define $u(e) := \infty$. Figure 4.2 illustrates the transformation for a single edge (asuming that no

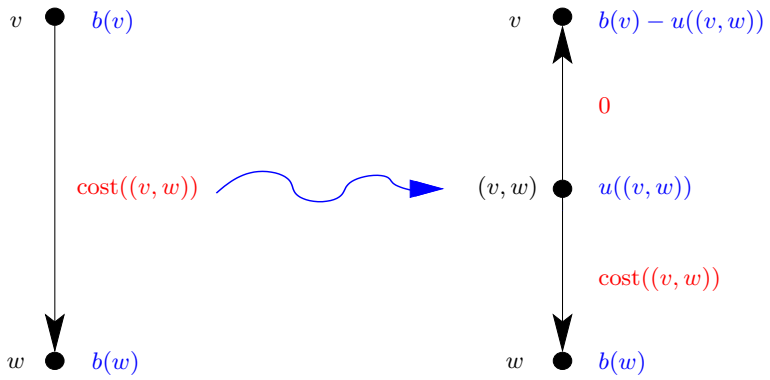


Figure 4.2: Transformation from an edge (v,w) with capacity $u((v,w)) < \infty$ to two uncapacitated edges. The supply/demand values are blue and the edge costs are red.

other edges leaving v and w are replaced). Obviously, $(G', b', u', \text{cost}')$ is a flow network, and G' has $|V(G)| + \sigma$ nodes and $|E(G)| + \sigma$ edges. It remains to prove that the two instances are equivalent.

First, let f be a flow in (G, b, u, cost). Then, we set $f'(((v,w),w)) := f((v,w))$ and $f'(((v,w),v)) := u((v,w)) - f((v,w))$ for $(v,w) \in E_{\text{cap}}$, and $f'(e) := f(e)$ for $e \in E(G) \setminus E_{\text{cap}}$. It is easy to see that f' is a flow in $(G', b', u', \text{cost}')$ with $\text{cost}'(f') = \text{cost}(f)$.

For the other direction, let f' be a flow in $(G', b', u', \text{cost}')$. We set $f((v,w)) := f'(((v,w),w))$ for $(v,w) \in E_{\text{cap}}$, and $f(e) := f'(e)$ for $e \in E(G) \setminus E_{\text{cap}}$. Again, it is not difficult to see that f is a flow in (G, b, u, cost) with $\text{cost}(f) = \text{cost}'(f')$. □

The idea of our algorithm is based on a SUCCESSIVE SHORTEST PATH ALGORITHM (see Jewell [1958], Iri [1960], Busacker and Gowen [1961]) that solves general MINIMUM COST FLOW PROBLEMS. For our instances, the SUCCESSIVE SHORTEST PATH ALGORITHM can be described as follows:

SUCCESSIVE SHORTEST PATH ALGORITHM

Input: An instance (G, b, u, cost) of the TRANSPORTATION PROBLEM, as described above.

Output: A minimum cost flow f in (G, b, u, cost).

① $f(e) := 0$ for all $e \in E(G)$.

② Let $C = \{c_1, \ldots, c_n\}$.

③ FOR($i = 1$; $i \leq n$; $i++$)
  WHILE($f((s,c_i)) < u((s,c_i))$)
    Find a shortest c_i-t-path P in $G_{f,u}$ (w.r.t. the residual edge costs).
    $\gamma := \min\{\min_{e \in E(P)} u_f(e), u_f((s,c_i))\}$.
    Augment f along (s,c_i) and P by γ.

The correctness of the algorithm follows from the following theorem (see, again, standard textbooks, e.g., Korte and Vygen [2002]):

Theorem 4.3 *Let $(G, b, u, cost)$ be an instance of a* MINIMUM COST FLOW PROBLEM *with conservative edge costs, and let f be a minimum cost flow in $(G, b, u, cost)$. Let $v, w \in V(G)$ be two vertices, and let P be a shortest v-w-path in $G_{f,u}$ (with respect to the residual edge costs $cost_f$). Let f' be a flow obtained by augmenting f along P by $\gamma \in \mathbb{R}$ where γ is at most the minimum residual capacity on P. Then, f' is a minimum cost flow in $(G, b', u, cost)$, where $b'(v) = b(v) + \gamma$, $b'(w) = b(w) - \gamma$ and $b'(x) = b(x)$ for $x \in V(G) \setminus \{v, w\}$.* □

This theorem shows that the SUCCESSIVE SHORTEST PATH ALGORITHM computes correctly a flow of minimum cost. On the other hand, it is quite obvious that *any* minimum cost flow g can be computed by successive augmentations along shortest paths: we run the SUCCESSIVE SHORTEST PATH ALGORITHM but set the capacity of each e to $\min\{u(e), g(e)\}$. If, during the main loop, one of the shortest paths that we compute in this modified instance was not a shortest path in the original instance, then the flow g was not optimum as it could be improved along a negative cycle.

This leads to the following corollary which shows that we can compute a minimum cost flow by iterative computations of minimum cost flows between a single supply node and a single demand node.

Corollary 4.4 *Let $(G, b, u, cost)$ be an instance of the* Minimum Cost Flow Problem *with conservative edge costs, and let f be a minimum cost flow in $(G, b, u, cost)$. Let $v, w \in V(G)$ be two vertices. Let g be a minimum cost flow in $(G_{f,u}, b', u_f, cost)$ where $b'(v) = \gamma$, $b'(w) = -\gamma$ and $b'(x) = 0$ for $x \in V(G) \setminus \{v, w\}$ for some $\gamma \in \mathbb{R}_{>0}$. Let f' be a flow obtained by augmenting f by g, i.e., for edges e with $f(e) = 0$, we set $f'(e) := g(e)$, and for edges e with $f(e) > 0$, we set $f'(e) := f(e) + g(e) - g(\overleftarrow{e})$. Then, f' is a minimum cost flow in $(G, b + b', u, cost)$.*

Proof: As g could have been computed by the Successive Shortest Path Algorithm, the augmentation by g can be replaced by a number of augmentations along shortest paths. Thus, the corollary follows from Theorem 4.3. □

We call an iteration of the main loop in step ③ of the Successive Shortest Path Algorithm a *phase* of the algorithm. In order to bound the running time of the algorithm, we have to bound the running time of a phase. However, even in the case of integer edge capacities, the number of augmentations for a single vertex $c \in C$ can be as big as $u((s, c))$ (if $\gamma = 1$ in each augmentation), so a single phase can have a running time that is exponential in the input. Let $f_0(e) := 0$ for $e \in E(G)$, and let f_i be the flow at the end of phase i. In order to get a polynomial running time, we can apply Corollary 4.4 and replace a complete phase of the algorithm by computing a $c_i - t$-flow of value $u((s, c_i))$ and minimum cost in $G_{f_{i-1}}$. This method yields a polynomial running time, but the residual graph $G_{f_{i-1}}$ will not be smaller than G. Fortunately, if we sort the nodes in C such that $u((s, c_1)) \geq u((s, c_2)) \geq \cdots \geq u((s, c_n))$, we do not have to consider the complete residual graph $G_{f_{i-1}}$ but a small subgraph G_i whose size does not depend on n. The vertex set $V(G_i)$ contains exactly the following nodes:

- $R \subset V(G_i)$.
- $c_i \in V(G_i)$.
- $t \in V(G_i)$.
- $F_{f_{i-1}} \subseteq V(G_i)$.
- For a vertex $r \in R$ let M_r^i be the set of all vertices $c \in C$ with $f_{i-1}((c, r)) = u((s, c))$. For each pair of vertices $r, r' \in R$ with $M_r^i \neq \emptyset$, $V(G_i)$ contains an arbitrary $c \in M_r^i$ with

$$\text{cost}((c, r')) - \text{cost}((c, r)) = \min\{\text{cost}((c', r')) - \text{cost}((c', r)) \mid c' \in M_r^i\}.$$

The edge set $E(G_i)$ contains exactly the following edges:

- $((R \times \{t\}) \cup (\{t\} \times R)) \cap E(G_{f_{i-1}}) \subset E(G_i)$.
- $((\{c_i\} \times R) \cup (R \times \{c_i\})) \cap E(G_{f_{i-1}}) \subset E(G_i)$.
- For each $c \in F_{f_{i-1}}$: $((\{c\} \times R) \cup (R \times \{c\})) \cap E(G_{f_{i-1}}) \subset E(G_i)$.
- Let $r, r' \in R$ be two vertices with $M_r^i \neq \emptyset$, and let $c \in M_r^i \cap V(G_i)$ be the corresponding vertex in $V(G_i)$. Then $\{(r,c),(c,r')\} \cap E(G_{f_{i-1}}) \subset E(G_i)$.

The size of G_i depends on the number of elements of $F_{f_{i-1}}$. However, as mentioned above, if $|F_{f_{i-1}}| > k-1$, we may apply Lemma 4.1 and find a flow f'_{i-1} of the same cost with $|F_{f'_{i-1}}| \leq k-1$. After each phase of the algorithm, we will call a subroutine ADJUST(f_i) that applies the algorithm in the proof of Lemma 4.1, so we always have $|F_{f_i}| \leq k-1$ and $\sum_{c \in F_{f_i}} \tau_{f_i}(c) \leq 2k-2$. Thus, we have $|V(G_i)| \leq k+1+1+(k-1)+k \cdot (k-1) = k^2+k+1$ and $|E(G_i)| \leq 2(k+k+(k-1) \cdot k + 4k \cdot (k-1)) = 10k^2 - 6k$. Note that the size of G_i does not depend on n but only on k.

TRANSPORTATION FLOW ALGORITHM

Input: An instance (G, b, u, cost) of the TRANSPORTATION PROBLEM (as described above)

Output: A minimum cost flow f in (G, b, u, cost).

① $f(e) := 0$ for all $e \in E(G)$.

② Sort the set of nodes in C such that $C = \{c_1, \ldots, c_n\}$ with $u((s,c_1)) \geq u((s,c_2)) \geq \cdots \geq u((s,c_n))$.

③ FOR($i = 1$; $i \leq n$; $i++$)
Construct G_i.
Compute a minimum cost flow g in $(G_i, b', u_f|_{E(G_i)}, \text{cost}_f|_{E(G_i)})$ where $b'(c_i) = u((s,c_i))$, $b'(t) = -b'(c_i)$ and $b'(v) = 0$ for $v \in (C \cup R) \cap V(G_i)$.
Augment f by g.
Set $f((s,c_i)) = u((s,c_i))$.
ADJUST(f).

Theorem 4.5 *The* TRANSPORTATION FLOW ALGORITHM *solves the* TRANSPORTATION PROBLEM *in time* $O(nk^2(\log n + k \log k))$.

Proof: Correctness: Consider a fixed iteration i. If we used the residual graph $G_{f_{i-1}}$ instead of the smaller graph G_i in the main loop of the algorithm, then the correctness would follow from Corollary 4.4.

We have to show that there is a feasible flow in $(G_i, b', u_{f_{i-1}}|_{E(G_i)}, \text{cost}_{f_{i-1}}|_{E(G_i)})$ and that a minimum cost flow in $(G_i, b', u_{f_{i-1}}|_{E(G_i)}, \text{cost}_{f_{i-1}}|_{E(G_i)})$ is a minimum cost flow in $(G_{f_{i-1}}, b', u_{f_{i-1}}, \text{cost}_{f_{i-1}})$. To see this, assume that g is a minimum cost flow in $(G_{f_{i-1}}, b', u_{f_{i-1}}, \text{cost}_{f_{i-1}})$ computed by the SUCCESSIVE SHORTEST PATH ALGORITHM. Let $P_1, \ldots, P_l$ be the corresponding sequence of augmenting paths. Assume that there is a path P_j containing a node that is not an element of $V(G_i)$. Choose j as small as possible with this property. The node sequence of P_j is of the form $c_{i_1}, r_{i_2}, c_{i_3}, r_{i_4}, \ldots, c_{i_{\lambda-1}}, r_{i_\lambda}, t$ with $\{c_{i_1}, \ldots, c_{i_{\lambda-1}}\} \subset C$ and $\{r_{i_2}, \ldots, r_{i_\lambda}\} \subset R$. Choose κ as small as possible such that $c_{i_\kappa} \notin V(G_i)$. Let g' be the flow after augmenting along the paths $P_1, \ldots, P_{j-1}$.

Since $c_{i_\kappa} \notin V(G_i)$, we have $f_{i-1}((c_{i_\kappa}, r_{i_{\kappa-1}})) \in \{u((s, c_{i_\kappa})), 0\}$. If $f_{i-1}((c_{i_\kappa}, r_{i_{\kappa-1}})) = 0$, then the flow on $(r_{i_{\kappa-1}}, c_{i_\kappa})$ must have been augmented during the first $j-1$ augmentations, so (by the choice of j) we can conclude $c_{i_\kappa} \in V(G_i)$. Thus, we assume that $f_{i-1}((c_{i_\kappa}, r_{i_{\kappa-1}})) = u((s, c_{i_\kappa}))$. Therefore $c_{i_\kappa} \in M^i_{i_{\kappa-1}}$. Note that the total flow augmented along the paths $P_1, \ldots P_{j-1}$ is smaller than $u((s, c_i))$. So there must be a $c_{i'} \in V(G_i)$ with $f_{i-1}((c_{i'}, r_{i_{\kappa-1}})) = u((s, c_{i'}))$, $g'((r_{i_{\kappa-1}}, c_{i'})) < u((s, c_i)) \leq u((s, c_{i'})) = f_{i-1}((c_{i'}, r_{i_{\kappa-1}}))$ and $\text{cost}((c_{i'}, r_{i_{\kappa+1}})) - \text{cost}((r_{i_{\kappa-1}}, c_{i'})) \leq \text{cost}((c_{i_\kappa}, r_{i_{\kappa+1}})) - \text{cost}((r_{i_{\kappa-1}}, c_{i_\kappa}))$. Therefore, we can replace c_{i_κ} in P_j by $c_{i'}$ without increasing the length of P_j. This can be done with any vertex outside $V(G_i)$ occurring in an augmenting path. Therefore, it is sufficient to consider the subgraph G_i.

Running time: Obviously, step ① can be done in time $O(nk)$ and step ② takes time $O(n \log n)$. The construction of G_i can be done in time $O(k^2 \log n)$ for each iteration if one stores each set M^i_r in $k-1$ heaps: For each pair $r, r' \in R$ with $r \neq r'$, we use a heap to store the elements c of M^i_r with $\text{key}_{r,r'}(c) = \text{cost}(c, r') - \text{cost}(c, r)$. For the worst-case running time, it does not matter which kind of heaps we use as long as finding an element with smallest key, inserting, and deleting of an element can be done in time $O(\log n)$ for a heap with n elements.

If we apply the fastest minimum cost flow algorithms for instances with capacities, this would lead to a running time of $O\big(|E(G_i)| \cdot \log |E(G_i)| \cdot \big[|E(G_i)| + |V(G_i)| \cdot \log |V(G_i)|\big]\big) = O(k^4 \log^2 k)$ for each minimum cost flow computation. We can improve this by a factor of $\Theta(k \log k)$: First, we do not have to add the elements c of the sets $M^i_r \cap V(G_i)$ explicitly to our vertex set as they have excatly one incoming edge (r, c) and one outgoing edge (c, r'), so we may connect r and r' directly by an edge. This edge has capacity size(c), but by construction this capacity is at least as big as the flow to be realized in the iteration, so the edge can be considered as uncapacitated. The only edges with capacities are the ones connecting elements of $F_{f_{i-1}}$ to elements of R. But according to Lemma 4.1, their number is $O(k)$. Hence, we can apply Lemma 4.2 and replace them by $O(k)$ additional vertices and edges. Putting this together, we can compute the single minimum cost flows on uncapacitated instances with $O(k)$ vertices and $O(k^2)$ edges. Using Orlin's [1993] algorithm, this can be done in time $O(k^3 \log k)$.

Using Lemma 4.1, the flow can be adjusted in time $O(k|V(G_i)|) = O(k^3)$. To update the heaps after a flow augmentation, there are $O(k^2)$ delete-operations (at most one per heap) necessary. The number of insert-operations after a flow augmentation is also $O(k^2)$: only the elements of $V(G_i)$ can be inserted to a heap, and for each heap that stores a set M^i_r, at most one of the elements $c' \in V(G_i)$ for which there is a vertex $r' \in R \setminus \{r\}$ with $f_{i-1}((c', r')) = u((s, c'))$ can be inserted to M^i_r. Since there are at most $k-1$ elements

$c' \in V(G_i)$ for which there is no $r' \in R$ with $f_{i-1}((c', r')) = u((s, c'))$ and each vertex can be added to at most $k-1$ heaps, we need at most $O(k^2)$ insert-operations. Since no heap contains more than n elements, each operation can be done in time $O(\log n)$. Therefore, the $k(k-1)$ heaps can be updated in time $O(k^2 \log n)$. □

Chapter 5

Global Placement

This chapter contains a description of our global placement algorithm. We will explain the algorithm, show how it can be implemented efficiently, and analyze it with a number of experiments.

5.1 Global Placement by Multisection

Our global placer follows a top-down partitioning strategy. For the partitioning steps we apply the idea that has been proposed by Vygen [1997]: Starting with a placement that minimizes quadratic netlength, we try to move the circuits to subregion meeting the capacity constraint while changing the placement as little as possible. The main difference between our placer and the method described by Vygen is our new partitioning algorithm. In the old algorithm, the area is divided into four quadrants and the L_1-distance had to be used to compute the cost of moving a circuit to a region. In contrast to that, we will apply the Transportation Flow Algorithm presented in the previous chapter.

The theoretical running that we can show for the Transportation Flow Algorithm, $O(n \log n)$ (for fixed k), is slightly worse than the running time of the American-Map Algorithm ($O(n)$) described by Vygen [1997], but the Transportation Flow Algorithm solves a much more general problem, as it can handle

- any number of regions,
- any shape of regions, and
- any function defining the distance between circuits and regions.

Figure 5.1 shows how a partitioning with L_1-movement as objective function but nine subwindows may look like. The grey rectangle in the lower left corner are blockages, and, as in Figure 3.2, the circuits' colors indicate the region they are assigned to. Similar to American Maps, the different circuit sets can be separated by horizontal, vertical or diagonal lines, so the geometric structures we get here can be seen as a generalization of

the American Maps. Of course, for other objective functions, the partitioning may look completely different.

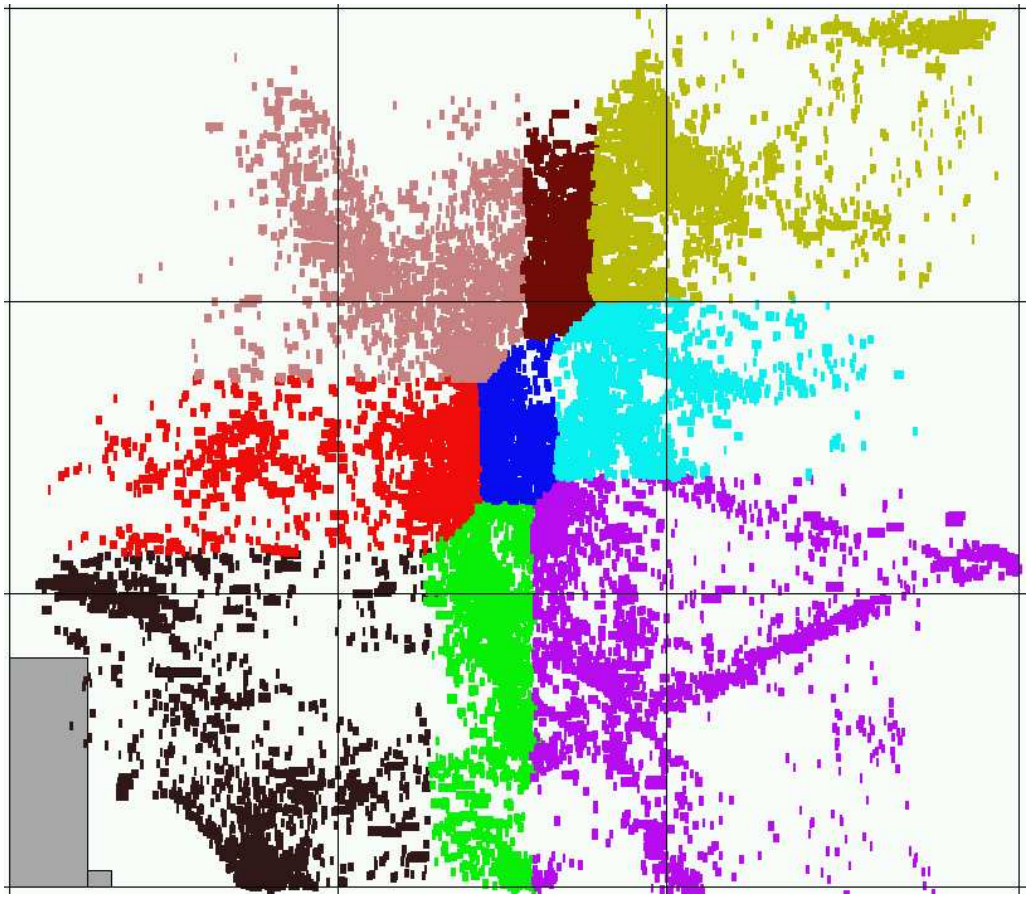


Figure 5.1: A multisection example.

The estimated cost for moving circuit c to region r may depend, for example, on the Euclidean distance, on possible connections between c and preplaced objects, on the size or the shape of c and r, or on the weights of the nets connected to c.

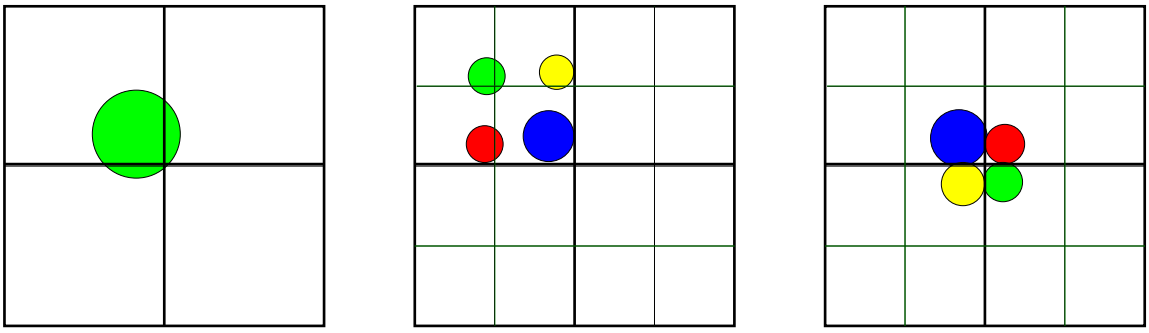
5.2 Improved Partitioning Methods

In partitioning-based placement strategies, as described above, we have to solve the problem of dividing a set of circuits assigned to a specified area into subsets and to assign these circuit subsets to subsets of the area meeting some capacity constraints. In this pure recursive approach (i.e., without local optimization steps like repartitioning), no circuit can leave the region it has been assigned to in an early stage of the algorithm. Therefore, wrong decisions in early steps of the run will create more work for local optimization afterwards, which, or course, increases running time.

We describe a way to handle the problems of recursive partitioning in the context of a quadrisection based algorithm, but it should be noted that the same problems can occur if one considers more than 4 windows in a multisection based placer, and the ideas presented here can also be applied to this case. Other ideas presented in this chapter will not improve the quality of results but the running time.

5.2.1 Global Partitioning

Figure 5.2 visualizes a situation when a recursive partitioning strategy would lead to unnecessary movements. According to the QP solution, a set of circuits is placed near the center of the chip area (symbolized by the green cycle in (Figure 5.2 (a)). In level 1, the set is assigned to the upper left region. Assume that none of the windows that occur in level 2 is able to contain the whole set of circuits. Then, if the circuits are not allowed to leave the upper left region in the partitioning step of level 2, they have to be assigned to the 4 upper left subregions (b). The problem is, that the algorithm minimizes movement *locally* (i.e., in the subregions of a given region) but not *globally* for the whole chip.



(a) (b) (c)

Figure 5.2: The first two levels of a quadrisection based placement run. The green cycle in picture (a) symbolizes the placement of the set of circuits in the QP solution of level 0. We assume that, in level 1, the upper left window is big enough to keep all the circuits while the capacities of the windows in level 2 are too small to contain all of them. A standard quadrisection step would assign the circuits to the four upper left regions (b) while an assignment to the four center regions (c) would lead to a placement much closer to the initial QP solution. Obviously, depending on the exact circuit position in picture (a), the assigment shown in picture (c) can correspond to a much smaller total movement than the one shown in picture (b).

To cope with this problem, we can make use of the fact that the Transportation Flow Algorithm that is used in our partitioning steps can handle any number of regions: Instead of solving one partitioning in each single window, we can run a multisection partitioning on all windows simultaneously where each circuit can be placed in *any* subregion. However, after a few levels, the number of regions would be too large to run the Transportation Flow Algorithm whose running grows fast with this number. But running time is not the only problem of this approach, we also need a new method that replaces the constrained QPs. These QPs, that move the centers of gravity of groups of circuits to certain positions, are necessary in order to get significant differences in the coordinates

of the circuits before a partitioning. However, in the standard recursive approach, we use these QPs to move the center of gravity of each the circuit set towards the center of the old region it is assigned to. If we allow the circuits to leave their region in partitioning, this strategy is no longer reasonable, since we do not know where to move their center of gravity. Note that circuits can even be moved out of their window by the constrained QP which would lead to very bad assignments if the partitioning step does not bring them back to (the subsets of) their old area. It is not clear how we can add additional constraints to a QP to spread out circuits a little bit *without* moving their center of gravity. Possibly, some iterations of force-directed placement or similar techniques that spread the circuits could help here.

5.2.2 Iterative Partitioning

Iterative partitioning, that we will propose in this subsection, allows circuits to leave the area they were assigned to during partitioning and therefore reduces the amount of work that has to be done by local optimization steps.

In a quadrisection step, we are given a partitioning of the chip area into a set R of $l \times l$ rectangles, given by $2(l-1)$ cut lines, and a subdivision R' of R into $(2l) \times (2l)$ rectangles, given by $2(2l-1)$ cutlines such that each region $r \in R$ consists of exactly 4 elements of R'. In addition, we are given an assignment of the circuits to the rectangles in R and for each circuit a position in the rectangle it is assigned to. Now, we ask for an assignment of the circuits to the subregions in R' such that, if we move each circuit into its subregion, the total movement costs are minimized.

The iterative partitioning algorithm we propose works in two steps: First, each circuit is assigned to the subregion $r' \in R'$ it is placed in. If this assignment does not violate any capacity constraints, it is obviously optimum since the corresponding movement is 0. However, in general, we cannot hope that no subregion contains more circuits than fit into it. In the case that some subregions are too full, we run, similarly to the standard partitioning approach, single quadrisection steps on 2×2-windows, but we do not only consider 2×2-windows that correspond to single windows of the initial partitioning. Instead, we choose the windows in the following way. For each 2×2-window W we compute the maximum overload $\mathrm{ov}_{\max}(W)$ of one of its 4 single windows and the average overload $\mathrm{ov}_{\mathrm{av}}(W)$, i.e., the total size of circuits assigned to the 4 subwindows divided by the total capacity of the 4 windows. In addition, we compute the distance $d(W)$ of the center of gravity of the circuits in the subwindow with the largest overload to the center of the 2×2-window. For each window W, these numbers are combined in a key $(\max(1, \mathrm{ov}_{\mathrm{av}}(W)), -\mathrm{ov}_{\max}(W), d(W))$. Then, we ask for a window with lexicographically smallest key. In other words, we ask for a 2×2-window that has enough capacity to contain all circuits assigned to it, but where one of the subwindow has a large overload, and, as a third criterion, where the circuits in the subregion with the largest overload are placed near the center of the 2×2-window. We run a single repartitioning step on the 2×2-window with the smallest key and accept the new placement if the maximum overload is reduced (which will normally be the case). Afterwards we update the keys and continue with the next 2×2-window until there are no significant overloads any more. For example, in Figure 5.2 we would have first chosen the

2×2-window in the center and a single quadrisection step on this window would have lead to a legal assignment to the regions. The whole procedure is comparable to the standard repartitioning loop, only the way how we choose the 2×2-windows to be optimized and the way how we decide if a new placement is accepted are different.

5.2.3 More Accurate Movement Costs

One application of the flexibility of multisection is quite obvious: instead of using the L_1-distance between a circuit c and a region r, we can use the distance between C and the nearest free unblocked position in r where c can be placed. This is, of course, a more accurate estimation of the cost for moving c to r. Nevertheless, experiments have clearly demonstrated that such a more accurate estimation does not correspond to shorter netlength. Often it can even lead to much worse results. For example, consider a group of strongly connected circuits that are placed by the QP solution on top of a large macro. Assume that in the next partitioning a vertical cutline in the macro is added and that both the subregion to the left of the macro and the subregion to the right of the macro have enough capacity to contain the whole group. In this situation, one rather wants to move the whole group of circuits to one side of the macro than to partition the group with minimum movement. However, the more accurate cost function will compute a borderline between two subgroups of the circuits in the middle of the macro and it is quite likely that such a partition produces a large cut.

5.2.4 Partitioning with Lookahead

Another way to avoid wrong decisions in partitioning is to have a look in the next level when computing the assignment of the circuits to the windows. If we want to partition a window in, say, 4 subwindows $r_1, \dots r_4$, we partition each subwindow r_i into 4 windows $r_i^1, \dots r_i^4$ before we assign the circuits. Then, we compute a multisection for the circuit list and the 16 subwindows. We assign a circuit to window r_i if it was assigned to one of the windows $r_i^1, \dots r_i^4$. This way, we can see in advance if circuits have to be moved a long distance inside its window r_i in the next level. On the other hand, these partitioning steps are quite time-consuming, and experiments have shown that the iterative partitioning produces better results in terms of netlength.

5.2.5 Movebound-Aware Partitioning

In some cases it may be desirable to define certain areas for specified circuits in which they have to be placed (*movebounds*). This can be useful if there a regions of different voltage or if there are IO-drivers or IO-receivers (i.e., circuits connected to an IO-pin) that have to be placed near their IO-pin. In addition, movebounds can be used if one wants to compute a new placement that does not differ too much from a previous solution. For our global placer, the last application is not that important as it is quite stable, so a new placement with slightly different parameters or local changes in the netlist will generally lead to similar results.

There are two ways how multisection can take such *movebounds* into account. A simple method consists of defining infinite (or at least very big) costs for moving a circuit into a window that does not intersect the allowed area for that circuit. This method is easy to implement and fast but it can compute wrong assignments, if for a bigger number of circuits, the intersection between their allowed regions and one of the subwindows is the same small (but non-empty) area. Then, possibly all of them could be assigned to that window though only a few of them can be placed there without a movebound violation. To cope with that problem we may partition the subwindows into parts such that for each circuit and each part, the part is either completely contained in the circuit's allowed area or has no intersection with it. Then, the assignment to the smaller parts can correctly take care of the movebounds. If there is only a very small number of different movebounds (which should be the case, e.g., in the presence of different voltage regions), this approach should work. In our applications, we only consider movebounds for IO-pads, so there are many different allowed regions, and since these regions are spread over the chip area (so the worst-case behaviour described above will hardly occur), we only tested the simple first method.

5.2.6 Reducing the Number of Levels

Obviously, multisection can be applied to reduce the number of levels. We could just replace all quadrisection steps by, for example, 3×3-partitionings which would decrease the number of levels. However, experiments show that this approach does not improve the running time very much, because the single levels need more time and, in addition, more repartitioning steps are necessary to get placements of the same quality.

However, at the end of a placement run, the number of horizontal cutlines will generally be bigger than the number of vertical cutlines. This is due to the fact that we consider row based designs, so the windows at the end of the global placement are part of row. The width of these windows is controlled by a user parameter but it is reasonable to choose the width in such a way that the windows may contain about three or four standard circuits, so the widths of the windows is significantly bigger than their heights. Therefore, only horizontal cuts (bisections) are computed in the last levels. We can skip the last levels if we apply 2×3-partitionings in the levels before. If $\#\text{cut}_y$ and $\#\text{cut}_x$ are the numbers of the horizontal and vertical cutlines at the end of global placement, then the number of levels can be reduced from $\lceil \log_2(\#\text{cut}_y) \rceil$ to $\max\{\lceil \log_3(\#\text{cut}_y) \rceil, \lceil \log_2(\#\text{cut}_x) \rceil\}$. We will demonstrate that we can reduce the number of levels with this trick without increasing the netlength significantly.

5.2.7 3×3-Repartitioning

The size of the regions that we consider in repartitioning can be controlled by a parameter. In the standard version on BonnPlace, we apply the repartitioning steps to 2×2-windows. In order to receive an improved netlength, 3×3-windows can be used. Experiments will show that having such a more global view in repartitioning leads to better results but also increases the running time.

5.3 Accelerating the Algorithm

Experiments show that the most time-consuming parts of our global placer are the computation of the QP solutions and the repartitioning. To save running time in these steps, we will describe how they can be implemented efficiently using a new net model and parallelization.

5.3.1 Hybrid Net Model

As mentioned in Section 2, clique is among the linear net models with fixed topology the best approximation to the Steiner tree net model. Also for quadratic netlength, it seems to be reasonable to connect all pairs of pins in a net as we do not know in advance how the routing Steiner tree connecting the pins in a net will look like. However, we have already noted in Chapter 2 that clique and star with quadratic edge lengths are equivalent, if we weight the edges of the star by a factor of $\frac{n}{n-1}$ (for a net with n pins). This is not longer true if we split nets at cutlines between the partitioning regions. If we use the star net model together with a paritioning of the chip area into windows, we also have to assign the additional Steiner node in each net to one window. Obviously, it is a reasonable choice to assign the Steiner node of a net to a window such that the difference between the number of pins of the net to the right of the window and the number of pins of the net to the left of the window is as close to 0 as possible identical (and correspondingly for the pins above and below the window). Figure 5.3 demonstrates what may happen in such a situation. The picture shows a net consisting of six pins (indicated by black cycles) that are connected to one Steiner node (the black square). The chip area is partitioned into 16 windows, and the pins are placed in the window they are assigned to. For the computation of the y-coordinate, the connections between the pins and the Steiner node are not split as they are not cut by a horizontal cutline. For the x-coordinates, three of the connections cross a cutline and have to be split. Now assume that we remove the rightmost pin p from the net. This would lead to a different position of the Steiner node in the QP solution, but, since the connection between pin p' and the Steiner node is split, it would not change the position of p' in the QP solution. With the clique net model, the position of p' would depend on the existence of p. As the connection between p and p' may be important, a reasonable net model should take the existence of p into consideration when the position of p' is computed.

In the old version of BonnPlace (Vygen [1997]), such drawbacks of the star net model made the usage of the clique model for almost all nets mandatory. Hence, by default only the very few nets with more than 20 terminals were represented by a star. Experiments provide evidence that using the star model also for smaller nets leads to worse placements. On the other hand, using clique for nets with up to 20 pins is quite inefficient since a net with 20 pins causes 190 non-zero entries in the matrix for the QP computation.

Therefore, we propose a new net model that is equilvalent to clique (i.e., the QP solutions for the two net models are identical) but for which the connectivity matrix in the QP formulation is almost as sparse as for star. As the new net model combines useful properties of star and clique, we call it *hybrid net model*. The net model is the sum of an x-netlength

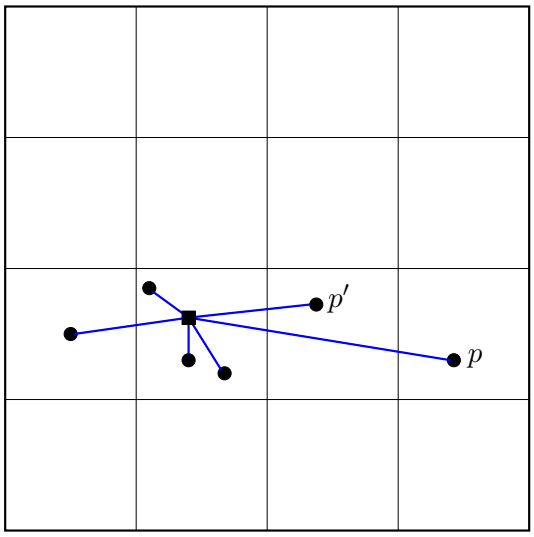


Figure 5.3: The star model with cutlines. The six black dots are the pins of a net, and the black square is the additional Steiner node.

and a y-netlength. We will describe the contribution of the x-netlength, the y-direction is handled correspondingly. As we only consider x-coordinates, we are only have to take vertical cutlines into consideration. Assume that the chip area $[0, W] \times [0, H]$ is partitioned by vertical cutlines at positions $x_0, x_1, \ldots, x_k$ where $x_0 = 0$ and $x_k = W$. Hence, the interval $[0, W]$ is parititoned into intervals $I_i := [x_i, x_{i+1}]$ (for $i \in \{0, \ldots, k-1\}$). Let $\mathcal{I} := \{I_0, \ldots, I_{k-1}\}$. For a net N and an interval $I \in \mathcal{I}$ let N_I be the set of pins in N whose x-coordinate has to stay in I. Let $n_0 \in \mathbb{N} \backslash \{0, 1, 2, 3\}$, and let $\mathcal{I}_{\text{small}} := \{I \in \mathcal{I} \mid 0 < |N_I| \leq n_0\}$ and $\mathcal{I}_{\text{big}} := \{I \in \mathcal{I} \mid |N_I| > n_0\}$. Then, we define

$$\text{HYBRID}_x^2(N, x_0, \ldots, x_k, n_0) :=$$
$$\frac{1}{|N|-1} \left[\sum_{i=0}^{k-1} \left(\sum_{j=0}^{i-1} |N_{I_j}| \sum_{p \in N_{I_i}} (x(p) - x_i)^2 + \sum_{j=i+1}^{k-1} |N_{I_j}| \sum_{p \in N_{I_i}} (x(p) - x_{i+1})^2 \right) + \sum_{I \in \mathcal{I}_{\text{small}}} \text{CLIQUE}_x^2(N_I) + \sum_{I \in \mathcal{I}_{\text{big}}} |N_I| \text{STAR}_x^2(N_I) \right]$$

where $\text{CLIQUE}_x^2(N_I)$ and $\text{STAR}_x^2(N_I)$ are the x-components of the quadratic clique and star net model for the net N_I. This means that we use for the subsets $N_I \subseteq N$ (for $I \in \mathcal{I}$) that are not divided by cutlines, either the clique net model (if N_I is small enough) or the star net model.

Figure 5.4 illustrates the hybrid net model for a net consisting of nine pins (blue dots). There are five vertical cutlines at coordinates $x_1, \ldots, x_4$ creating four intervals $I_0, \ldots, I_3$. Again, we focus on the x–coordinates only, so we do not consider horizontal cutlines. We have $|N_{I_3}| = 1$, $|N_{I_1}| = 5$, $|N_{I_2}| = 0$, and $|N_{I_3}| = 3$. In this example we assume $n_0 = 4$, hence $\mathcal{I}_{\text{small}} = \{I_0, I_2, I_3\}$ and $\mathcal{I}_{\text{big}} = \{I_1\}$. The artificial fixed pins at the coordinates of the cutlines are represented by red squares, and their connections to the pins of the net are shown as red lines. The pins in $I_1 = [x_1, x_2]$ are connected by a star (the black square with

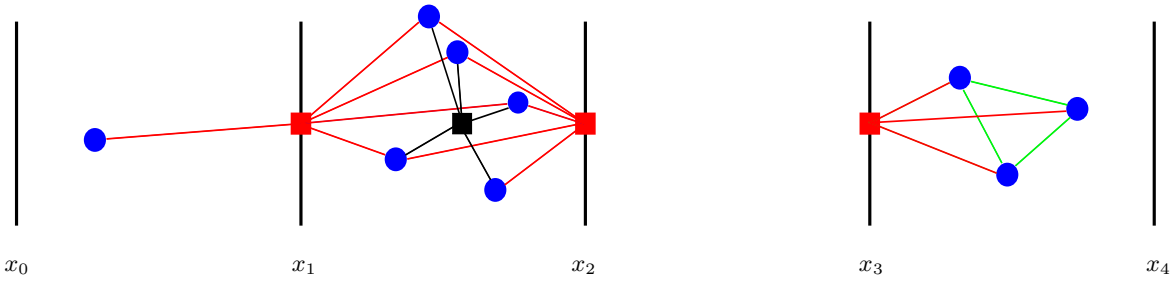


Figure 5.4: The hybrid net model for a net with nine pins.

the black edges). The three green lines are the edges of the clique connecting the pins in the interval $I_3 = [x_3, x_4]$.

The new net model has the following properties:

Lemma 5.1 *(a) The hybrid net model and the clique net model lead to the same QP solutions.*

(b) A net N causes at most $|N|(n_0+1)$ non-zero entries in the connectivity matrix.

(c) The contribution of a net N to the connectivity matrix can be computed in time $O(|N|(n_0 + \log k))$.

(d) The number of additional Steiner nodes is $|\mathcal{I}_{big}|$.

Proof:

(a) We have $|N_{I_i}|\textsc{Star}^2(N_{I_i}) = (|N_{I_i}|-1)\textsc{Clique}(N_{I_i})$. Using this equality, it is obvious that the hybrid net model and the clique model are equivalent.

(b) Each $p \in N$ is connected to at most $n_0 - 1$ nodes inside its interval and two fixed nodes at the borders of its interval.

(c) We can compute a list of all $I \in \mathcal{I}$ with $|N_I| > 0$ sorted by the left border of the intervals in time $O(|N| \log k)$. Using this list, we can compute all connections that are split and all connections to Steiner nodes easily in time $O(|N|)$. The connections for the local clique can be computed in time $\sum_{I \in \mathcal{I}_{small}} \frac{|N_I|(|N_I|-1)}{2} = O(|N|n_0)$.

(d) Trivial. □

The number n_0 is a threshold parameter: for subsets $N_I \subseteq N$ with $|N_I| \leq n_0$ we use clique and for larger subsets, we use the star model. Using the star model for subsets N_I with $|N_I| \leq 3$ is not reasonable because this would increase the number of Steiner nodes without decreasing the number of non-zero entries in the matrix. Hence, we demand $n_0 > 3$. The larger n_0 is, the more subsets will be represented by a clique, so the number of non-zero entries in the matrix grows with increasing values of n_0. On the other hand, the number of additional Steiner nodes is bigger if n_0 is small. In our experiments, we choose $n_0 := 8$.

5.3.2 Parallelization

The QPs that compute, e.g., the x-coordinates of circuits between two cutlines can be solved without considering the circuits outside the two cutlines. Once we have computed the set of circuits to be placed between each pair of cutlines, we can solve the global QP by solving k small equation systems instead of one big system (if we compute the x-coordinates and the chip area $[0, W] \times [0, H]$ is divided by vertical cutlines at coordinates $\{x_0 = 0, x_1, \ldots, x_k = W\}$). Of course, solving these smaller equation systems is generally much faster than solving the big equation system, and since the equation systems can be solved independently, they allow parallel computation. In addition, we can solve two single repartitioning steps in parallel if the corresponding areas do not intersection if we project them to the x- or y-axis. Our implementation contains both sorts of parallelization, and for our experiments, we run BonnPlace on up to 4 processors. Of course, it is possible to use more processors, but 4 processors provide a reasonable speedup (as our experiments will show). Only for very large instances, it may be worthwile to use more than 4 processors.

5.4 Experimental Results

In this section, we will examine some individual aspects of the features and the implementation of our global placement algorithm. A general study of our whole placement tool (with comparison to other tools) will be done in Chapter 9. All tests presented in this chapter were made on an IBM 680 with 600 MHz RS-IV processors. Note using our fastest available machines instead (Opteron machines with 2.6 GHz processors) would reduce all running times roughly by a factor of four, but here we are interested only in the relative performance of different algorithms.

5.4.1 Flow-based Partitioning vs. American Maps

In this subsection, we want to examine how the Transportation Flow Algorithm used for multisection performs on quadrisection instances. The theoretical running time is by a logarithmical factor bigger than the running time of the American-Map Algorithm. However, the experiments will show that the flow based partitioning algorithm is, in practice, even faster than the American-Map Algorithm.

Table 5.1 gives an overview of the results of our experiments with one single quadrisection step. We ran levels 0 and 1 of BonnPlace and measured the running time for the quadrisection in level 1, computed either with the American-Map Algorithm or with our Transportation Flow Algorithm. For one set of experiments, we used the smallest possible initial density on the chips (results are shown in columns two and three), and for a second set of experiments, we used 80% as initial density (columns four and five).

We also considered the total running time spent on quadrisection in a complete placement run. Note that the two algorithms used for quadrisection will normally produce slightly different solutions. Therefore, the two placement runs will differ a little bit and (from level two on) the two quadrisection algorithms will get different inputs. Even the number of

Chip	Low Density		High Density	
	American Map	Transportation Flow	American Map	Transportation Flow
Jens	0.9	1.4 +55.6 %	0.9	1.4 +55.6 %
Hans	1.6	1.4 -12.5 %	1.5	1.3 -13.3 %
Christian	14.1	7.0 -50.4 %	10.2	5.4 -47.1 %
James	23.8	10.3 -56.7 %	16.0	7.6 -47.5 %
Sven	31.9	21.8 -31.7 %	34.1	16.6 -51.3 %
Dagmar	51.5	22.8 -55.7 %	37.2	19.6 -47.3 %
Dieta	28.9	25.7 -11.1 %	23.8	20.6 -13.4 %
Sandra	48.4	35.7 -26.2 %	53.8	26.6 -50.6 %
Reinhardt	47.3	39.1 -17.3 %	43.4	36.5 -15.9 %
Nadine	150.1	44.3 -70.5 %	69.9	33.1 -52.6 %
Hardy	126.8	53.5 -57.8 %	131.4	50.0 -61.9 %
Wolf	111.5	78.9 -29.2 %	101.8	47.1 -53.7 %

Table 5.1: Comparison of the Transportation Flow Algorithm and the American-Map Algorithm: running time (in seconds) for the first quadrisection step.

quadrisection problems to be solved can vary. Table 5.2 shows the results: Columns two and three contain the total running time for all quadrisection steps in a placement run, computed with the American-Map Algorithm or with the Transportation Flow Agorithm. The average running time reduction shown in the last row is computed by the geometric mean of the ratios of the running times.

Of course, such experiments compare implementations of algorithms rather than the algorithms themselves. Nevertheless, we can draw the following conclusions from the experiments:

- The Transportation Flow Algorithm is faster on instances with bigger capacity in the subregions. This is not surprising, since with increasing capacity it is more likely for a circuit to stay in its initial region, and in this case the minimum cost flow computation in G_i is trivial.

Chip	American Map	Transportation Flow
Jens	0:01:31	0:01:21 -11.0 %
Hans	0:02:28	0:01:30 -39.2 %
Christian	0:10:25	0:07:15 -30.4 %
James	0:16:12	0:12:03 -25.6 %
Sven	0:30:51	0:22:13 -28.0 %
Dagmar	0:39:21	0:26:20 -33.1 %
Dieta	0:41:33	0:30:33 -26.5 %
Sandra	1:01:24	0:48:08 -21.6 %
Reinhardt	1:09:17	0:50:17 -27.4 %
Nadine	1:28:05	0:59:24 -32.6 %
Hardy	1:48:32	1:25:51 -20.9 %
Wolf	1:58:13	1:31:39 -22.5 %
Average		-26.9 %

Table 5.2: Comparison of the Transportation Flow Algorithm and the American-Map Algorithm: running time (h:mm:ss) of all quadrisection steps in a placement run.

- The running time of the implementation of the Transportation Flow Algorithm scales roughly linearly in the input size.
- The implementation of the Transportation Flow Algorithm is significantly faster than the implementation of the American-Map Algorithm on all input sizes and densities that we have tested.

One may assume that there are more efficient implementations of the American-Map Algorithm, but the experimental results show that the Transportation Flow Algorithm is fast enough to be used even on large instances. Therefore, we will always use the new algorithm to solve partition problems even in the case of quadrisection. Note that although partitioning is not the most critical part of placement in terms of running time,

the time for all partitionings in a placement run can sum up to 10% of the total running time.

5.4.2 Iterative Partitioning

We will examine how the amount of repartitioning can be reduced by using the concept of iterative partitioning. By experience, it is reasonable to run repartitioning steps at the end of a placement level as long as the last repartitioning step reduced the netlength by at least 1 %. With the improved strategy of iterative partitioning, it is sufficient to run a repartitioning step only if the previous step improved the netlength by at least 8 %, so normally only one repartitioning step is applied. Figure 5.5 shows why this can be done without losing anything in terms of netlength. The charts compare the netlength in a standard BonnPlace run to the netlength in an BonnPlace run with iterative partitioning. One can see that the increase of netlength caused by the initial partitioning in each level is much smaller with the new approach. Therefore, we can afford running a smaller number of repartitioning steps afterwards. Without the new partitioning algorithm, the reduction of the amount of repartitioning would increase significantly the netlength (by 5.8 % in this example).

The reduced number of repartitiong steps helps, of course, improving the running time. Table 5.3 compares two BonnPlace versions: The first one runs with standard partitioning and repartitioning, the second one applies iterative partitioning and reduced repartitioning. On average, the runtime decreases by 30.1 % and the netlength by 0.5 %. We have also run experiments with iterative partitioning but without reducing the number of repartitioning step hoping that this would improve the result. But it turned out that the additional repartitioning steps did not have an impact on the final netlength that would be worth the increase of running time.

5.4.3 Multisection Experiments

As a feature, BonnPlace can run 2×3-partitionings in the last levels in order to reduce the number of levels. We tested this method on our chips and compared it to the standard method. The results are shown in Table 5.4. Note that we used the same initial maximum density (70 %) in the standard runs and in the runs with reduced number of levels and increased the allowed density in each level by 1 %. Hence, in the runs with less levels, the maximum allowed density was somewhat smaller than it was at the end of the runs with the standard method. This explains, why the netlength is slightly bigger in the run with the modified version. If the density increment per level is chosen in such a way that the maximum allowed density at the end of the placement run is the same as in the run with the old version, then the new version yields almost the same result as the old version.

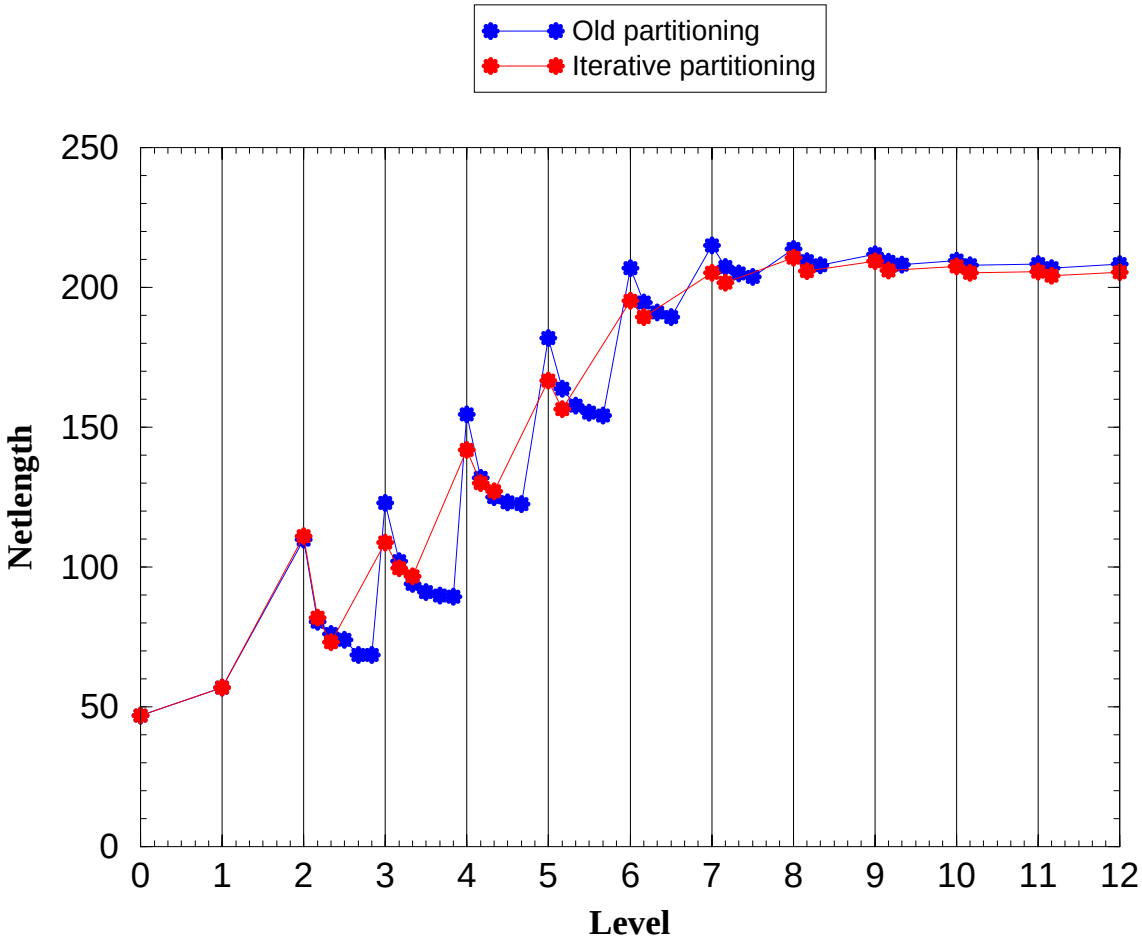


Figure 5.5: Netlength during placement runs on the chip Dieta. The blue line shows the netlength for a standard partitioning run and the red line for the same algorithm with the iterative partitioning strategy. The dots on the integer x-coordinates from 1 to 11 correspond to netlengths after the partitioning in the corresponding level. The additional dots on the fractional x-coordinates correspond to netlengths after the repartitioning steps. The dot on x-coordinate 12 is the netlength after legalization.

Chip	Old Partitioning		Iterative Partitioning	
	CPU time	Netlength	CPU time	Netlength
Jens	0:16:48	7.29 m	0:14:53 - 11.4 %	6.81 m - 6.6 %
Hans	0:19:42	7.64 m	0:13:26 - 31.8 %	7.55 m - 1.2 %
Christian	1:40:47	160.48 m	1:07:46 - 32.8 %	163.99 m + 2.2 %
James	2:27:01	108.98 m	1:49:01 - 25.8 %	109.57 m + 0.5 %
Sven	5:30:02	251.93 m	4:16:23 - 22.3 %	254.10 m + 0.9 %
Dagmar	4:52:54	182.94 m	3:24:28 - 30.2 %	183.53 m + 0.3 %
Dieta	8:08:56	208.18 m	5:17:10 - 35.1 %	205.42 m - 1.3 %
Sandra	8:31:16	381.78 m	5:57:40 -30.0 %	377.76 m - 1.1 %
Reinhardt	9:59:07	346.07 m	7:37:59 - 23.6 %	342.52 m - 1.0 %
Nadine	12:22:47	368.31 m	8:18:15 - 32.9 %	380.00 m + 3.2 %
Hardy	15:47:41	367.12 m	8:56:58 - 43.3 %	350.90 m - 4.4 %
Wolf	15:18:42	564.96 m	9:36:48 - 37.2 %	579.42 m + 2.6 %
Average			- 30.1 %	- 0.5 %

Table 5.3: Comparison of old partitioning and iterative partitioning.

Chip	Quadrisection only			Quadrisection and 2 × 3-partitioning		
	# Levels	Running Time	Netlength	# Levels	Running Time	Netlength
Jens	9	0:14:53	6.81 m	8	0:13:03 -12.3 %	6.84 m +0.4 %
Hans	9	0:13:26	7.55 m	9	0:13:26 + 0.0 %	7.55 m +0.0 %
Christian	11	1:07:46	163.99 m	9	0:53:01 -21.8 %	165.25 m +0.8 %
James	12	1:49:01	109.57 m	10	1:30:22 -17.1 %	110.51 m +0.9 %
Sven	12	4:16:34	254.10 m	10	3:37:37 -15.2 %	255.40 m +0.5 %
Dagmar	11	3:24:28	183.53 m	10	3:08:50 - 7.6 %	184.17 m +0.3 %
Dieta	11	5:17:10	205.42 m	10	4:48:09 - 9.1 %	206.49 m +0.5 %
Sandra	11	5:57:39	342.52 m	10	5:39:52 - 5.0 %	344.15 m +0.5 %
Reinhardt	11	7:37:58	377.76 m	10	7:21:31 - 3.6 %	379.63 m +0.5 %
Nadine	12	8:18:15	380.00 m	10	7:23:33 -11.0 %	383.87 m +1.0 %
Hardy	11	8:56:59	350.90 m	10	8:49:11 - 1.5 %	353.58 m +0.8 %
Wolf	12	9:36:48	579.42 m	10	8:36:26 -10.5 %	586.08 m +1.1 %
Average					- 9.8 %	+ 0.6 %

Table 5.4: Effect of 2 × 3-partitionings in the last levels: number of levels, total running time (h:mm:ss), and bounding-box netlength are shown.

Chapter 6

Detailed Placement

After global placement, the circuits are spread over the chip area and are assigned to small regions. In the partitioning steps, we try to meet the capacity constraints of the subwindows, but in the last levels, this will not always be possible. Hence, there will be some regions that are too full, but we may assume that there are no larger areas on the chip that contain too many circuits. Therefore, it will be possible to legalize the placement by local movements. In this chapter, we will describe a method that computes such a legal placement while minimizing the total movement of the circuits.

We assume that all macros have been fixed when legalization starts, so we only have to consider standard circuits. Formally, the problem we deal with is the following: We are given a rectangular chip image, partitioned into rows of standard height y_δ and a set of rectangular blockages (macros or user-defined blockages). Moreover, we are given a set of standard circuits, and a location for each of them on the chip area. The task is to place these objects on integral coordinates within the rows, disjointly from the blockages, and without any overlaps. The initial positions of the circuits can either be the output of a global placement or the result of a timing optimization on a placed chip. As we assume that this initial placement is already optimized (in terms of timing, routability, wirelength etc.), we try to legalize it while minimizing the (weighted) sum of the (squared) movement of each circuit. We will show that this legalization has minimum impact on other design objectives such as wirelength, routability and timing.

6.1 Overview of the Literature

Compared to global placement, the literature on detailed placement is quite sparse. Mostly greedy heuristics (Wang, Yang, and Sarrafzadeh [2000b], Khatkhate et al. [2004], Kahng and Wang [2004], Chan, Cong, and Sze [2005], and Chen et al. [2005]) and simulated annealing approaches (Sarrafzadeh, Wang [1997], see also Sechen [1998] and Wong, Leong, Liu [1988]) are applied. Some authors combine the detailed placement with postopt steps that change a given placement locally in order to improve netlength or similar objective functions (see, e.g., Hur and Lillis [2000], Wang, Yang, and Sarrafzadeh [2000b], and Faroe, Pisinger, and Zachariasen [2001]).

If all circuits have equal width, then the problem can be easily formulated as an assignment problem (assigning circuits to slots) or, more interestingly, as a transportation problem (transporting circuits from places with overlaps to free slots). It is natural to divide the chip area into regions and to formulate a minimum cost flow or a shortest path problem in a graph whose nodes correspond to the regions, even in the more realistic case where circuits have different widths. Such an approach has been followed by Doll, Johannes, and Sigl [1991], Vygen [1998], and Hur and Lillis [2000].

Doll, Johannes, and Sigl [1991] (see also Doll, Johannes, and Antreich [1994]) describe a combined global and detailed placement algorithm that iteratively tries to improve a given placement. This method divides the chip area into small regions and optimizes the placement in each region by solving a transportation problem. In order to allow for escaping from local optima, the algorithm works with overlapping regions.

In the detailed placement algorithm proposed by Vygen [1998], the chip area is divided into areas of height y_δ. If there are areas that contain more circuits than fit into it, a minimum cost flow problem is solved in order to decide between which areas circuits have to be moved. This step is repeated until there is no area that contains too many circuits. Then, each area can be legalized separately.

Hur and Lillis [2000] compute a shortest path between a region with an overload to a region with free capacity. In their shortest path instance, the cost of a directed edge between two regions is the minimal cost for moving a circuit from the head to the tail of the edge. By moving circuits along the edges of the path from one region to another, they iteratively reduce violations of the capacity constraints.

Only for very restricted subproblems of the detailed placement problem, optimal polynomial time algorithms are known: Hur and Lillis [2000], Kahng, Tucker, and Zelikovsky [1999], Brenner and Vygen [2000], and Garey, Tarjan, and Wilfong [1988] describe algorithms that can be applied to the problem of legalizing a placement in a single row with additional ordering constraints (see Section 6.7).

Recently, Ren et al. [2005] proposed a "circuit-diffusion algorithm" that does not end up with a legal placement but can be seen as a preprocessing step before legalization. It spreads the circuits in such a way that they can be legalized afterwards by a simple greedy strategy without causing significant additional movement. To compute the spreading, the authors divide the chip area into tiles and compute for each tile its local density, i.e., the total size of circuits in it divided by its capacity. Then, for each circuit, the density in its tile and the densities in the two horizontally neighbouring tiles are used to compute its horizontal movement. Correspondingly, vertical movements are computed, and all circuits are moved over a short distance according to these numbers. Similar to force-directed placement, the method is iterated untill a sufficient spreading is reached. The strategy does not seem to make sense for a legalization after global placement because then circuits should be spread well enough for legalization, but after a timing optimization it may happen that there are some hot spots where many additional buffers or inverters have been inserted. In order to move circuits away from these critical areas, some iterations of the circuit diffusion may be reasonable.

6.2 Preprocessing with Global Placement

For our legalization approach, we assume that circuits are distributed over the chip area such that all circuits can be legalized close to their given position. If the input to legalization is the result of a global placement this will be the case, but, as mentioned above, a timing optimization may lead to some areas that are much too full. Figure 6.1 (a) illustrates this for an ASIC with about 400 000 circuits. The picture shows a "balance table" of the chip after a timing optimization. This means that the colors indicate how full the areas are: green and blue colors mean that there is some free capacity while red and pink areas are too full. The hot pink areas have an overload of at least 20 %. Blocked parts of the chip are show in black. It should be clear that a legalization that only wants to apply local changes to the placement would have a hard time on this instance. In order to receive a reasonable legalization instance even in such cases, we propose the following strategy: before legalization, we run a global placement (or at least the last levels of it), but in this placement we connect each circuit by an artificial net to its initial position. If we put a weight of 1 on these artificial nets (which is the default value for all nets), the QP solution in level 0 will reproduce the given placement quite well, and the global placer will only change the placement in parts of the chip where the density constraints are violated. Figure 6.1 (b) shows the balance table for such a placement. We can see that there are only some small density violations and the legalization should be able to fix these. Hence, we always assume that the input of our legalizer corresponds to a balance table that is comparable to Figure 6.1 (b).

6.3 Our Approach

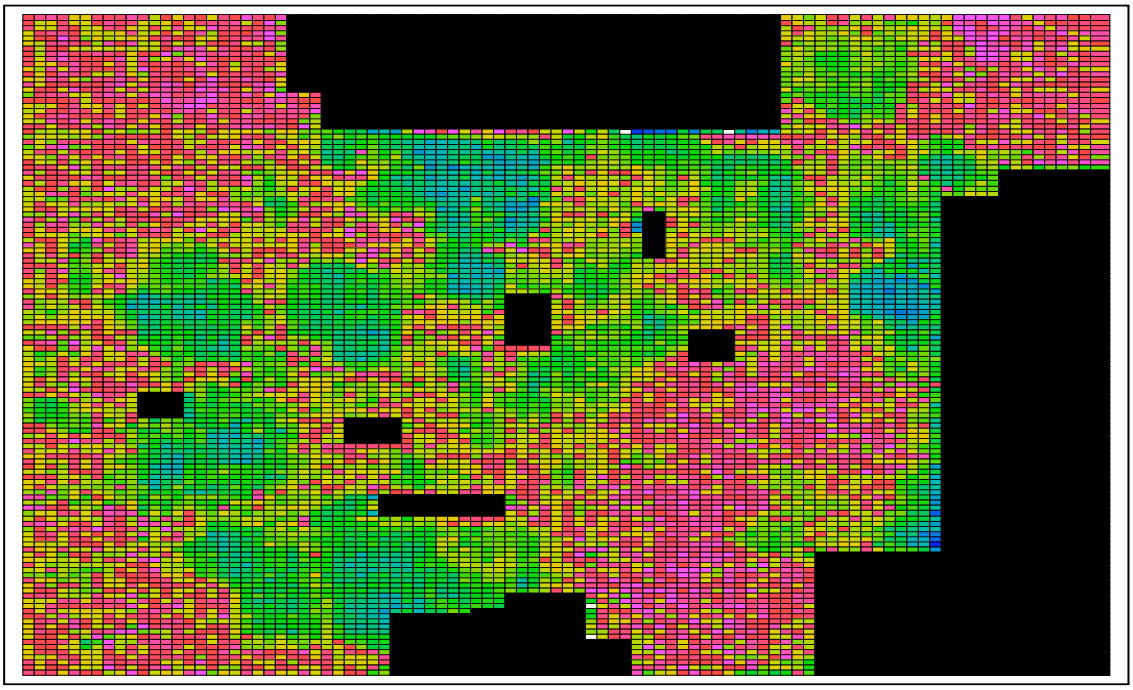
Our legalization algorithm consists of three phases. The first phase is based on a minimum cost flow computation that decides where circuits will be moved in order to reduce the overload in areas that are too full. After this first phase, circuits can be legalized within their row which is done in the second phase using known algorithms for optimal single-row placement. After phase two, the placement is legal. The third phase consists of a postoptimization routine that is able to reduce the largest movements of circuits during legalization. This is based on a shortest-path formulation that is solved by dynamic programming.

The following sections will give a detailed description of our legalization algorithm.

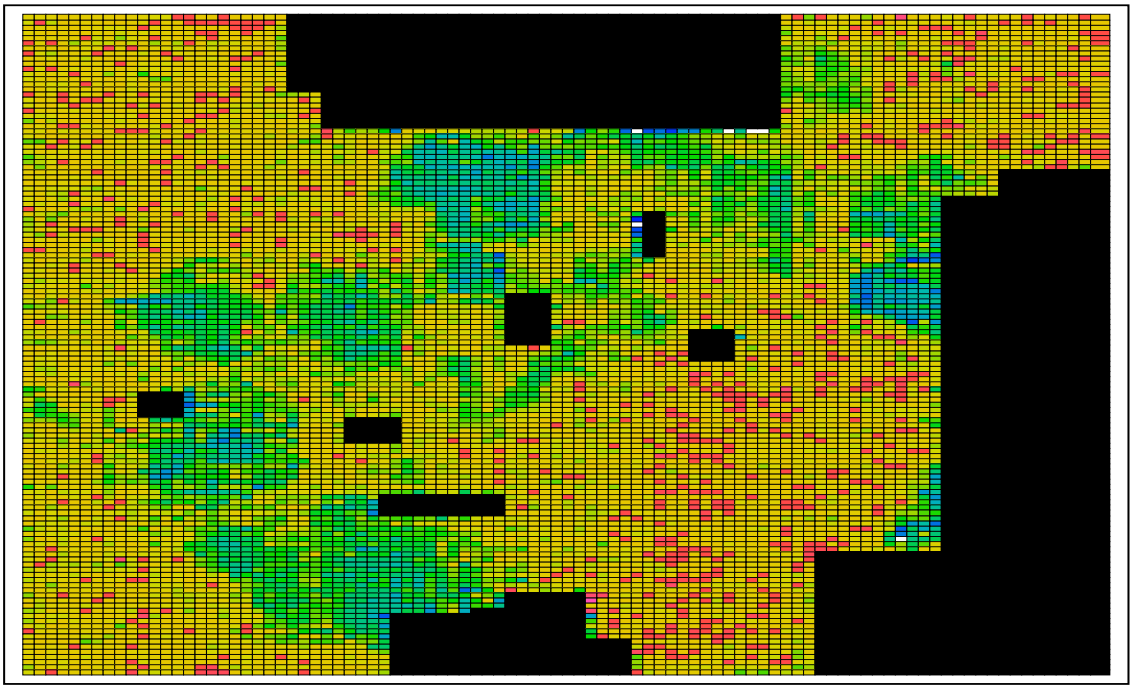
6.4 Minimum Cost Flow Formulation

The chip area is partitioned into columns of equal size, and at least for the first phase of our algorithm, we consider all circuits wider than twice the width of the columns to be fixed at their position.

We call a maximal part of a row that is either completely free (i.e., it does not intersect any blockage) or completely blocked a *zone*. A non-empty intersection of a column and a zone is called a *region*. (see Figure 6.2). The task is to place the circuits disjointly in the



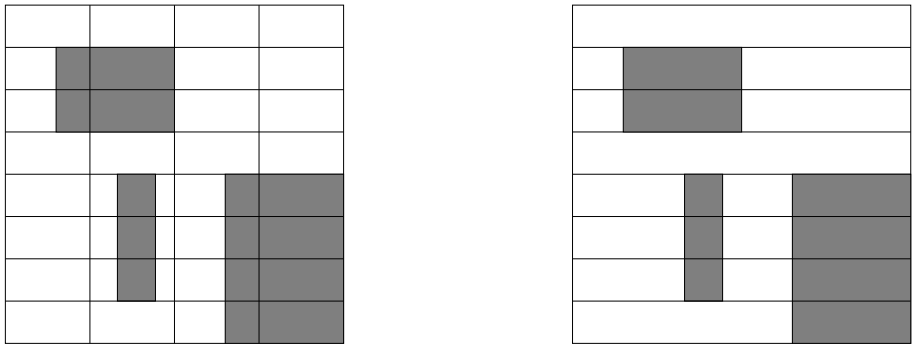
(a)



(b)

Figure 6.1: Balance tables for a placement after timing optimization (a) and after some global placement levels on this placement (b).

free zones.



(a) (b)

Figure 6.2: The placement area of a chip consisting of eight rows with three preplaced macros (grey). In (a) the 44 regions are shown, picture (b) shows the 22 zones.

The first two phases of our algorithm work as follows. First, we move circuits between zones such that no zone contains more circuits than fit into it. In a second step, we compute feasible locations for each circuit in the zone it is assigned to.

However, if we just follow this strategy, movement within the zones is not considered at all in the first phase, and therefore large movements within wide zones may be necessary in the second phase. Figure 6.3 shows five circuits that are placed in one of two zones. If we only considered zones, the assignment of all circuits to the lower zone would be feasible (and cheapest) but in order to minimize movement, two of the circuits should leave their zone. Generally spoken, we have to assign the circuits to the zones in such a way that they can be placed there without being moved too far within the zone. Experiments clearly demonstrated that ignoring this horizontal movement within the zones during the first phase yields poor results.

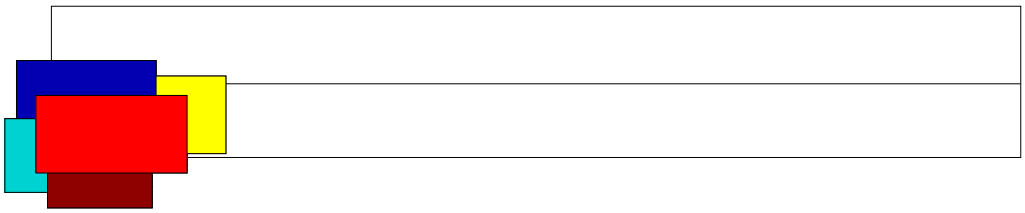


Figure 6.3: Two zones of a chips containing five circuits. The circuits could be placed in the lower zone, but if two of them were moved to the upper zone, the total movement could be reduced.

To cope with this problem, we start by using regions rather than zones and allow circuits to belong to two horizontally adjacent regions. If necessary, we iterate this with increasing region widths and may finally use even complete zones. But then only relatively few circuits have to be moved before we have a feasible assignment to the zones.

We fix a global order of the circuits that is induced by the left-to-right-order of the x-coordinates of their centers before the legalization starts. For the rest of this chapter let

$\{A_1, \ldots, A_l\}$ be a set of unblocked regions that form a zone (ordered from left to right). Let $C^i = \{c^i_1, \ldots, c^i_{k_i}\}$ be the set of circuits that are assigned to region A_i, ordered with respect to the fixed global order (for $i \in \{1, \ldots, l\}$). The width of region A_i is denoted by $w(A_i)$, and, as usual, the width of a circuit c^i_j by $w(c^i_j)$. The width of a set of regions or circuits is the sum of the single widths. Let $C = C^1 \cup \cdots \cup C^l$.

Previous approaches required each set $\{c^i_1, \ldots, c^i_{k_i}\}$ to be placed completely within the region r_i. However, this causes a lot of unnecessary movement because the subdivision of the zones into regions is only an artificial construction for the algorithm and there is no reason why circuits should not overlap two neighbouring regions. Therefore, we only require that the *center* of each circuit c^i_j is placed within A_i (for $i = 1, \ldots, l$, $j = 1, \ldots, k_i$), and that there are no overlaps of circuits. We will call such a placement *legal*. What we are looking for is a legal placement that respects our given left-to-right order. We will call an assignment of the circuits to the regions a *feasible assignment* if there is a legal placement with respect to that assignment and the given left-to-right order. In the first part of the legalization, we search for such a feasible assignment.

To decide if an assignment of circuits is feasible (and, in the case that it is not feasible, to decide how to make it feasible) we cannot consider the regions separately but we have to consider sets of consecutive regions. We will call such a set of consecutive regions an *interval*. We denote the interval that consists of the regions $A_\mu, A_{\mu+1}, \ldots, A_{\nu-1}, A_\nu$ as $A_{\mu,\nu}$ (for $1 \le \mu \le \nu \le l$).

The vertex set of the graph G that we construct for the minimum cost flow instance consists of all regions and some intervals. We have so-called *supply intervals* which are too full and *demand intervals* which contain some free space.

To simplify the notation, we may assume w.l.o.g. that for each zone $\{A_1, \ldots, A_l\}$ we have $w(c^1_1) = w(c^l_{k_l}) = 0$ and that there are two additional regions A_0 and A_{l+1} with $w(A_0) = w(A_{l+1}) = 0$, $k_0 = k_{l+1} = 1$, and $w(c^0_1) = w(c^{l+1}_1) = 0$.

6.4.1 The Supply Nodes

If $w(C^i)$ $(:= \sum_{j=1}^{k_i} w(c^i_j))$ is greater than $w(A_i)$ for an $i \in \{1, \ldots, l\}$, then, of course, the circuits in C^i do not fit completely into A_i. But if $w(C^i) - \frac{1}{2}\left(w(c^i_1) + w(c^i_{k_i})\right) \le w(A_i)$, then there is still a chance to find a feasible placement, provided that there is enough free capacity in the regions that are horizontal neighbours of A_i.

To compute the size of circuits that have to be removed from an interval $A_{\mu,\nu}$ we define for $1 \le \mu \le \nu \le l$:

$$s_{\mu,\nu} := \max\left\{\sum_{i=\mu}^{\nu}\left(w(C^i) - w(A_i)\right) - \frac{1}{2}\left(w(c^\mu_1) + w(c^\nu_{k_\nu})\right),\quad 0\right\}.$$

We will show that if we have $s_{\mu,\nu} = 0$ for all μ, ν with $1 \le \mu \le \nu \le l$, then all circuits in the zone can be placed feasibly.

For $1 \leq \mu \leq \nu \leq l$ we define recursively:

$$\text{supp}(A_{\mu,\nu}) := \max \left\{ s_{\mu,\nu} - \sum_{\substack{\mu \leq \mu' \leq \nu' \leq \nu \\ (\mu,\nu) \neq (\mu',\nu')}} \text{supp}(A_{\mu',\nu'}), \quad 0 \right\}.$$

For $1 \leq \nu < \mu \leq l$, it is convenient to define $s_{\mu,\nu} := \text{supp}(A_{\mu,\nu}) := 0$. An interval $A_{\mu,\nu}$ (for $\mu, \nu \in \{1, \ldots, l\}$) is called a *supply interval* if $\text{supp}(A_{\mu,\nu}) > 0$.

Figure 6.4 shows an example of a zone that consists of three regions A_1, A_2, and A_3 each of which has width 15. Obviously, the placement is not legal. We could place the circuits of each single region legally within their region (without considering the other regions) but we cannot do this simultaneously for all regions or for at least two neighbouring regions. The supply numbers are: $\text{supp}(A_{1,1}) = \text{supp}(A_{2,2}) = \text{supp}(A_{3,3}) = 0$, $\text{supp}(A_{1,2}) = \text{supp}(A_{2,3}) = 1$, $\text{supp}(A_{1,3}) = 4 - 1 - 1 = 2$. The interpretation of these numbers is that circuits of total size 4 have to be removed from the complete zone, but that (among these cells) circuits of size of at least 1 have to be removed from $A_{1,2}$ and from $A_{2,3}$, respectively. So, for example, removing the circuit c_2^2 of width 5 from region A_2 would be sufficient, while removing a circuit of width 4 from region A_1 or A_2 would not lead to a feasible assignment. With similar examples it can be shown that supply intervals can consist of arbitrarily many regions.

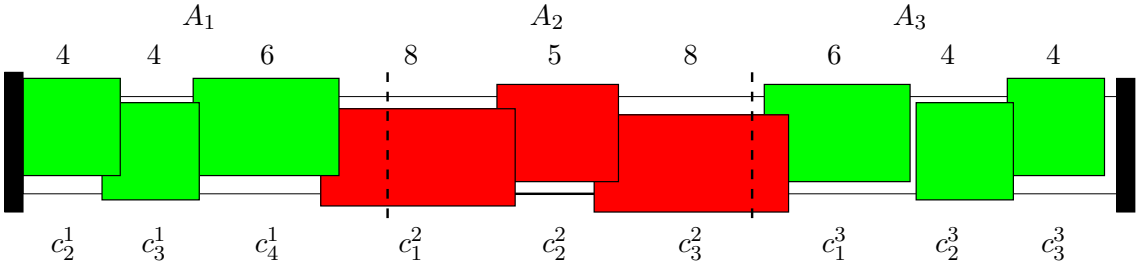


Figure 6.4: A supply interval consisting of three regions of width 15. The numbers are the widths of the circuits. The dummy circuits c_1^0, c_1^1, c_4^3, and c_1^4 of width 0 are not shown.

6.4.2 The Demand Nodes

To compute the free space in an interval $A_{\mu,\nu}$ we also have to consider the neighbouring regions of $A_{\mu,\nu}$ ($1 \leq \mu \leq \nu \leq l$). For two numbers μ, ν with $1 \leq \mu \leq \nu \leq l$ we define:

$$t_{\mu,\nu} := \min \left\{ \sum_{i=\mu}^{\nu} \left(w(\mathcal{C}^i) - w(A_i) \right) + \frac{1}{2} \left(w(C_{k_{\mu-1}}^{\mu-1}) + w(C_1^{\nu+1}) \right), \quad 0 \right\}.$$

For $1 \leq \mu \leq \nu \leq l$ we define recursively:

$$\text{dem}(A_{\mu,\nu}) := \min \left\{ t_{\mu,\nu} - \sum_{\substack{\mu \leq \mu' \leq \nu' \leq \nu \\ (\mu,\nu) \neq (\mu',\nu')}} \text{dem}(A_{\mu',\nu'}), \quad 0 \right\}.$$

For $1 \leq \nu < \mu \leq l$ we define $t_{\mu,\nu} = \text{dem}(A_{\mu.\nu}) = 0$. We call an interval $A_{\mu,\nu}$ with $\text{dem}(A_{\mu,\nu}) < 0$ a *demand interval*.

Figure 6.5 shows an example of a zone that consists of three regions A_1, A_2, and A_3. Again, each region has width 15. In this example, we get the demand values $\text{dem}(A_{1,1}) = \text{dem}(A_{2,2}) = \text{dem}(A_{3,3}) = \text{dem}(A_{1,2}) = \text{dem}(A_{2,3}) = 0$, and $\text{dem}(A_{1,3}) = -4$. Note that region A_2 is part of a demand interval although it contains circuits with larger total size than region A_1 in Figure 6.4 which is part of a supply interval. As for the supply intervals, there are similar examples that show that there can be arbitrarily large demand intervals.

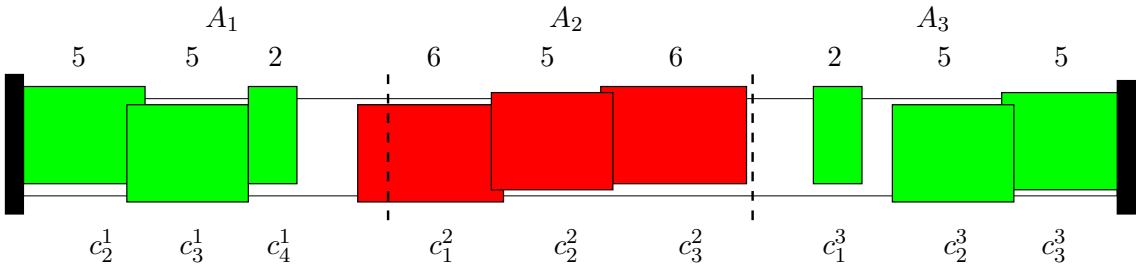


Figure 6.5: A demand interval consisting of three regions of width 15. The numbers are the widths of the circuits. Again, the dummy circuits c_1^0, c_1^1, c_4^3, and c_1^4 of width 0 are not shown.

The following lemma is important for the analysis of our legalization algorithm. We have to ensure that during the flow realization (to be explained in Section 6.5) for an interval $A_{\mu,\nu}$ $(1 \leq \mu \leq \nu \leq l)$ circuits of a total size $\text{supp}(A_{\mu,\nu})$ are removed from the interval (if $A_{\mu,\nu}$ is a supply interval) or circuits of a total size at most $-\text{dem}(A_{\mu,\nu})$ are added to the interval (if $A_{\mu,\nu}$ is a demand interval).

Lemma 6.1 *No region can be both part of a demand interval and part of a supply interval.*

Proof: Let $A_{\mu,\nu}$ be a demand interval and let $A_{\kappa,\lambda}$ be a supply interval. Assume that $A_{\mu,\nu}$ and $A_{\kappa,\lambda}$ intersect, i.e., that $\kappa \leq \nu$ and $\mu \leq \lambda$.

Case 1: $\kappa \leq \mu \leq \nu \leq \lambda$.
Using the definition of $\text{supp}(A_{\kappa,\mu-1})$ we get

$$\sum_{i=\kappa}^{\mu-1} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\left(c_1^{\kappa}\right) + w\left(c_{k_{\mu-1}}^{\mu-1}\right)\right) \leq \sum_{\kappa \leq \kappa' \leq \mu' \leq \mu-1} \text{supp}(A_{\kappa',\mu'}), \tag{6.1}$$

and the definition of $\text{supp}(A_{\nu+1,\lambda})$ yields

$$\sum_{i=\nu+1}^{\lambda} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\left(c_1^{\nu+1}\right) + w\left(c_{k_\lambda}^{\lambda}\right)\right) \leq \sum_{\nu+1 \leq \nu' \leq \lambda' \leq \lambda} \text{supp}(A_{\nu',\lambda'}). \tag{6.2}$$

Since $A_{\mu,\nu}$ is a demand interval we also have $t_{\mu,\nu} < 0$, so

$$\sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\left(c_{k_{\mu-1}}^{\mu-1}\right) + w\left(c_1^{\nu+1}\right)\right) < 0. \tag{6.3}$$

Summing up inequalities (6.1), (6.2), and (6.3) and using the fact that $s_{\kappa,\lambda} > 0$, we can conclude

$$s_{\kappa,\lambda} = \sum_{i=\kappa}^{\lambda} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^{\kappa}\Big) + w\left(c_{k_\lambda}^{\lambda}\right)\right) < \sum_{\substack{\kappa \le \kappa' \le \lambda' \le \lambda \\ (\kappa,\lambda) \neq (\kappa',\lambda')}} \operatorname{supp}(A_{\kappa',\lambda'}),$$

so we have $\operatorname{supp}(A_{\kappa,\lambda}) = 0$, which is a contradiction to the assumption that $A_{\kappa,\lambda}$ is a supply interval.

Case 2: $\mu \le \kappa \le \lambda \le \nu$. This case is very similar to case 1:
Using the definition of the demand values, we have

$$\sum_{i=\mu}^{\kappa-1} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\left(c_{k_{\mu-1}}^{\mu-1}\right) + w\Big(c_1^{\kappa}\Big)\right) \ge \sum_{\mu \le \mu' \le \kappa' \le \kappa-1} \operatorname{dem}(A_{\mu',\kappa'}) \tag{6.4}$$

and

$$\sum_{i=\lambda+1}^{\nu} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\left(c_{k_\lambda}^{\lambda}\right) + w\Big(c_1^{\nu+1}\Big)\right) \ge \sum_{\lambda+1 \le \lambda' \le \nu' \le \nu} \operatorname{dem}(A_{\lambda',\nu'}). \tag{6.5}$$

Since $A_{\kappa,\lambda}$ is a supply interval we also have

$$\sum_{i=\kappa}^{\lambda} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^{\kappa}\Big) + w\left(c_{k_\lambda}^{\lambda}\right)\right) > 0. \tag{6.6}$$

We sum up equations (6.4), (6.5), and (6.6) and use $t_{\mu,\nu} < 0$:

$$t_{\mu,\nu} = \sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\left(c_{k_{\mu-1}}^{\mu-1}\right) + w\Big(c_1^{\nu+1}\Big)\right) > \sum_{\substack{\mu \le \mu' \le \nu' \le \nu \\ (\mu,\nu) \neq (\mu',\nu')}} \operatorname{dem}(A_{\mu',\nu'}),$$

so we have $\operatorname{dem}(A_{\kappa,\lambda}) = 0$, again a contradiction.

Case 3: $\kappa < \mu \le \lambda < \nu$.
We have

$$\operatorname{supp}(A_{\kappa,\mu-1}) \ge s_{\kappa,\mu-1} - \sum_{\substack{\kappa \le \kappa' \le \mu' \le \mu-1 \\ (\kappa,\mu-1) \neq (\kappa',\mu')}} \operatorname{supp}(A_{\kappa',\mu'}).$$

Since $A_{\kappa,\lambda}$ is a supply interval and $A_{\mu,\nu}$ is a demand interval we can conclude:

$$\begin{aligned}
\operatorname{supp}(A_{\kappa,\lambda}) &= s_{\kappa,\lambda} - \sum_{\substack{\kappa\le\kappa'\le\lambda'\le\lambda \\ (\kappa',\lambda')\neq(\kappa,\lambda)}} \operatorname{supp}(A_{\kappa',\lambda'}) \\
&\le s_{\kappa,\lambda} - \sum_{\kappa\le\kappa'\le\mu'\le\mu-1} \operatorname{supp}(A_{\kappa',\mu'}) \\
&\le s_{\kappa,\lambda} - s_{\kappa,\mu-1} \\
&\le \sum_{i=\kappa}^{\lambda} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\left(c_1^{\kappa}\right) + w\left(c_{k_\lambda}^{\lambda}\right)\right) \\
&\quad - \sum_{i=\kappa}^{\mu-1} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\left(c_1^{\kappa}\right) + w\left(c_{k_{\mu-1}}^{\mu-1}\right)\right) \\
&= \sum_{i=\mu}^{\lambda} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\left(c_{k_{\mu-1}}^{\mu-1}\right) - w\left(c_{k_\lambda}^{\lambda}\right)\right) \\
&= \sum_{i=\mu}^{\lambda} \big(w(C^i) - w(A_i)\big) + \sum_{i=\lambda+1}^{\nu} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\left(c_{k_{\mu-1}}^{\mu-1}\right) + w\left(c_1^{\nu+1}\right)\right) \\
&\quad - \sum_{i=\lambda+1}^{\nu} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\left(c_{k_\lambda}^{\lambda}\right) + w\left(c_1^{\nu+1}\right)\right) \\
&\le \sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\left(c_{k_{\mu-1}}^{\mu-1}\right) + w\left(c_1^{\nu+1}\right)\right) \\
&\quad - \operatorname{dem}(A_{\lambda+1,\nu}) - \sum_{\substack{\lambda+1\le\lambda'\le\nu'\le\nu \\ (\lambda',\nu')\neq(\lambda+1,\nu)}} \operatorname{dem}(A_{\lambda',\nu'}) \\
&\le t_{\mu,\nu} - \sum_{\substack{\mu\le\mu'\le\nu'\le\nu \\ (\mu',\nu')\neq(\mu,\nu)}} \operatorname{dem}(A_{\mu',\nu'}) \\
&= \operatorname{dem}(A_{\mu,\nu}).
\end{aligned}$$

This is, of course, a contradiction.

Case 4: $\mu < \kappa \le \nu < \lambda$.
This case is symmetric to Case 3 and can be handled correspondingly. □

6.4.3 The Minimum Cost Flow Problem

To move circuits from supply nodes to demand nodes, we set up a minimum cost flow problem. We construct a graph G with vertex set $V(G)$ that contains of all regions and the supply and demand intervals. In addition, it is convenient to insert a super-source s and a super-sink t. Graph vertices that are regions are called *region vertices* or *region nodes* and vertices that are intervals are called *interval vertices* or *interval nodes*.

Two region vertices are connected by a pair of two edges with opposite direcction of infinite capacity if the regions are adjacent. For an edge (A, A') between two region nodes, the

number cost$((A, A'))$ should denote $\frac{1}{\gamma}$ times the cost for moving circuits of total width γ from A to A' for any $\gamma \in \mathbb{N}$. Naturally, such edge costs do not exist, and we have to work with rough estimates (for example by considering the cost of moving the three cheapest circuits, divided by their total width, and some extra positive cost for the case when there are less than three circuits in zone A). We may introduce parallel edges with finite capacity and different costs to partially overcome this difficulty. Usually the edge costs will be nonnegative, but when we consider objectives other than minimizing movement, some moves may be profitable. However, we require that every directed cycle in G has non-negative total cost.

The node s is connected to each supply node A by an edge (s, A) of zero cost and capacity supp(A). The node t is connected to each demand node A by an edge (A, t) of zero cost and capacity $-\text{dem}(A)$. If A' is a supply node and A is in $V(G) \setminus \{s, t, A'\}$ such that A is contained (as a region node or a subinterval) in A', then we insert an edge from A' to A of infinite capacity and zero cost. Similarly, if A' is a demand node and A is in $V(G)\setminus\{s, t, A'\}$ such that A is contained in A' then we insert an edge from A to A' of infinite capacity and zero cost. We also add edges of infinite capacity and zero cost from t to each region or interval node of the graph. Figure 6.6 illustrates this contruction on a small example (without the edges leaving t).

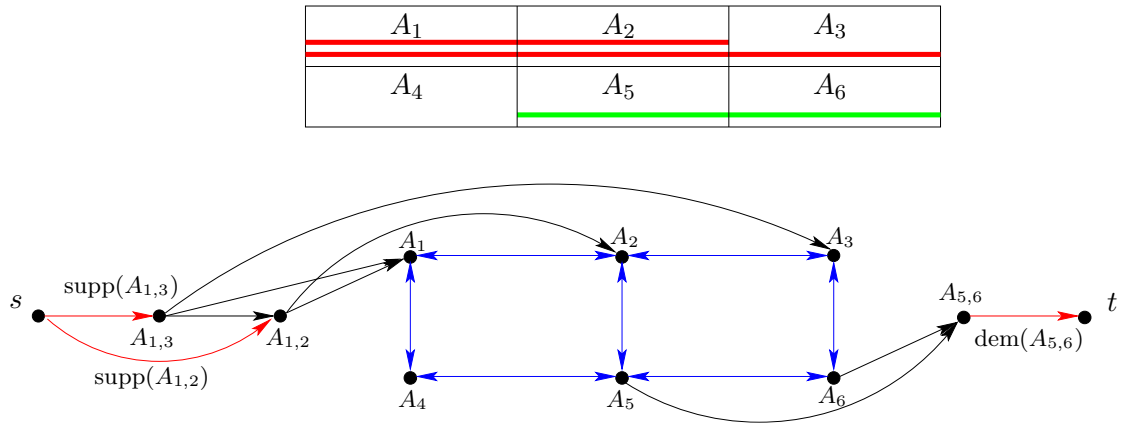


Figure 6.6: Minimum cost flow instance for a chip area consisting of six regions. Intervals $A_{1,2}$ and $A_{1,3}$ are supply intervals (marked by red lines), interval $A_{5,6}$ is a demand interval (marked by the green line). The edges leaving the node t are not shown. The numbers on the red edges are the capacities, the black and the blue edges have infinite capacity. The costs on the red and the black edges are 0, while the costs on the blue edges estimate the cost for moving a circuit of width 1 between the two corresponding regions.

We can assume that the sum of all supply values is less than or equal to the absolute value of the sum of all demand numbers. This assumption is reasonable, since otherwise the total size of all circuits was larger than the size of the non-blocked area. With this assumption, we have for any maximum s-t-flow f in the flow network that $f((s, A)) = \text{supp}(A)$ for each supply node A.

What we are looking for is a maximum s-t-flow f of minimum cost. With the algorithms described by Orlin [1993] or Vygen [2002b] such a flow can be computed in time

$O(m \log m(m + n \log n))$ where n is the number of nodes and m the number of edges. In our implementation, we apply a network simplex algorithm for the computation of the flow. It is not the fastest minimum cost flow algorithm in terms of worst-case running time (Armstrong and Jin [1997] show how it can be implemented in to run in time $O(mn(m + n \log n) \log n)$), but it performs very well on our instances. Nevertheless, if the flow computation turns out to be too slow on larger instances, one can partition the chip area and the corresponding graph into smaller parts and solve the minimum cost flow problem on these parts. However, our experiments show that this is not necessary for current VLSI instances.

The result of the minimum cost flow algorithm determines where circuits should be moved. If A and A' are region nodes and there is a flow of size $f((A, A'))$ on edge (A, A'), then circuits of a total size of $f((A, A'))$ should be moved from zone A to zone A'. We call such a movement an *(exact) realization* of f. We shall describe our flow realization method in Section 6.5.

Observe that none of the edges leaving t will be used by an optimum flow if all edges have positive cost. But even in this case they have a meaning. As we cannot always realize a given flow exactly, we can try to realize another maximum s-t-flow which has slightly higher cost than optimum. This will usually require the use of some edges leaving t.

In order to analyse the effect of a realization of the flow f, the following lemma will be useful:

Lemma 6.2 *(a) Let $\mu, \nu \in \{1, \ldots, l\}$ be indices such that $A_{\mu,\nu}$ is the union of supply intervals and every supply intervals that intersects $A_{\mu,\nu}$ is contained in $A_{\mu,\nu}$. Then, we have*

$$\sum_{\mu \le \mu' \le \nu' \le \nu} \operatorname{supp}(A_{\mu',\nu'}) = \sum_{i=\mu}^{\nu} (w(C^i) - w(A_i)) - \frac{1}{2}\left(w\left(c_1^{\mu}\right) + w\left(c_{k_\nu}^{\nu}\right)\right).$$

(b) Let $\mu, \nu \in \{1, \ldots, l\}$ indices such that $A_{\mu,\nu}$ is the union of demand intervals and every demand intervals that intersects $A_{\mu,\nu}$ is contained in $A_{\mu,\nu}$. Then, we have

$$\sum_{\mu \le \mu' \le \nu' \le \nu} \operatorname{dem}(A_{\mu',\nu'}) = \sum_{i=\mu}^{\nu} (w(C^i) - w(A_i)) + \frac{1}{2}\left(w\left(c_{k_{\mu 1-1}}^{\mu-1}\right) + w\left(c_1^{\nu+1}\right)\right).$$

Proof:

(a) By definition of $\operatorname{supp}(A_{\mu,\nu})$ we have

$$\sum_{\mu \le \mu' \le \nu' \le \nu} \operatorname{supp}(A_{\mu',\nu'}) \ge \sum_{i=\mu}^{\nu} (w(C^i) - w(A_i)) - \frac{1}{2}\left(w\left(c_1^{\mu}\right) + w\left(c_{k_\nu}^{\nu}\right)\right),$$

and if $A_{\mu,\nu}$ is a supply interval, we have equality. So, let us assume that $A_{\mu,\nu}$ is

not a supply interval. Let $A_{\mu_1,\nu_1},\ldots,A_{\mu_\rho,\nu_\rho}$ be a set of ρ supply intervals such that $A_{\mu,\nu}=\bigcup_{j=1}^{\rho}A_{\mu_j,\nu_j}$ and such that no interval A_{μ_j,ν_j} is contained in a larger supply interval (for $j=1,\ldots,\rho$). Let the intervals A_{μ_j,ν_j} be sorted such that $\mu_j<\mu_{j+1}$ and $\nu_j<\nu_{j+1}$ for $j=1,\ldots,\rho$. Then, we have

$$\begin{aligned}
\sum_{\mu\le\mu'\le\nu'\le\nu}\operatorname{supp}(A_{\mu',\nu'}) &= \sum_{j=1}^{\rho}\sum_{\mu_j\le\mu'\le\nu'\le\nu_j}\operatorname{supp}(A_{\mu',\nu'})-\sum_{j=2}^{\rho}\sum_{\mu_j\le\mu'\le\nu'\le\nu_{j-1}}\operatorname{supp}(A_{\mu',\nu'})\\
&\le \sum_{j=1}^{\rho}s_{\mu_j,\nu_j}-\sum_{j=2}^{\rho}s_{\mu_j,\nu_{j-1}}\\
&\le \sum_{j=1}^{\rho}\left(\sum_{i=\mu_j}^{\nu_j}\big(w(C^i)-w(A_i)\big)-\frac{1}{2}\left(w\Big(c_1^{\mu_j}\Big)+w\Big(c_{k_{\nu_j}}^{\nu_j}\Big)\right)\right)\\
&\quad -\sum_{j=2}^{\rho}\left(\sum_{i=\mu_j}^{\nu_j-1}\big(w(C^i)-w(A_i)\big)-\frac{1}{2}\left(w\Big(c_1^{\mu_j}\Big)+w\Big(c_{k_{\nu_{j-1}}}^{\nu_{j-1}}\Big)\right)\right)\\
&= \sum_{j=\mu}^{\nu}\big(w(C^i)-w(A_i)\big)-\frac{1}{2}\left(w\Big(c_1^{\mu}\Big)+w\Big(c_{k_\nu}^{\nu}\Big)\right).
\end{aligned}$$

(b) The equality can be proven analogously to part (a). □

Now, we will examine the effect of a realization of a minimum cost flow in G. Whether removing a set C' of circuits from a supply interval A will lead to a feasible assignment, does, in general, not only depend on the total size of the removed circuits, but also on the choice of the circuits. This is because the leftmost or the rightmost circuit of a region may be removed. We call a movement that does not change the leftmost and the rightmost circuit of each non-empty region an *interior movement*.

If we can realize the flow with interior movements, then we will get a feasible assignment:

Theorem 6.3 *Let f be a solution of the above-described minimum cost flow problem. Then, the result of an interior movement that realizes exactly the flow values of f yields a feasible assignment of the circuits.*

Proof: Let $\{A_1,\ldots,A_l\}$ be a set of unblocked regions that form a zone. To prove the claim of the theorem it is sufficient to show that for each interval $A_{\mu,\nu}$ in the zone we have the inequality

$$\sum_{i=\mu}^{\nu}\big(w(C^i)-w(A_i)\big)\le\sum_{\mu\le\mu'\le\nu'\le\nu}\operatorname{supp}(A_{\mu',\nu'})+\sum_{A_{\mu',\nu'}\cap A_{\mu,\nu}\neq\emptyset}\operatorname{dem}(A_{\mu',\nu'})+\frac{1}{2}\left(w\Big(c_1^{\mu}\Big)+w\Big(c_{k_\nu}^{\nu}\Big)\right)\quad(6.7)$$

Let $1=\mu_1\le\nu_1\le\mu_2\le\nu_2\le\cdots\le\mu_\rho\le\nu_\rho=l$ be indices with $\mu_{j+1}=\nu_j+1$ (for $j=1,\ldots,\rho-1$) such that each A_{μ_j,ν_j} either does not intersect any demand interval (then we call A_{μ_j,ν_j} an interval of type 1) or is the union of demand intervals (then we call A_{μ_j,ν_j}

an interval of type 2). Let ρ be as small as possible with this property, so if A_{μ_j,ν_j} is the union of demand intervals, then it contains all demand intervals that intersect it.

Let $A_{\mu,\nu}$ be an interval in the zone. We will show that (6.7) is valid for $A_{\mu,\nu}$. If $A_{\mu,\nu}$ is contained in an interval A_{μ_j,ν_j} of type 1, inequality (6.7) follows from the definition of the supply values. If $A_{\mu,\nu}$ is contained in an interval A_{μ_j,ν_j} of type 2, we can apply Lemma 6.2 (b) to A_{μ_j,ν_j} and have:

$$\begin{aligned}
\sum_{\substack{\mu',\nu':\\ A_{\mu',\nu'}\cap A_{\mu,\nu}\neq\emptyset}} \operatorname{dem}(A_{\mu',\nu'}) &= \sum_{\mu_j\le\mu'\le\nu'\le\nu_j} \operatorname{dem}(A_{\mu',\nu'}) - \sum_{\mu_j\le\mu'\le\nu'\le\mu-1} \operatorname{dem}(A_{\mu',\nu'}) - \sum_{\nu+1\le\mu'\le\nu'\le\nu_j} \operatorname{dem}(A_{\mu',\nu'}) \\
&= \sum_{i=\mu_j}^{\mu-1} \big(w(C^i) - w(A_i)\big) + \sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) \\
&\quad + \sum_{i=\nu+1}^{\nu_j} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\Big(c^{\mu_j-1}_{k_{\mu_j-1}}\Big) + w\Big(c^{\nu_j+1}_1\Big)\right) \\
&\quad - \sum_{\mu_j\le\mu'\le\nu'\le\mu-1} \operatorname{dem}(A_{\mu',\nu'}) - \sum_{\nu+1\le\mu'\le\nu'\le\nu_j} \operatorname{dem} A_{\mu',\nu'} \\
&\ge \sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\Big(c^{\mu_j-1}_{k_{\mu_j-1}}\Big) + w\Big(c^{\nu_j+1}_1\Big)\right) \\
&\quad - \frac{1}{2}\left(w\Big(c^{\mu_j-1}_{k_{\mu_j-1}}\Big) + w\Big(c^{\mu}_1\Big)\right) - \frac{1}{2}\left(w\Big(c^{\nu}_{k_\nu}\Big) + w\Big(c^{\nu_j+1}_1\Big)\right) \\
&= \sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\big(c^{\mu}_1\big) + w\left(c^{\nu}_{k_\nu}\right)\right)
\end{aligned}$$

This shows that inequality (6.7) holds for $A_{\mu,\nu}$.

Now we consider the case that $A_{\mu,\nu}$ is not completely contained in any interval A_{μ_j,ν_j}. Assume that there is an interval A_{μ_j,ν_j} that is completely contained in $A_{\mu,\nu}$ (for an index $j \in \{1,\dots,\rho\}$). If A_{μ_j,ν_j} is of type 1, then we have

$$\sum_{i=\mu_j}^{\nu_j} \big(w(C^i) - w(A_i)\big) \le \sum_{\mu_j\le\mu'\le\nu'\le\nu_j} \operatorname{supp}(A_{\mu',\nu'}) + \frac{1}{2}\left(w\Big(c^{\mu_j}_1\Big) + w\Big(c^{\nu_j}_{k_{\nu_j}}\Big)\right). \tag{6.8}$$

If A_{μ_j,ν_j} is of type 2, then Lemma 6.2 (b) implies

$$\sum_{i=\mu_j}^{\nu_j} \big(w(C^i) - w(A_i)\big) = \sum_{\mu_j\le\mu'\le\nu'\le\nu_j} \operatorname{dem}(A_{\mu',\nu'}) - \frac{1}{2}\left(w\Big(c^{\mu_j-1}_{k_{\mu_j-1}}\Big) + w\Big(c^{\nu_j+1}_1\Big)\right). \tag{6.9}$$

Now assume that A_{μ_j,ν_j} intersects $A_{\mu,\nu}$, but neither A_{μ_j,ν_j} is contained in $A_{\mu,\nu}$ nor $A_{\mu,\nu}$ is contained in A_{μ_j,ν_j}. W.l.o.g. we can assume that $\mu < \mu_j \le \nu < \nu_j$. If A_{μ_j,ν_j} is of type 1, we have

$$\sum_{\mu_j\le\mu'\le\nu'\le\nu} \operatorname{supp}(A_{\mu',\nu'}) \ge \sum_{i=\mu_j}^{\nu} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c^{\mu_j}_1\Big) + w\Big(c^{\nu}_{k_\nu}\Big)\right). \tag{6.10}$$

If A_{μ_j,ν_j} is of type 2, then we can conclude (applying Lemma 6.2 (b) to A_{μ_j,ν_j}):

$$\begin{aligned}\sum_{\substack{A_{\mu',\nu'}\cap A_{\mu,\nu}\neq\emptyset\\ \mu_j\le\mu'\le\nu'\le\nu_j}} \operatorname{dem}(A_{\mu',\nu'}) &= \sum_{\mu_j\le\mu'\le\nu'\le\nu_j} \operatorname{dem}(A_{\mu',\nu'}) - \sum_{\nu+1\le\mu'\le\nu'\le\nu_j} \operatorname{dem}(A_{\mu',\nu'}) \qquad (6.11)\\ &\ge \sum_{i=\mu_j}^{\nu_j} \big(w(C^i)-w(A_i)\big) + \frac{1}{2}\left(w\left(c^{\mu_j-1}_{k_{\mu_j-1}}\right) + w\left(c_1^{\nu_j+1}\right)\right)\\ &\quad - \sum_{i=\nu+1}^{\nu_j} \big(w(C^i)-w(A_i)\big) - \frac{1}{2}\left(w\left(c^{\nu}_{k_\nu}\right) + w\left(c_1^{\nu_j+1}\right)\right)\\ &= \sum_{i=\mu_j}^{\nu} \big(w(C^i)-w(A_i)\big) + \frac{1}{2}\left(w\left(c^{\mu_j-1}_{k_{\mu_j-1}}\right) - w\left(c^{\nu}_{k_\nu}\right)\right).\end{aligned}$$

Since the types of the intervals $A_{\mu_1,\nu_1},\dots,A_{\mu_\rho,\nu_\rho}$ alternate, summing up the inequalities (6.8), (6.9), (6.10) and (6.11) for all intervals A_{μ_j,ν_j} that intersect $A_{\mu,\nu}$ shows the correctness of inequality (6.7). □

Theorem 6.3 shows that the absolut values of the supply and demand numbers we have defined are big enough, while the following theorem shows that if we want to have a feasible assignment after *any* interior flow realization, then we decrease the absolute values of these numbers:

Theorem 6.4 *(a) Assume that the minimum cost flow instance is changed by decreasing one supply value. Let f be a solution of this modified instance. Then, no exact interior realization of f will lead to a feasible assignment.*

(b) Assume that the sum of all supply numbers is at least $\max\{w(Z) - w(C_Z) \mid Z \text{ zone}, C_Z \text{ set of circuits in zone } Z\}$. *Modify the minimum cost flow instance by increasing any demand value. Then, there is a maximum s-t-flow in G (not necessarily with minimum cost) such that no interior realization of f' will lead to a feasible assignment.*

Proof:

(a) Let $A_{\kappa,\lambda}$ be a supply node whose supply number has been decreased. Let $A_{\mu,\nu}$ be the smallest interval such that $\mu\le\kappa\le\lambda\le\nu$ and that each supply interval that has at least one region with $A_{\mu,\nu}$ in common is completely contained in $A_{\mu,\nu}$. Let C_i be the set of circuits in region A_i before the realization. Before realizing the flow, there were circuits of total size $\sum_{i=\mu}^{\nu} w(C_i)$ in the interval. Using Lemma 6.2 (a), we see that the exact interior realization has removed circuits with total size less than

$$\sum_{\mu\le\mu'\le\nu'\le\nu} \operatorname{supp}(A_{\mu',\nu'}) = \sum_{i=\mu}^{\nu} \big(w(C^i)-w(A_i)\big) - \frac{1}{2}\left(w\left(c_1^{\mu}\right) + w\left(c^{\nu}_{k_\nu}\right)\right).$$

Therefore, the remaining circuits have a size greater than

$$\sum_{i=\mu}^{\nu} w(A_i) + \frac{1}{2}\left(w\left(c_1^{\mu}\right) + w\left(c_{k_\nu}^{\nu}\right)\right),$$

so they cannot be placed legally in that interval.

(b) Let A be a demand interval whose demand number has been increased. Let f be any solution of the modified flow problem. Now increase the incoming flow for every interval in the zone that A belongs to, as much as possible (we have to lower the incoming flow for other demand regions to do so). Then, applying Lemma 6.2 (b) it is easy to see that any interior realization of the modified flow will lead to an assignment such that the circuits assigned to A will not fit into it. □

In Theorem 6.3 we considered very restricted realizations of a given flow f, namely exact interior realizations. However, we cannot assume that such a realization is possible. We will now examine what happens if we skip these restrictions.

Instead of considering exact realizations of a flow f, we will consider approximative realizations: If we have a flow $f((A, A'))$ between two regions A and A', then we may move circuits whose total size is "a little bit" smaller or bigger than $f((A, A'))$. We also allow the leftmost and the rightmost circuit of each region to be moved. However, to have a chance to meet the capacity constraints after the approximative realization we have to make some assumptions on the balances $\mathrm{bal}(A_{\mu,\nu})$ of the intervals $A_{\mu,\nu}$ (i.e., the total size of circuits that leave $A_{\mu,\nu}$ minus the total size of circuits that are moved into $A_{\mu,\nu}$). We demand that for each interval $A_{\mu,\nu}$ we have

$$\mathrm{bal}(A_{\mu,\nu}) \geq \sum_{i=\mu}^{\nu} \left(w(C^i) - w(A_i)\right) - \frac{1}{2}\left(w\left(c_1^{\mu}\right) + w\left(c_{k_\nu}^{\nu}\right)\right). \tag{6.12}$$

Note that if an approximative realization meets the above condition, then we can (using the edges leaving t) construct a maximum s-t-flow f' such that the approximative realization is an exact realization of f'. The difference between the cost of f' and f is what we lose due to the fact that we move only complete circuits.

Theorem 6.5 *Let f be an optimum solution of the minimum cost flow problem. Then, any approximate realization of f meeting condition (6.12) for each interval $A_{\mu,\nu}$ leads to an assignment of the circuits to the regions for which there is a non-overlapping placement such that each circuit is placed within the region it is assigned to or within a neighbouring region.*

Proof: Let $\{A_1, \ldots, A_l\}$ be a set of regions that form a zone, and let $l\_\mathrm{coor}(A_i)$ ($r\_\mathrm{coor}(A_i)$) be the x-coordinate of the left (right) border of A_i ($i = 1, \ldots, l$). Let $\hat{C}^i = \{\hat{c}_1^i, \ldots, \hat{c}_{\hat{k}_i}^i\}$

be the set of circuits that was assigned to A_i before the movement, and $C^i = \{c_1^i, \ldots, c_{k_i}^i\}$ be the set of circuits that is assigned to A_i after the movement ($i = 1, \ldots, l$). Condition (6.12) implies that we have for each interval $A_{\mu,\nu}$ ($1 \leq \mu \leq \nu \leq l$)

$$\sum_{i=\mu}^{\nu}(w(C^i) - w(A_i)) \leq \frac{1}{2}\left(w\left(\hat{c}_1^{\mu}\right) + w\left(\hat{c}_{k_\nu}^{\nu}\right)\right). \tag{6.13}$$

Now, place the circuits in the set $\bigcup_{i=1}^{l} C^i$ from left to right in such a way that each circuit is placed at the leftmost possible position under the following constraints: no two circuits may overlap, and for $i = 1, \ldots, l$, the left border of circuit C_1^i may not be placed to the left of $l\_coor(A_i) - \frac{1}{2}w\left(\hat{c}_1^i\right)$. Then, using (6.13), it is easy to see that the right border of each circuit $c_{\hat{k}_i}^i$ will not be to the right of $r\_coor(A_i) + \frac{1}{2}w\left(\hat{c}_{k_i}^i\right)$. The theorem follows from the fact that no movable circuit is wider than twice the width of the regions. □

6.4.4 Construction of the Graph

Before we will describe how we choose the circuits that we move in a flow realization, we will examine the size of the graph G in the minimum cost flow instance. Obviously, the graph is bigger than the one used in the previous approach by Vygen [1998] as the latter contains only regions in its node set. As we have pointed out above, considering intervals in addition to the regions is desirable since it helps avoiding unneccessary movements, but this interval approach would be quite useless in practice if it could not be implemented efficiently.

By definition of the supply and demand intervals it is not immediately clear how many of these intervals can exist (compared to the number of regions) and how these intervals can be computed efficiently. We will show that the number of supply and demand intervals is linear in the number of regions and that all these intervals can be computed in linear time, provided that regions and circuits are sorted. This is essential when dealing with large practical instances.

The following lemma will help us to bound the number of supply nodes in the minimum cost flow instance:

Lemma 6.6 *For $\mu < \kappa \leq \lambda < \nu$ with $supp(A_{\kappa,\lambda}) > 0$, we have $supp(A_{\mu,\nu}) = 0$.*

Proof: Assume that we have $\text{supp}(A_{\mu,\nu}) > 0$ and $\text{supp}(A_{\kappa,\lambda}) > 0$ for intervals $A_{\mu,\nu}$ and $A_{\kappa,\lambda}$ with $\mu < \kappa \leq \lambda < \nu$. Because of $\text{supp}(A_{\mu,\nu}) > 0$ we have $s_{\mu,\nu} > 0$ and therefore

$s_{\mu,\nu} = \sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\big(w(c_1^\mu) + w(c_{k_\nu}^\nu)\big)$. We can conclude:

$$\begin{aligned}
\mathrm{supp}(A_{\mu,\nu}) &= s_{\mu,\nu} - \sum_{\substack{\mu\le\mu'\le\nu'\le\nu \\ (\mu,\nu)\neq(\mu',\nu')}} \mathrm{supp}(A_{\mu',\nu'}) \\
&= s_{\mu,\nu} - \mathrm{supp}(A_{\mu,\lambda}) - \mathrm{supp}(A_{\kappa,\nu}) - \sum_{\substack{\mu\le\mu'\le\nu'\le\nu \\ (\mu',\nu')\notin\{(\mu,\nu),(\mu,\lambda),(\kappa,\nu)\}}} \mathrm{supp}(A_{\mu',\nu'}) \\
&\le \sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^\mu\Big) + w\Big(c_{k_\nu}^\nu\Big)\right) - s_{\mu,\lambda} - s_{\kappa,\nu} \\
&\quad + \sum_{\kappa\le\mu'\le\nu'\le\lambda} \mathrm{supp}(A_{\mu',\nu'}) \\
&\le \sum_{i=\mu}^{\nu} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^\mu\Big) + w\Big(c_{k_\nu}^\nu\Big)\right) \\
&\quad - \left(\sum_{i=\mu}^{\lambda} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^\mu\Big) + w\Big(c_{k_\lambda}^\lambda\Big)\right)\right) \\
&\quad - \left(\sum_{i=\kappa}^{\nu} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^\kappa\Big) + w\Big(c_{k_\nu}^\nu\Big)\right)\right) \\
&\quad + \sum_{\kappa\le\mu'\le\nu'\le\lambda} \mathrm{supp}(A_{\mu',\nu'}) \\
&= -\sum_{i=\kappa}^{\lambda} \big(w(C^i) - w(A_i)\big) + \frac{1}{2}\left(w\Big(c_1^\kappa\Big) + w\Big(c_{k_\lambda}^\lambda\Big)\right) + \sum_{\kappa\le\mu'\le\nu'\le\lambda} \mathrm{supp}(A_{\mu',\nu'}) \\
&= -s_{\kappa,\lambda} + \sum_{\kappa\le\mu'\le\nu'\le\lambda} \mathrm{supp}(A_{\mu',\nu'}) \\
&= 0
\end{aligned}$$

□

We have a similar result for the demand intervals:

Lemma 6.7 *For $\mu < \kappa \le \lambda < \nu$ with $dem(A_{\kappa,\lambda}) < 0$ we have $dem(A_{\mu,\nu}) = 0$.*

Proof: The proof is analogous to the proof of Lemma 6.6. □

Corollary 6.8 *The number of nodes in the graph of the minimum cost flow instance is linear in the number of regions.*

Proof: Lemma 6.6 shows that the number of supply intervals is at most twice the number of regions, and Lemma 6.7 proves that also the number of demand intervals is at most twice the number of regions. □

Theorem 6.9 *Given the list of zones such that the regions in each zone and the circuits in each region are sorted from left to right, we can compute the supply and demand intervals in linear time.*

Proof: With the following algorithm we can compute the supply intervals in a zone:

SUPPLY INTERVALS

Input: A set of regions $A_1, \ldots, A_l$ that form a zone and sets $C_1, \ldots C_l$ of circuits that are assigned to the regions.

Output: Numbers $\text{supply}_{i,j}$ for each pair (i,j) with $1 \le i \le j \le l$ and $\text{supply}_{i,j} > 0$.

① $a_0 := 0$; $b_0 := 0$;

② FOR $(j = 1$ TO $l)$
$a_j := w(C^j) - w(A_j) - \min\left\{\frac{1}{2}w\left(c_1^j\right), b_{j-1} - a_{j-1}\right\}$;
$b_j := \max\left\{0, a_j - \frac{1}{2}w\left(c_{k_j}^j\right)\right\}$;

③ $i = l$;

④ FOR $(j = l$ DOWNTO $1)$
$i = \min\{i, j\} + 1$;
$ov_i := w(C^i \cup \cdots \cup C^j) - \frac{1}{2}w\left(c_{k_j}^j\right) - w(A_i) - \cdots - w(A_j)$;
WHILE $(b_j > 0)$
$i := i - 1$;
$ov_i := w(C^i) - w(A_i) - \min\left\{\frac{1}{2}w\left(c_{k_i}^i\right), -ov_{i+1}\right\}$;
$\text{supply}_{i,j} := \min\left\{b_j, \max\left\{0, ov_i - \frac{1}{2}w\left(c_1^i\right)\right\}\right\}$;
$b_j := b_j - \text{supply}_{i,j}$;
$ov_i := ov_i - \text{supply}_{i,j}$;

Obviously, the above algorithm has a linear running time. We will show that the numbers $\text{supply}_{i,j}$ it computes are correct, i.e., that $\text{supply}_{i,j} = \text{supp}(A_{i,j})$ for $1 \le i \le j \le l$.

The first loop, ②, computes numbers a_j and b_j for $j = 1, \ldots, l$. We have

$$a_j = w(C^j) - w(A_j) + \max\left\{-\frac{1}{2}w\left(c_1^j\right), a_{j-1} - b_{j-1}\right\},$$

for $j = 1, \ldots, l$, and thus

$$a_j = \max_{p=1}^{j}\left(\sum_{i=p}^{j}(w(C^i) - w(A_i)) - \frac{1}{2}w\left(c_1^p\right) - \sum_{i=p}^{j-1} b_i\right)$$

for $j = 1, \ldots, l$. Therefore, we have

$$b_j = \max\left\{ \max_{p=1}^{j} \left(\sum_{i=p}^{j} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^p\Big) + w\Big(c_{k_j}^{\nu}\Big)\right) - \sum_{i=p}^{j-1} b_i \right), 0 \right\} \tag{6.14}$$

for $j = 1, \ldots, l$.

Using (6.14), we can prove

$$b_j = \sum_{\mu=1}^{j} \operatorname{supp}(A_{\mu,j}) \tag{6.15}$$

for $j = 1, \ldots, l$, by induction on j. The case $j = 1$ is trivial. Let now $j > 1$.

If $b_j > 0$, then, by (6.14), there exists a $p \in \{1, \ldots, j-1\}$ with $b_j = \sum_{i=p}^{j} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^p\Big) + w\Big(c_{k_j}^{\nu}\Big)\right) - \sum_{i=p}^{j-1} b_i$. Hence

$$\begin{aligned} \sum_{i=p}^{j} b_i &= \sum_{i=p}^{j} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^p\Big) + w\Big(c_{k_j}^{\nu}\Big)\right) \\ &\leq \sum_{p \leq \mu \leq \nu \leq j} \operatorname{supp}(A_{\mu,\nu}) \leq \sum_{1 \leq \mu \leq \nu \leq j-1} \operatorname{supp}(A_{\mu,\nu}) + \sum_{1 \leq \mu \leq j} \operatorname{supp}(A_{\mu,j}). \end{aligned}$$

Using the induction hypothesis for $i = p, \ldots, j-1$ yields $b_j \leq \sum_{\mu=1}^{j} \operatorname{supp}(A_{\mu,j})$. Of course, this inequality is also true for $b_j = 0$.

To prove the converse inequality, let p be the minimum index such that each region among $A_p, \ldots, A_j$ belongs to a supply interval (if A_j itself does not belong to a supply interval, there is nothing more to prove). Note that, by (6.14), $b_j \geq \sum_{i=p}^{j} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w(c_1^p) + w(c_{k_j}^{\nu})\right) - \sum_{i=p}^{j-1} b_i$, hence

$$\sum_{i=p}^{j} b_i \geq \sum_{i=p}^{j} \big(w(C^i) - w(A_i)\big) - \frac{1}{2}\left(w\Big(c_1^p\Big) + w\Big(c_{k_j}^{\nu}\Big)\right) = \sum_{p \leq \mu \leq \nu \leq j} \operatorname{supp}(A_{\mu,\nu}) = \sum_{1 \leq \mu \leq \nu \leq j, p \leq \nu} \operatorname{supp}(A_{\mu,\nu}),$$

where the first equality follows from Lemma 6.2 (a). Using the induction hypothesis for $i = p, \ldots, j-1$ yields $b_j \geq \sum_{\mu=1}^{j} \operatorname{supp}(A_{\mu,j})$.

Knowing that (6.15) holds after ②, it is easy to see that the second loop ④ computes the correct numbers.

The demand intervals can be computed by a very similar algorithm. □

6.5 Flow Realization

In this section, we realize a maximum s-t-flow f of minimum cost, in the model described in the previous section. It will usually be impossible to realize f exactly. However, by Theorem 6.5, the realization of *any* maximum s-t-flow leads to an assignment of circuits

to regions, such that a legal placement within the zones exists where each circuit is placed within its region and/or a horizontally adjacent region. This is what we try to obtain.

Let G' be the subgraph of G induced by all regions (i.e., without intervals, s, and t). By construction, all edges in G' have positive cost. Hence, the edges in G' that have positive flow constitute an acyclic digraph. Thus, they can be scanned in topological order. For a set F of edges and a flow f, we define $f(F) := \sum_{e \in F} f(e)$.

Initially, we mark all vertices as unscanned and set $f' := f$. In each step we consider a vertex v with $f'(\delta^-_{G'}(v)) = 0$ and $f'(\delta^+_{G'}(v)) > 0$.

Let $X = \{x_1, \ldots, x_k\}$ be the set of neighbours of v that are not marked as scanned. We look for partitions $p^v : C^v \to X \cup \{v\}$ and $p^x : C^x \to \{x, v\}$ for $x \in X$ (where C^z is the set of circuits assigned to z for any region z) such that

$$(*) \quad \sum_{y \in X \cup \{v\}} \sum_{c \in C^y : p^y(c) = v} w(c) - \sum_{c \in C^v} w(c) + f'\left(\delta^+_{G'}(v)\right) \leq \text{lack}(v)$$

and

$$(**) \quad \sum_{c \in C^v : p^v(c) = x} w(c) - \sum_{c \in C^x : p^x(c) = v} w(c) - f'(v, x) \leq \text{lack}(x) \quad \forall x \in X$$

where $\text{lack}(z)$ is the additional amount of flow region z is able to take. So, at the beginning, we have $\text{lack}(z) = -\text{dem}(z) - f((z, t))$ for a demand region z and $\text{lack}(z) = 0$ for all other regions. If, during the flow realization, the total size of circuits moved to z differs from the flow on the corresponding edges or if the outflow of z is increased, the number $\text{lack}(z)$ is updated appropriately.

The cost of a partition is given by the sum of the moving costs over all circuits. Then, the optimum one among these partitions can be found by a dynamic programming algorithm (similarly to standard algorithms for the knapsack problem). To bound the running time it is useful to allow only few circuits to move along zero flow edges.

If we find such a partition, we move the circuits accordingly and mark v as scanned. Otherwise, we try to choose a partition for which condition $(*)$ is fulfilled and only condition $(**)$ is violated for some $x \in X$. We choose the partition with minimum such violation. Next we augment the flow f' along successive shortest paths from x to t (increasing $\text{lack}(x)$) until condition $(**)$ holds. We require that none of the augmenting paths contains a vertex that is marked as scanned. If no such augmenting path can be found, we choose a partition that violates (*) or $(**)$ minimally (in this case an infeasible assignment will result). After realizing the outflow of v we set $f'(e) := 0$ for all $e \in \delta^+_{G'}(v)$.

We compute the partitions $p^v : C^v \to X \cup \{v\}$ and $p^x : C^x \to \{x, v\}$ for $x \in X$ by dynamic programming. We choose upper bounds $max_{v \to x} \in \mathbb{N}$ and $max_{x \to v} \in \mathbb{N}$ on the total size of circuits we want to move from v to x and from x to v, respectively. Then, we compute for each vector $(w_1, \ldots, w_k)$ with $0 \leq w_i \leq max_{v \to x_i}$ a cheapest assignment $p^v : C^v \to X \cup \{v\}$ where $\sum_{c \in C^v, p^v(c) = x_i} size(c) = w_i$. By dynamic programming these partitionings can be computed in time $O(|C^v| \cdot k \cdot \Pi_{i=1}^k (max_{v \to x_i}))$. Similarly, we compute by dynamic programming for each index w'_i with $0 \leq w'_i \leq max_{x_i \to v}$, a cheapest assignment $p^{x_i} : C^{x_i} \to \{x_i, v\}$ with $\sum_{c \in C^{x_i}, p^{x_i}(c) = v} size(c) = w'_i$. We can combine these partitionings in time $O(k \cdot \Pi_{i=1}^k \max_{v \to x_i} \cdot \max_{x_i \to v})$ to compute a cheapest partitioning.

The running time of the flow realization depends on the choice of $max_{v\to x}$ and $max_{x\to v}$ (for $x \in X$). In our experiments, the following choice turned out to be reasonable. If we have $f'((v,x)) > 0$, then we choose $max_{v\to x_i} = f'((v,x_i)) + \max\{w(c) \mid c \in C^v\}$. The other values $max_{v\to x}$ and $max_{x\to v}$ (for $x \in X$) are bounded by 10.

If these numbers happen to be too large due to very large flow, we run a preprocessing step that greedily chooses some circuits out of C^v, moves them along the flow edges and reduces the flow correspondingly.

After (approximately) realizing the flow by moving circuits between the regions, the assignment of the circuits to the regions will, in general, not yet be feasible. In that case, we iterate phase one, but in each iteration we double the width of the grid we use to divide the zones. If necessary, we repeat the phase even for complete zones. After some iterations, the circuits can be legalized in their zone by the algorithm described in Section 6.7.

6.6 Implementational Improvements

To make the algorithm work efficiently in practice, even on large and difficult instances, experiments have shown that some changes of the algorithm are necessary:

- In a preprocessing step, all circuits overlapping a blockage a moved to the next location they can be placed at.
- After some of the iterations mentioned in the previous section, we increase all demand values in each iteration. This trick enforces a termination of the algorithm even on very hard instances. On most instances, this is not necessary, since experiments show that, in general, the algorithm terminates after at most 6 iterations (see Table 6.2).
- We use two parallel edges between the regions, one with small costs but finite capacity and the other one with higher costs but infinite capacity.

6.7 Single-Row Placement

The one-dimensional placement problem consists of placing a set of cells within a single row with minimum movement. Even this restricted problem is strongly *NP*-hard, it includes bin-packing, see Garey and Johnson [1975]. We consider an even more restricted problem, which is easier to solve: we assume a fixed ordering of the cells in each row (according to the original x-coordinates before legalization), so only the horizontal distances between the circuits can be changed.

Kahng, Tucker, and Zelikovsky [1999] developped a so-called Clumping Algorithm which solves the problem optimally. Their algorithm needs time $\Theta(n \log^2 n)$ in the unweighted case and $\Theta(n^2)$ in the weighted case, but with different data structures it can be implemented to run in $O(n \log n \log\log n)$ in the unweighted case and $O(n \log^2 n)$ in the weighted case (see Brenner and Vygen [2000]). Garey, Tarjan, and Wilfong [1988] even described an $O(n \log n)$-time algorithm for the weighted case. Since their algorithm is formulated in the

context of scheduling problems it has been overlooked so far by the chip design community. All these algorithms work not only with circuit movement as objective function but also if one wants to minimize the total length of the nets connected to the circuits of the row.

For the total (weighted) squared movement, the CLUMPING ALGORITHM can even be implemented in linear time. We will now describe this linear-time algorithm.

We start by recalling the CLUMPING ALGORITHM in a more general form. Let us define the SINGLE ROW OPTIMIZATION PROBLEM (SROP) as follows. Let $f_1, \ldots, f_n : \mathbb{R} \to \mathbb{R}$ be convex functions, $w_1, \ldots, w_n > 0$ and $x_{\min}, x_{\max} \in \mathbb{R}$ with $x_{\max} - x_{\min} \geq w_1 + \ldots + w_n$. Then we look for numbers $x_1, \ldots, x_n$ with $x_{\min} \leq x_1$, $x_i + w_i \leq x_{i+1}$ for $i = 1, \ldots, n-1$, $x_n + w_n \leq x_{\max}$, and $\sum_{i=1}^{n} f_i(x_i)$ minimum. By letting $w_1, \ldots, w_n$ be the widths of the circuits in a zone (in their horizontal order) and $f_i(x)$ be the costs for placing the i-th circuit at position x, our problem reduces to the SROP. Kahng, Tucker, and Zelikovsky [1999] and Brenner and Vygen [2000] discussed the following algorithm for the special case of piecewise linear functions:

CLUMPING ALGORITHM

Input: $n \in \mathbb{N}$. Convex functions $f_1, \ldots, f_n : \mathbb{R} \to \mathbb{R}$.
Widths $w_1, \ldots, w_n > 0$ and $x_{\min}, x_{\max} \in \mathbb{R}$ with $x_{\max} - x_{\min} \geq w_1 + \ldots + w_n$.

Output: $x_1, \ldots, x_n$ with $x_{\min} \leq x_1$, $x_i + w_i \leq x_{i+1}$ for $i = 1, \ldots, n-1$, $x_n + w_n \leq x_{\max}$, and $\sum_{i=1}^{n} f_i(x_i)$ minimum.

① Set $x_0 := x_{\min}$ and $W_0 := 0$ and $W_i := w_i$ for $i = 1, \ldots, n$.
Let $\mathcal{L}$ be the list with 0 as the only element.
Let $g_i : \mathbb{R} \to \mathbb{R}$ be a function with $g_i(x) = f_i(x)$ for $x \in \mathbb{R}$ and $i = 1, \ldots, n$.

② FOR $(i = 1, \ldots, n)$
Add i as the last element to $\mathcal{L}$.
ROWPLACE$((g_j)_{j\in\mathcal{L}}, (W_j)_{j\in\mathcal{L}})$.

③ FOR $(i \in \{1, \ldots, n\} \setminus \mathcal{L})$
Set $x_i := x_h + \sum_{j=h}^{i-1} w_j$ where h is the maximum index smaller than i that belongs to $\mathcal{L}$.

ROWPLACE$((g_j)_{j\in\mathcal{L}}, (W_j)_{j\in\mathcal{L}})$:

① Let k be the last element of $\mathcal{L}$, and let h be the predecessor of k in $\mathcal{L}$.

② IF($h = 0$ or $x_h + W_h \leq \min\{x_{\max} - W_k, \max\{x \mid g_k(x) \text{ minimum}\}\}$) THEN
Set $x_k := \max\{x_h + W_h, \min\{x_{\max} - W_k, \min\{x \mid g_k(x) \text{ minimum}\}\}\}$
RETURN.
ELSE
Redefine g_h by $g_h : x \mapsto g_h(x) + g_k(x + W_h)$.
Set $W_h := W_h + W_k$.
Remove k from $\mathcal{L}$.
ROWPLACE$((g_j)_{j\in\mathcal{L}}, (W_j)_{j\in\mathcal{L}})$.
RETURN.

Theorem 6.10 *The* CLUMPING ALGORITHM *finds an optimum placement. If all f_i are quadratic, it can be implemented in linear time.*

Proof: In the following instances of the *SROP* we will always use the same interval $[x_{\min}, x_{\max}]$.

Let $(x_i)_{i\in\{1,\dots,n\}}$ be the output of the algorithm. $(x_i)_{i\in\mathcal{L}}$ is an optimum solution of the instance of the SROP defined by $(g_i, W_i)_{i\in\mathcal{L}}$ because every single element $i \in \mathcal{L}$ is placed in the minimum of g_i. As we have $\sum_{i\in\mathcal{L}} g_i(x_i) = \sum_{i=1}^{n} f_i(x_i)$, the vector $(x_i)_{i\in\{1,\dots,n\}}$ is an optimum solution of the SROP instance defined by $(f_i, w_i)_{i\in\{1,\dots,n\}}$ under the additional constraint that for $i \in \{1,\dots,n\} \setminus \mathcal{L}$ we have $x_i = x_{i-1} + w_i$. These constraints are added in the "ELSE"-part of step ② where we restrict ourselves to solutions in which $x_k = x_{k-1} + w_{k-1}$, (*clumping operation*). We have to show that there is always an optimum solution of the initial SROP instance that meets all these additional constraints. Assume that this is not the case. Then consider the clumping operation in which we add the first constraint such that there is no optimum solution of the initial SROP meeting all the constraints added so far. Let i be the number of the corresponding iteration, and let k be the last element of $\mathcal{L}$ and h its predecessor in the call of the function ROWPLACE. Let x_h be the position of h (before clumping it). There is an optimum solution $(\tilde{x}_j)_{j\in\{1,\dots,n\}}$ meeting all the constraints we added before, so in this solution we have $\tilde{x}_j = \tilde{x}_{j-1} + w_{j-1}$ for $j \in \{h+1,\dots,k-1\} \cup \{k+1,\dots,i\}$. However, we have $\tilde{x}_k > \tilde{x}_{k-1} + w_{k-1}$ and hence $\tilde{x}_k > \tilde{x}_h + W_h$. The position x_h has been computed by the function ROWPLACE, therefore we know that $\min\{x_{\max} - W_h, \min\{x \mid g_h(x) \text{ minimum}\}\} \le x_h \le \min\{x_{\max} - W_h, \max\{x \mid g_h(x) \text{ minimum}\}\}$. We jump into the "ELSE"-part of step ②, so $h > 0$ and $x_h + W_h > \min\{x_{\max} - W_k, \max\{x \mid g_k(x) \text{ minimum}\}\}$.

If $\tilde{x}_k > x_h + W_h$, this implies $\tilde{x}_k > \min\{x_{\max} - W_k, \max\{x \mid g_k(x) \text{ minimum}\}\}$, therefore $g_k(x'_k) < g_k(\tilde{x}_k)$ for $x'_k := \min\{\tilde{x}_h + W_h, \max\{x \mid g_k(x) \text{ minimum}\}\}$. As we have $g_k(x) = \sum_{j=k}^{i-1} f_j(x + \sum_{l=k}^{j-1} w_l)$, this means that $(\tilde{x}_j)_{j\in\{1,\dots,n\}}$ was not an optimum solution. Therefore, we may assume that $\tilde{x}_k \le x_h + W_h$. This implies $\tilde{x}_h \le x_h$.

If $\tilde{x}_h < x_h$, then we have $\tilde{x}_h < \min\{x_{\max} - W_h, \max\{x \mid g_h(x) \text{ minimum}\}$. Hence $g_h(x'_h) \le g_h(\tilde{x}_h)$ for $x'_h := \min\{\tilde{x}_k - W_h, \max\{x \mid g_h(x) \text{ minimum}\}$. As we have $g_h(x) = \sum_{j=h}^{k-1} f_j(x + \sum_{l=h}^{j-1} w_l)$, the vector $(\hat{x}_j)_{j\in\{1,\dots,n\}}$ with $\hat{x}_j := \tilde{x}_j$ for $j \in \{1,\dots,n\} \setminus \{h,\dots,k-1\}$ and $\hat{x}_j := x'_h + \sum_{l=h}^{j-1} w_l$ for $j \in \{h,\dots,k-1\}$ would be a solution to the initial SROP instance whose cost is as most as big as the cost of $(\tilde{x}_j)_{j\in\{1,\dots,n\}}$. As we have $\hat{x}_k = \hat{x}_{k-1} + w_{k-1}$ this is a contradiction.

Having proved optimality, we consider the runtime. As n elements are be added to $\mathcal{L}$ in step ② of the algorithm (and each element only once) and ROWPLACE is called only if an element is added to or removed from $\mathcal{L}$, the function ROWPLACE is called at most $2n$ times. For quadratic functions $f_i : x \mapsto a_i x^2 + b_i x + \text{const}$, ROWPLACE can be done in constant time as $\{x \mid f_i(x) \text{ minimum}\} = \{\frac{-b_i}{2a_i}\}$ and $f_h(x) + f_i(x + W_h) = (a_h + a_i)x^2 + (b_h + b_i + 2a_i W_h)x + \text{const}$. □

For the correctness of the CLUMPING ALGORITHM, we require the cost functions f_i to

be convex but apart from this restriction, we may choose them arbitrarily. For example, bounding-box netlength (under the assumption that all circuits in the other zones are fixed) or linear or quadratic movement can be reflected by the functions f_i.

6.8 Postoptimization

In Section 6.10, the experimental results will show that our algorithm is able to legalize a placement with a small average movement. However, it is still possible that there is a small number of circuits that are moved quite far during legalization. In order to reduce the largest movements, we apply as a third phase a postoptimization routine to these circuits that tries to move them towards their initial positions. The algorithm chooses a circuit c^* with the largest movement and marks it. Then, we consider a directed graph whose node set consists of all circuits C with $w(c) \geq w(c^*)$ in a certain area around c^* and all empty parts of zones in the same area that are big enough to contain c^*. We have edges between each pair (c, c') of circuits if c is not smaller than c', and edges from each circuit c to each free slot S that is big enough to contain c. The cost of an edge (c, c') or (c, S) is the cost for moving c from its recent position to the position of c' or to S, respectively. So if (x, y) is the original position of circuit c (before legalization), (x', y') is its recent position, and (x'', y'') is the position of circuit c' or of the free slot S, respectively, then the cost of edge (c, c') (or (c, s)) is $(x'' - x)^2 + (y'' - y)^2 - (x' - x)^2 - (y' - y)^2$, possibly multiplied by the weight of circuit c. It would be desirable to compute a shortest path from from c^* to a free slot or a shortest cycle containing c^*, but since there can be negative cycles in the graph, this task is NP-hard. Instead, we use the following dynamic programming approach that does not necessarily find shortest paths.

Let C be the set of circuits we want to consider, i.e., all circuits of width at least $w(c^*)$ in an area around c^*. Let $\mathcal{S}$ consist of the location of c^* and all free slots in the area. Then, we choose an upper bound $k_{\max}$ on the length of the paths we want to take into consideration. Experiments show that $k_{\max} = 10$ is a reasonable choice. For each path-length $k \in \{0, \ldots, k_{\max}\}$ and each cell $c \in C$, we look for a path $P_{c,k}$ from c to an element of $\mathcal{S}$ in the above digraph containing exactly k edges. For $P_{c,1}$, we use a shortest edge starting at c and ending in an element of $\mathcal{S}$, so for all circuits $c \in C$, the paths $P_{c,1}$ can easily be computed in time $O(|C| \cdot |\mathcal{S}|)$. If the paths $P_{c,k-1}$ of length $k-1$ are known, we try to choose for each circuit $c \in C$ a circuit $c' \in C$ for which a path $P_{c',k-1}$ exists with $w(c) \leq w(c')$, $c \notin V(P_{c',k-1})$ and

$$\begin{aligned} &cost((c, c')) + cost(P_{c',k-1}) \\ &= \min\{cost((c, c'')) + cost(P_{c'',k-1}) \,|\, c'' \in C, P_{c'',k-1} \text{ exists}, w(c) \leq w(c''), c \notin V(P_{c'',k-1})\}. \end{aligned}$$

If such a circuit c' exists, $P_{c,k}$ is defined as the concatenation of the edge (c, c') and the path $P_{c',k-1}$. If $\{c'' \in C \mid P_{c'',k-1} \text{ exists}, w(c) \leq w(c''), c \notin V(P_{c'',k-1})\}$ is empty, then there is no path $P_{c,k}$. Using dynamic programming, these paths can be computed in time $O(k_{\max}^2 |C|^2)$. Then, we choose a path P_{c^*,k^*} out of $\{P_{c^*,k} \mid k \in \{0, \ldots, k_{\max}\}\}$ that has minimum cost, and we move the corresponding circuits along the edges of the path (starting with c^*). Note that the path may end at the position that c^* had before the postoptimization, so it is also possible that this method moves circuits along a cycle in order to reduce the total

(squared) movement. Since there is always a path of cost 0 (we can move c^* to its own position), we always find a path of non-positive cost, and therefore the postoptimization cannot make the solution worse.

We repeat the algorithm for the next unmarked circuit with the largest movement until we have considered a certain number of circuits or if the largest movement is smaller than a given threshold.

6.9 Overall Algorithm

Schematically, our algorithm can be described as follows:

Legalization Algorithm	
Input:	A global placement of a circuit set C. A chip area A partitioned into regions. A set of blockages.
Output:	A legal placement of C

① Assign each circuit to the region it is placed in.
WHILE(Assignment to the regions is not feasible)
    Construct minimum cost flow instance // Section 6.4.
    Compute minimum cost flow f.
    Realize f. // Section 6.5

② Solve single-row problem on each zone // Section 6.7

③ Run postoptimization on critical circuits // Section 6.8

6.10 Experiments

We will compare our algorithm both to a similar previous approach (Vygen [1998]) that stated a minimum cost flow problem by using only region vertices, and to lower bounds that we compute by solving a relaxation of the legalization problem.

6.10.1 Lower Bounds

By comparing our results to lower bounds, we will be able to show that our algorithm is less than a few percent worse than *any* possible algorithm.

To compute good lower bounds, we formulate the legalization problem as an integer linear program (ILP). Let $C = \{c_1, \dots, c_m\}$ be the set of circuits that have to be placed in a grid $\{x_\delta, 2x_\delta, \dots, Wx_\delta\} \times \{y_\delta, 2y_\delta, \dots, Hy_\delta\}$. To simplify the notation we assume that there are

no preplaced objects or blockages on the chip area. It is easy to see how blockages can be incorporated into our ILP formulation. For coordinates $(i, j) \in \{1, \ldots, W\} \times \{1, \ldots, H\}$. and a circuit c_k let $d_{i,j,k}$ be the (linear or squared) distance between the initial position of c_k and $(x_\delta i, y_\delta j)$. To compute lower bounds for the movements weighted by the circuits' widths, we multiply each distance by the width of c_k. We state the following ILP:

$$\text{minimize} \sum_{k=1}^{m} \sum_{i=1}^{W} \sum_{j=1}^{H} d_{i,j,k} \cdot x_{i,j,k}$$

s.t.

$$x_{i,j,k} \in \{0,1\} \qquad \begin{array}{l} \forall i \in \{1, \ldots, W\}, \\ j \in \{1, \ldots, H\}, \\ k \in \{1, \ldots, m\}; \end{array} \tag{6.16}$$

$$\sum_{i=1}^{W - \frac{w(c_k)}{x_\delta} + 1} \sum_{j=1}^{H} x_{i,j,k} = 1 \qquad \forall k \in \{1, \ldots, m\}; \tag{6.17}$$

$$\sum_{k=1}^{m} \sum_{i'=i-\frac{w(c_k)}{x_\delta}+1}^{i} x_{i',j,k} \leq 1 \qquad \forall i \in \{1, \ldots, W\}, \;\; j \in \{1, \ldots H\}; \tag{6.18}$$

The variable $x_{i,j,k}$ will be 1 if and only if the lower left corner of c_k is placed at position $(x_\delta i, y_\delta j)$. Conditions 6.16 and 6.17 model the constraint that each circuit has to be placed at exactly one position, while condition 6.18 guarantees disjointness.

Since the number of variables in this ILP fomulation is huge, we cannot hope for an exact solution of the ILP in reasonable time. Therefore we relax the problem in three ways:

- We skip the integrality constraints, so we replace the constraint $x_{i,j,k} \in \{0,1\}$ by $0 \leq x_{i,j,k} \leq 1$ (for $i \in \{1, \ldots, W\}$, $j \in \{1, \ldots, H\}$, $k \in \{1, \ldots, m\}$).
- We only consider locations close to the initial circuit position in the LP. So, we add a new variable y_k for $k \in \{1, \ldots, m\}$, replace the condition $\sum_{i=1}^{W-\frac{w(c_k)}{x_\delta}+1} \sum_{j=1}^{H} x_{i,j,k} = 1$ by $y_k + \sum_{i=1}^{W-\frac{w(c_k)}{x_\delta}+1} \sum_{j=1}^{H} x_{i,j,k} = 1$ (for $k \in \{1, \ldots, m\}$) and the objective function by "minimize $\sum_{k=1}^{m}(R \cdot y_k + \sum_{i=1}^{W} \sum_{j=0}^{H} d_{i,j,k} \cdot x_{i,j,k})$" where R is a parameter. In this new formulation, we can omit all variables $x_{i,j,k}$ with $d_{i,j,k} > R$. In the presence of blockages, it may be reasonable to use different values of R for the different circuits depending on the number of free locations near the initial circuit position. This way, we can consider the same number of free possible positions for each circuit.
- On large chips, we partition the chip area into an appropriate number of rectangular regions and compute the lower bounds on these smaller instances separately.

Using these relaxations, we are able to compute lower bounds even on very large chips by the LP-solver CPLEX 8.0, within a few hours of computing time. Since we do not need lower bounds for the algorithm but just for comparisons, this is not a problem.

6.10.2 The Testsuite

Chip	Blockages	Row height y_δ	Density	Overlaps
Jens	49	6.72 μm	77 %	2.207
Hans	146	9.60 μm	50 %	1.605
Christian	4 804	8.64 μm	50 %	1.394
James	579	6.72 μm	50 %	1.885
Aidan	937	9.72 μm	70 %	1.629
Dieta	1 961	6.72 μm	50 %	1.540
Sandra	16 115	6.72 μm	50 %	1.780
Josef	5 624	6.72 μm	50 %	1.484
Nadine	9 065	6.72 μm	50 %	1.605
Wolf	1 788	4.80 μm	50 %	1.421

Table 6.1: The chips used for the legalization experiments.

We tested our algorithm on some of the ASICs introduced in Table 2.1. Some additional facts about these chips that are relevant for legalization are given in Table 6.1. The second column shows the number of blockages on the chip area including all circuits that have been fixed before legalization. Column three contains the height of a standard circuit. The number in the next column is a user-defined parameter that controls the density in global placement. It is the maximum allowed density at the beginning of global placement. In each level, this density is increased by 1%, so at the end of global placement the maximum allowed density is about 10% higher than the number in this column. To estimate how far a given placement is away from a legal placement, we haven chosen the following computation: If C_{row} is the set of standard circuits to be placed and C_{macro} the set of macros on the chip, then we compute

$$\left(\sum_{c \in C_{row}} \sum_{c' \in (C_{row} \cup C_{macro})} D(c, c') \right) / \left(\sum_{c \in C_{row}} w(c) \cdot y_\delta \right),$$

where $D(c, c')$ is the size of the areas covered both by circuit c and circuit c'. This number is 1 for a legal placement, and the more overlaps exist, the larger this number is. The table shows the number for our instances in the last column.

6.10.3 Running Time and Memory Consumption

We tested our detailed placement algorithm, implemented in C, on an IBM 680 with 600MHz RS-IV processors. In the postoptimization step, we considered the 2000 circuits with the largest movement.

Columns two, three, and four of Table 6.2 show the running time for the single steps of our legalizer according to the schematic description in Section 6.9. Columns five contains the number of iterations that are necessary to compute a feasible assignment in phase 1. In the last column, the maximum allocated memory during legalization is presented. The numbers in the table demonstrate that the first phase is the most time-consuming part of the algorithm, and the running time of this phase does not only depend on the numbers of

Chip	Runtime			Number of Iterations	Maximum Memory
	Step ①	Step ②	Step ③		
Jens	0:11	0:10	0:08	4	144 MB
Hans	0:20	0:10	0:04	2	185 MB
Christian	3:30	0:39	1:19	5	941 MB
James	12:27	1:00	1:31	2	1 939 MB
Aidan	4:40	1:57	1:31	3	1 350 MB
Dieta	7:48	2:34	1:06	3	2 243 MB
Sandra	9:02	3:59	5:50	3	3 004 MB
Josef	17:36	4:05	9:46	6	3 792 MB
Nadine	18:35	4:50	8:48	5	4 481 MB
Wolf	56:08	3:33	13:29	6	8 534 MB

Table 6.2: Running times (mm:ss), number of iterations, and peak memory consumption in the legalization runs.

circuits to be placed (which is not too surprising). Nevertheless, even the first part does not take longer than an hour on a chip with 2.4 millions of circuits.

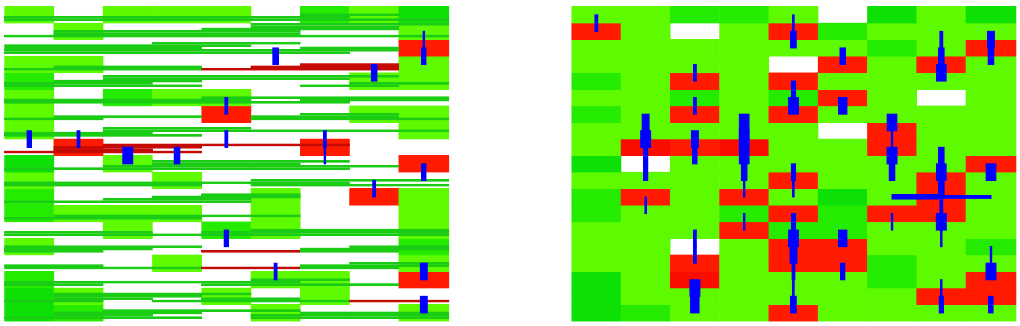
6.10.4 Comparison to Hard Bounds

Chip	Total supply		Total flow	
	Hard bounds	Soft bounds	Hard bounds	Soft bounds
Jens	227.9	28.2	258.5	28.4
Hans	48.9	0.8	54.6	0.8
Christian	2 626.0	28.5	2 770.7	3.0
James	1 565.6	127.1	1 714.4	130.3
Aidan	1 084.7	26.9	1 119.5	26.9
Dieta	2 265.2	21.9	2 460.5	21.9
Sandra	508.0	25.0	536.6	25.1
Josef	1 553.8	86.6	1 625.0	100.8
Nadine	817.5	41.2	856.8	42.9
Wolf	8 664.0	191.6	9 466.7	204.9

Table 6.3: Comparison between "hard" and "soft" bounds.

For an analysis of the results of our legalizer, we first show the amount of unnecessary movement we would produce if we used "hard bounds" between the regions during the first phase. In this approach, only regions are considered and we have a positive supply value if the circuits that are assigned to a region cannot be placed *complety* within this region (so circuits may not overlap a neighbouring region). Such an approach has been proposed by Vygen [1998]. We consider the first minimum cost flow instance of our algorithm and compute the total supply in this instance and the total flow in its solution. Table 6.3 shows the result both for our algorithm ("soft bounds") and for the variant of the algorithm where we used single regions as nodes of a minimum cost flow instance and computed

supply and demand values for each single region without considering neighbouring regions ("hard bounds"). The supply and flow values presented in the table are micrometers. The results show that more than 90 % of the supply computed with the hard bound approach is superfluous and just creates unnecessary movements. Therefore, it is absolutely mandatory to consider intervals in addition to regions.



(a) (b)

Figure 6.7: (a) Illustration of our minimum cost flow instance for a small part of a chip. Regions are shown as rectangles, intervals as horizontal lines. Green colors correspond to demand regions/intervals and red colors to supply regions/intervals. The blue lines are edges with positive flow in an optimum solution, where the width of a line corresponds to the amount of flow on that edge (only the flow between regions is shown). (b) The corresponding minimum cost flow instance and solution without taking intervals into account but using hard bounds between regions instead. The picture shows that one gets a significantly larger flow (and therefore, of course, larger movements).

Figure 6.7 illustrates this result by showing a minimum cost flow instance and its solution computed with our algorithm and comparing it to the corresponding instance and solution with hard bounds between the regions.

6.10.5 Movement Experiments

Table 6.4 and Table 6.5 summarize the results of our movement experiments. In Table 6.4, we consider the average movement (i.e., each circuit has weight 1), while for Table 6.5, we weight each circuit by its width and divide the sum of these weighted movements by the total width of all circuits. The latter movement computation is preferable if one assumes that the movement of a large circuit creates a bigger pertubation of the placement than the movement of a small circuit. Table 6.4 and Table 6.5 are organized in the same way. We measure the average L_1-movement (columns two to six) and the average squared L_2-movement (columns seven to eleven). All distances are given in (squared) micrometers. The number in column two is the lower bound computed by our ILP relaxation. Column three presents the movement in a legalization run with hard bound in the first phase (as described above). The columns four and five show the result of our legalizer without and with postoptimization, respectively. Again, we ran our postoptimization for the 2000 worst circuits. The relative gap between the result of our legalizer and the lower bound is shown in column six. Columns seven to eleven show the same numbers for the average linear

Chip	Squared movement					Linear movement				
	Lower bound	Hard bound legalization	Our algorithm Without PostOpt	Our algorithm With PostOpt	Gap	Lower bound	Hard bound legalization	Our algorithm Without PostOpt	Our algorithm With PostOpt	Gap
Jens	11.19	16.14	13.65	13.53	20.9 %	3.18	3.87	3.60	3.59	12.9 %
Hans	7.33	11.68	7.59	7.57	3.3 %	2.83	3.32	3.01	2.89	2.1 %
Christian	5.52	38.55	6.01	5.93	7.4 %	2.25	4.72	2.50	2.36	4.9 %
James	9.57	15.39	11.68	11.57	20.9 %	2.89	3.80	3.39	3.24	12.1 %
Aidan	7.79	11.51	8.41	8.38	7.6 %	2.79	3.29	3.01	2.95	5.7 %
Dieta	5.44	10.44	5.99	5.97	9.7 %	2.30	2.94	2.55	2.44	6.1 %
Sandra	6.22	7.61	6.63	6.61	6.3 %	2.46	2.73	2.61	2.59	5.3 %
Josef	10.13	13.94	12.06	10.94	8.0 %	2.27	2.74	2.56	2.50	10.1 %
Nadine	5.32	7.43	5.84	5.83	9.6 %	2.26	2.63	2.47	2.41	6.6 %
Wolf	2.41	8.14	2.84	2.62	8.3 %	1.41	2.29	1.59	1.51	7.1 %
Average					10.1 %					7.2 %

Table 6.4: Average movement during legalization.

Chip	Squared movement					Linear movement				
	Lower bound	Hard bound legalization	Our algorithm Without PostOpt	Our algorithm With PostOpt	Gap	Lower bound	Hard bound legalization	Our algorithm Without PostOpt	Our algorithm With PostOpt	Gap
Jens	12.37	18.06	13.77	13.65	10.3 %	3.32	3.97	3.51	3.51	5.7 %
Hans	7.34	18.67	7.59	7.57	3.1 %	2.64	3.86	3.05	2.70	2.3 %
Christian	6.28	75.18	7.18	6.95	10.7 %	2.29	7.63	2.51	2.41	5.2 %
James	9.85	17.32	11.02	10.92	10.9 %	2.81	3.69	3.24	2.99	6.4 %
Aidan	8.30	16.86	8.59	8.58	3.4 %	2.75	3.82	2.97	2.83	2.9 %
Dieta	5.26	18.06	5.74	5.71	8.6 %	2.11	3.55	2.33	2.20	4.3 %
Sandra	5.84	8.44	6.09	6.09	4.3 %	2.26	2.75	2.37	2.34	3.5 %
Josef	14.83	21.80	18.11	15.53	4.7 %	2.06	2.84	2.32	2.20	6.8 %
Nadine	5.01	9.70	5.42	5.41	7.9 %	2.07	2.90	2.24	2.21	6.8 %
Wolf	3.08	14.95	4.02	3.41	10.7 %	1.40	3.22	1.63	1.50	7.1 %
Average					7.4 %					5.1 %

Table 6.5: Average movement during legalization, weighted by the circuits' widths.

Density	Lower bound	Integral solution	Our algorithm	
			Without PostOpt	With PostOpt
23 %	1.567	1.614 3.0 %	1.614 0.0 %	1.614 0.0 %
30 %	1.739	1.740 0.1 %	1.790 2.9 %	1.760 1.1 %
40 %	2.932	3.076 4.9 %	3.500 13.8 %	3.500 13.8 %
50 %	4.771	5.050 5.8 %	6.250 23.8 %	6.080 20.4 %
60 %	6.324	6.716 6.2 %	9.128 35.9 %	8.813 31.2 %
70 %	9.606	10.016 ... 10.094 4.3 % 5.1 %	13.657 36.4 %	12.983 29.6 %
80 %	11.512	12.020 4.4 %	17.622 46.6 %	17.059 41.9 %

Table 6.6: Unweighted squared movement on a part of the chip James containing 408 circuits.

movement. The table demonstrates that our algorithm computes a legalization close to the optimum. On the unweighted instances, our results differ from the lower bound, on average, by 10.1 % for the squared movement and 7.2 % for the linear movement. When we weight the movement of each circuit by its width, the gap between the movement in our legalization and the lower bound is, on average, even only 7.4 % (squared movement) or 5.1 % (linear movement).

For the chips Jens and James, we get in the unweighted case a significantly larger gap between our results and the lower bound. For both chips, we used a higher density in global placement. Therefore it is interesting to find out if the higher density caused this effect and if the gap is big because our result is bad or because the lower bound is too weak. To analyze how much we lose due to skipping the integrality constraints we computed (approximately) ILP solutions on very small instances. To this end we placed the chip James with the smallest possible density (so we spread the circuits evenly over the chip area), and then considered a small window on it containing 408 circuits. The density in this window was 23 %. We ran the global placer on this window using different values for the parameter that controls the initial placement density. In the experiments we only considered total unweighted squared movement. For each of the global placements we computed the relaxed lower bounds (but of course without dividing the problem into parts) and an integral solution of the corresponding ILP (or at least an approximation of it). Table 6.6 summarizes the result of these experiments. The number in the first column is the maximum allowed density at the beginning of the global placement run on the window. We increased this allowed density by 1 % in each level, so at the end of the global placement the maximum allowed density was 6 % bigger than the number in the table. Column two shows the lower bounds computed with the LP (all numbers in squared micrometers). The third column contains the corresponding ILP solutions or (for the run with 70 % density) lower and

upper bounds for their solutions. Again we used CPLEX 8.0 to compute these numbers. The column also shows the relative gap to the fractional solutions (red numbers). In the last two columns we present the total squared movement during legalization before and after postoptimization. The red numbers are the relative gap to the best lower bound (i.e., the ILP solution or its lower bound in column three).

The experiments show that on this small instance the gap between the fractional LP solution and the ILP solution is between 4.3% and 6.2% if the placement density is at least 40%. This gap does not seem to increase with growing placement density, so at least on this example it is not the reason why our placement results differ more from the LP solution when the density is higher. On this window even the gap between the movement in legalization and (the approximations of) the ILP solution increases with growing density. Nevertheless, this is only a very small instance and it is not clear if this is general phenomenon. Note that on this window the legalization results are much worse than on the whole chip. Here we will not analyze this effect in detail, since due to routability restrictions and because free space may be needed afterwards for inserting additional buffers and inverters in timing optimization, the designers often do not want to use very high densities in global placement. This is different on benchmark instances where only netlength is important, but we are more interested in real-world industrial design than on benchmarks.

The experimental results presented in Table 6.4 and Table 6.5 demonstrate that our postoptimization often does not improve the average movement very much. Nevertheless, it has a significant effect even if the average movement is hardly changed as it helps to reduce the largest movements. Figure 6.8 illustrates this for the chip James. In each of the three diagrams, the x-axis is labeled with distances (in micrometers). The height of the bars is proportional to the number of circuits whose movement is in the corresponding range. Figure 6.8(a) shows the movement distribution after legalization (without postoptimization). The diagrams (b) and (c) allow a closer look at the circuits that have been moved over a big distance where (b) illustrates the movement before and (c) after postoptimization. Although the largest movement could not be improved in this instance, the diagrams show that the number of circuits with large movements has been reduced by postoptimization.

6.10.6 Netlength

In Table 6.7, we analyze how netlength changes during legalization. The instances for these experiments were produced by a global placement run Columns two and three contain the sums of the bounding-box netlength before and after legalization, respectively. The numbers in the last column is the relative increase of netlength during legalization. This increase of netlength due to legalization is less than 0.8 % on average, and only on the two smallest instances, the increase is bigger than 1 %, on all other instances it is at most 0.8 %.

6.10.7 Routability

Figure 6.9 shows routing congestion maps for the chip Nadine, before (a) and after legalization (b). Dark lines in the pictures correspond to routing-critical edges of the global

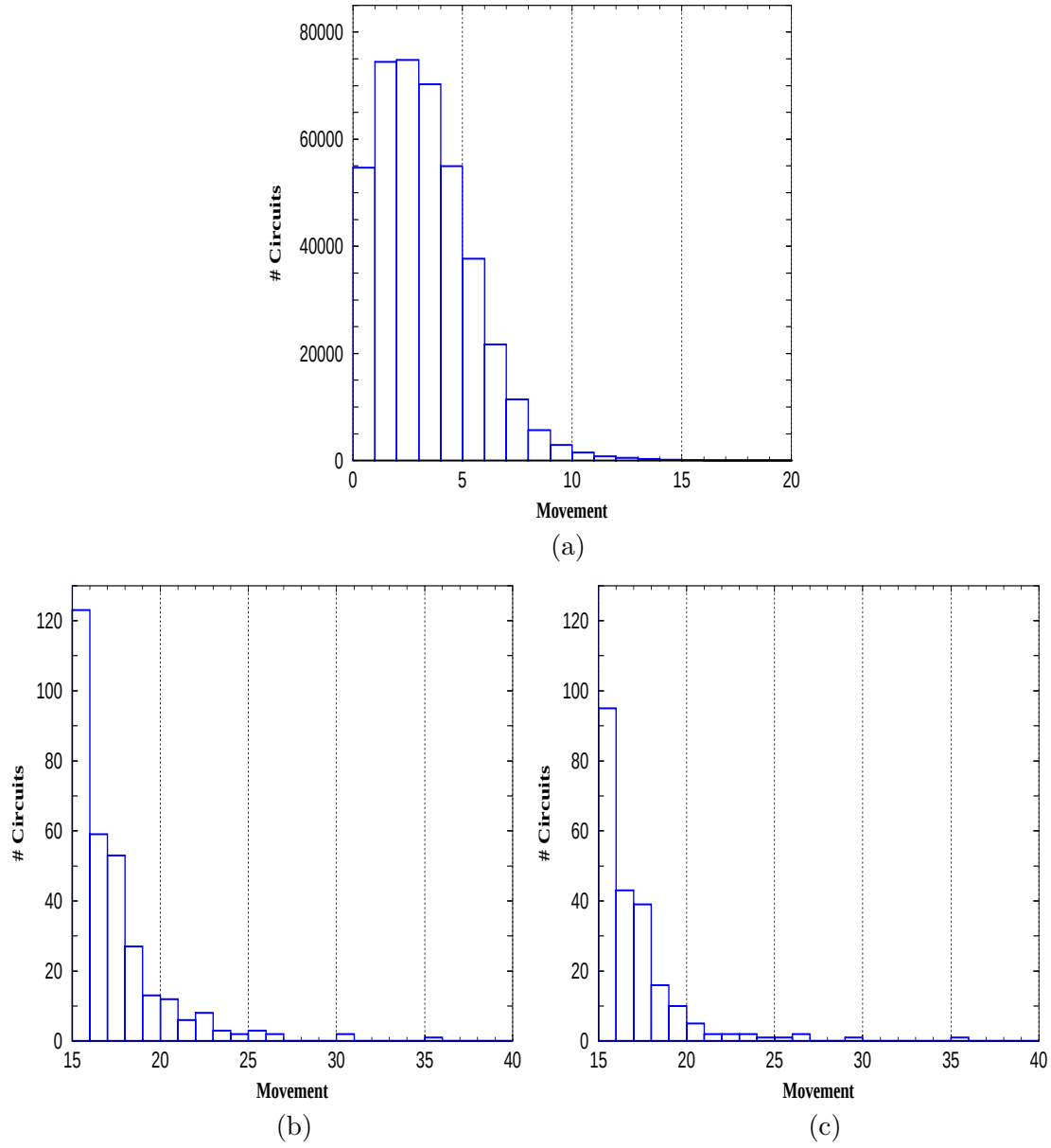
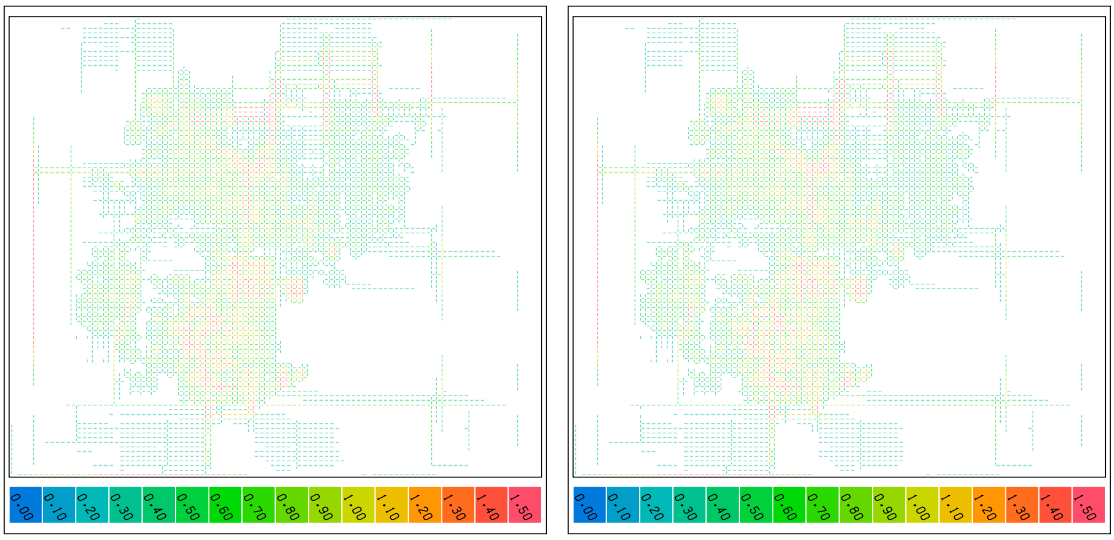


Figure 6.8: Movement histograms for the chip James. (a) and (b) show the movement distribution after legalization but before postoptimization, (c) illustrates the movement after postoptimization.

routing grid. To compute these congestion maps, we applied the congestion estimation that we will describe in Chapter 8, because the current implementation of the more accurate global router proposed by Albrecht [2001] that we will use for other routability experiments cannot handle illegal placements. However, Chapter 8 will demonstrate that the two congestion estimators correlate very well, so for a rough routability check, this kind of experiments is sufficiently precise. Since our legalizer changes the placement only locally it does not create any additional congestion problems.

Chip	BB netlength before legalization	BB netlength after legalization	Increase
Jens	6.42 m	6.56 m	2.2 %
Hans	7.72 m	7.86 m	1.8 %
Christian	184.75 m	184.75 m	0.0 %
James	105.13 m	105.74 m	0.6 %
Aidan	281.08 m	283.38 m	0.8 %
Dieta	208.57 m	209.78 m	0.6 %
Sandra	352.21 m	354.20 m	0.6 %
Josef	297.42 m	299.34 m	0.6 %
Nadine	403.50 m	405.46 m	0.5 %
Wolf	470.36 m	472.67 m	0.5 %
Average			0.8 %

Table 6.7: Effect of legalization on netlength



(a) (b)

Figure 6.9: Global routing plots before (a) and after legalization (b).

6.10.8 Timing

We also considered the impact of our legalization on timing. For these experiments, we ran a full placement (inclusive legalization) and then applied a timing optimization to the design, i.e., we removed and inserted buffers and inverters, changed the size of many circuits and modified the logic locally. The result was a placement where the majority of the circuits was still placed legally while in some regions of the chip there were lots of overlaps. Figure 6.10 gives an impression of the typical effect of our legalization on timing properties. The y-axis in these slack histograms is labeled with slacks (i.e., differences between required arrival times and arrival times) and the length of the horizontal bars

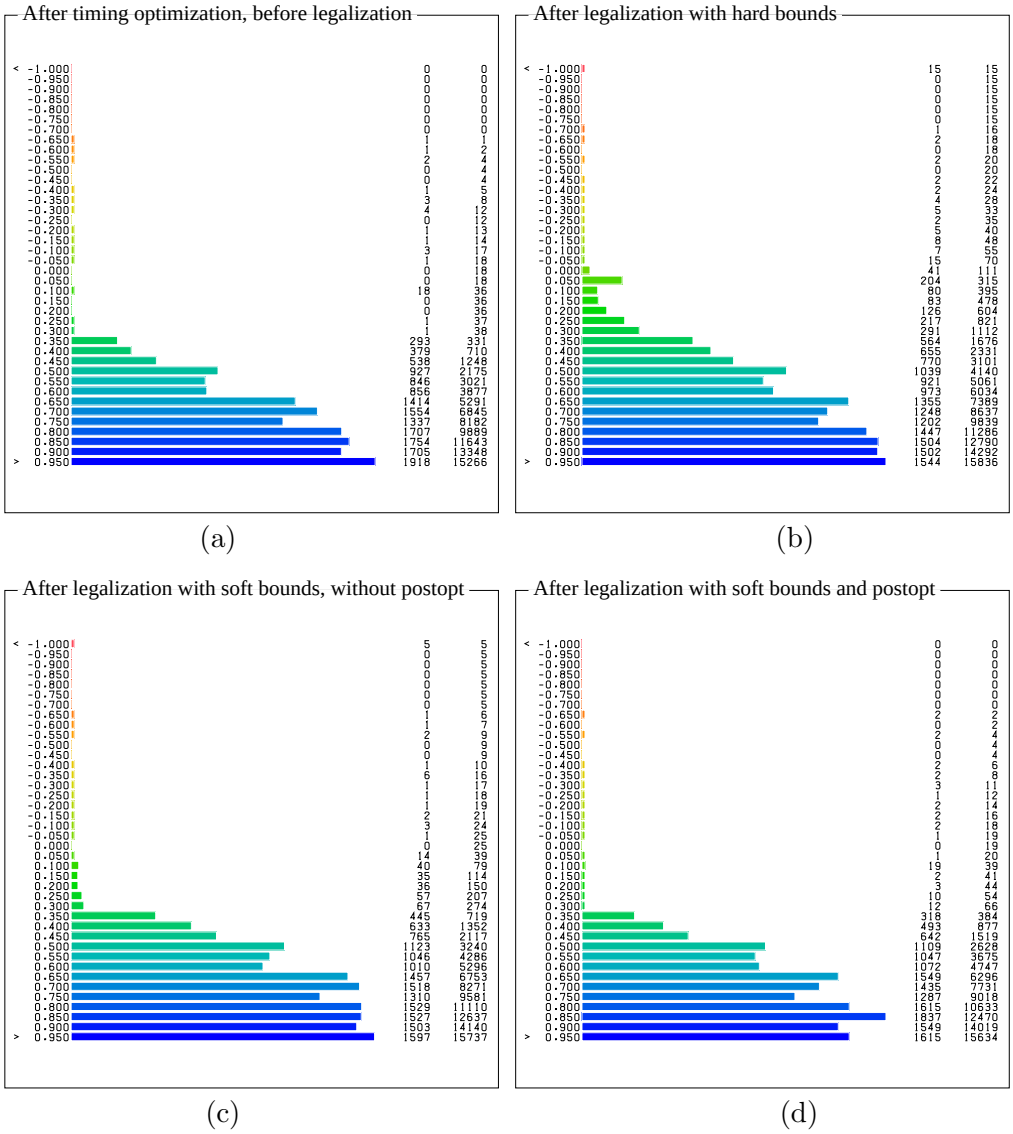


Figure 6.10: Effect of legalization on the slack distribution.

is proportional to the number of pins with this slack. The total number of pins on this chip is about 1.3 million. All slacks are given in nanoseconds. The four histograms show slack distributions for the design after timing optimization (a), after a legalization with hard bounds (b), after the first two phases of our legalizer (c), and after our complete legalization including postoptimization. The pictures show that after timing optimization, most of the pins have a slack larger than 0.3 ns. However, large movements during the legalization with hard bounds increases the number of pins with negative slacks and the number of pins with a slack close to zero (which can cause problems in further layout steps). The number of (nearly) negative pins is much smaller after the legalization with the first two phases of our detailed placement approach. However, only with the help of the postoptimization routine one gets a slack distribution that is almost as good as the

distribution before legalization. We also tested this on several other chips and got very similar distributions. See Brenner and Vygen [2004] and Pauli [2003] for more experiments concerning the interaction of timing optimization and placement legalization.

Chapter 7

Macro Placement

The global placement algorithm presented in Chapter 5 can be used for a simultaneous placement of standard circuits and macros. Though it produces excellent placements if there are only a few movable bigger macros (as we will see in Chapter 9), the results may be quite bad if there are many larger macros to be placed. The reason is that macros will be fixed as soon as they are too big compared to the partitioning windows. Therefore, large macros may be fixed too early. Although most of them should be placed in corners of the chip area (where they do not disturb the rest of the logic), they will probably be placed close to the center. In this chapter, we will show how our global placement approach can be used for a placer that can also handle macros in a reasonable way.

7.1 Overview of the Literature

On most real-world chips there is a huge number of standard cells and quite a small number (normally not more than several hundreds) of macros that have to be placed. Though both for placing only a few rectangles of significantly different sizes and aspect ratios (where the aspect ratio of a rectangle is defined as its widths divided by its height) and for placing lots of standard circuits methods are known that work well in practice, it seems to be much harder to place both groups of circuits together. Therefore, most known approaches either cluster the standard circuits (and small macros) together in order to reduce the number of objects, or partition the larger macros into small fragments in order to get rid of the big rectangles. Sometimes both techniques are combined.

The problem of placing a small number of rectangles in the plane is called Floorplanning Problem. Mostly the minimization of area of a bounding box containing all rectangles is considered as objective function, but the general concepts can also be used for minimizing interconnect length between the rectangle or movement if initial positions are given. In order to compute an optimal floorplan, several papers ask for efficient representations of all relevant arrangements of a given set of rectangles. Onodera, Taniguchi, and Tamaru [1991] represent an arrangement of rectangles by specifying relative positions (top-down, left-right) for each pair of rectangle. There are $4^{\frac{1}{2}n(n-1)}$ such representations, but most of them are redundant or inconsistant. Murata et al. [1995] propose a representation with *sequence*

pairs which leads to $O((n!)^2)$ possible configurations. Guo, Cheng, and Yoshimura [1999] and Takahashi [2000] improved this by using *O-trees* to $O\left(\frac{n!2^{2n-2}}{n^{1.5}}\right)$ combinations. The B^*-*Tree* representation proposed by Chang et al. [2000] also leads to $O\left(\frac{n!2^{2n-2}}{n^{1.5}}\right)$ combinations. These representations can be used for branch-and-bound algorithms or simulated-annealing algorithms. In principle, any objective function can be used in such an environment, but it should be noted that only so-called *admissible* floorplans (see Guo, Cheng, and Yoshimura [1999]) can be represented by sequence pairs, O-trees, or B*-trees. (i.e., no rectangle may be moved to the left or downwards without overlapping other rectangles), so if one wants to minimize interconnect length or movement these two represenations cannot be used directly.

Adya and Markov [2002] relax the macro placement problem by "shredding" macros into small pieces that can be handled by a standard cell placer. The parts of each macro are connected by artificial nets in order to keep them together during the placement process. The authors use the min-cut based placer Capo (Caldwell, Kahng, and Markov [2000]) to place the shredded circuit list. Then, each macro is placed at the center of gravity of its fragments. As this placement will normally contain overlaps, a legalization step for the macros is necessary. To this end, Adya and Markov [2002] cluster the standard circuits and place the clusters together with the macros by a floorplanner based on simulated annealing (Adya and Markov [2001]). The positions of the centers of gravity only serve as an input for the floorplaner, the method does not try to find a solution with minimum movement. The drawback of this method is that the final placement will most likely be very different from the placement of the shredded circuits, so the question is why one runs the placement on the shredded circuits at all. In a newer paper, Adya and Markov [2005] recommend to run some steps of force-directed placement in order to reduce at least the overlaps of macros. However, this method will hardly lead to a legal placement, and even for removing small remaining overlaps, large movements may be necessary.

Force-directed placement approaches (see Johannes and Eisenmann [1998] for the general strategy and Mo, Tabbara, and Brayton [2000] for some special adaptions to macro placement) can place circuits of very different size directly, but, as mentioned in Chapter 3, the relative positions of the circuits will be fixed too early. They may be changed by local optimization steps (see Vorwerk, Kennings, and Vannelli [2004]), but the relative positions of two large macros can hardly be changed without perturbing the whole placement. In addition, the placement will not be legal after the end of the force-directed loop, and in the presence of big macros it may be necessary to move some circuits quite far in order to make it legal.

7.2 Our approach

The macro placement algorithm that we propose makes use of the global placement algorithm desribed in Chapter 5. Since the global placement algorithm is strong in handling small circuits, a main ingredient of our macro placer consists of shredding bigger macros. We place the macro parts and use their positions as a guideline for the macro locations. However, for very large macros, this method will not lead to good results because after only

a few levels the macro parts will be placed in different windows and, hence, can hardly be moved by repartitioning. Experiments have shown that very large macros would be placed too close to the center of the chip area, so we will propose a different method for them. In general, we consider three classes of macros (mainly separated according to their size) that are handled in different ways. Here is an overview of the strategy of how we place the macros of the different classes:

- **Class 1:** The largest macros are placed in such a way that they just do not disturb the rest of the logic too much. For each of them, some candidate locations are computed (mainly in the corners and on the borders of the chip), and all possible placements of them are enumerated and evaluated with a cost function that takes an estimation for the interconnect length and for the area consumption of the rest of the logic into account. Then, the macros of the group are fixed at their positions.
- **Class 2:** The medium-sized macros are shredded, and then, some levels of our global placer are run (placing the shredded macros and the rest of the circuits). For each macro of this class, the average of the positions of its parts are used as a desired location for it. We will propose a strategy how we can find legal positions for these macros close to their desired locations. For the rest of the process, we will fix the macros there.
- **Class 3:** Finally, we run a complete global placement on the remaining, relatively small macros that are still movable and on the standard circuits. The macros are fixed when they are too big compared to the window size. To compute their locations, we just run a simple greedy heuristic.

The following subsections contain detailed descriptions of the three steps of our macro placer. For an illustration of the method, we will show pictures of intermediate steps for the chip Klaus, an ASIC with about 215 000 standard circuits and 123 macros. Figure 7.1 shows a placement in which the macros have been placed manually by a designer, only the the standard circuits have been placed by BonnPlace.

Figure 7.2 shows what happend on that instance if we simply run BonnPlace without any preplacement of macros: The large macros are clustered near the center, a result that is almost always unacceptable both for timing optimization and routability.

7.2.1 Phase 1: Placing Large Macros

Our placement strategy for the largest macro is based on the observation that their positions are not that important as long as they do not harm the placement of the rest of the circuits, e.g., by forcing parts of circuit groups to be placed on different sides of large macros or by making short connections to IO-pins impossible. In the first phase, we consider macros that cover at least 5 % of the chip area. If there are more than five of them, we only consider the five largest ones. For each of them, we allow nine location candidates, four of them in the corners, four in the middle of the borders, and one in the center of the chip. Then, we enumerate all possible assignments of the macros to the candidate loactions. For each

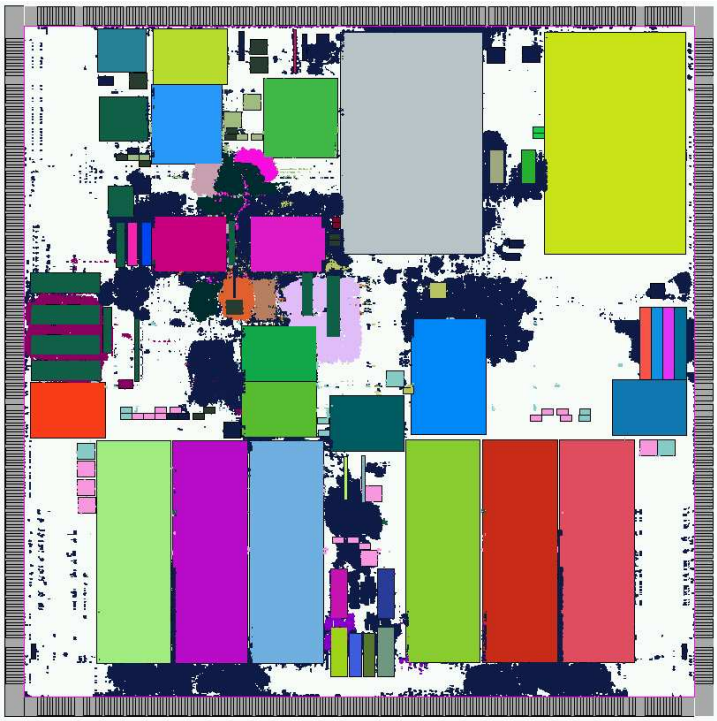


Figure 7.1: A placement of the chip Klaus. The macros have been placed by a designer.

assignment we run over all macros (sorted by size in non-increasing order) and place it at the closest free position to their candidate location (without moving the macros that have already been placed). We evaluate each placement with the following simple cost function: We build one large cluster containing all the remaining circuits and place it as a square such that the total overlap to the placed macros is minimized. Then, we compute the total interconnect length under the assumption that the pins for the large rest cluster are all placed in its center, i.e., we add up the lengths of the interconnections between the macros and between the cluster and the macros and of the connections to preplaced objects and IO-pins. We take the placement that minimizes a weighted sum of an estimation of the overlap costs and the interconnect costs. For the rest of the macro placement process, the macros considered in this phase will be fixed at their position. Figure 7.3 shows the result of the first phase for the chip Klaus.

7.2.2 Phase 2: Placing Medium-Sized Macros

We partition the medium-sized macros into small squares and connect the parts of each macro by dummy nets of high weights (400 times the weight of a standard net in our experiments). Then, they are placed by our global placement algorithm. As we are not interested in a legal placement in this stage but only want an idea were reasonable locations for the macros could be, we only run the first five levels of our placer. Figure 7.4 depicts the placement of the shredded circuit list for the chip Klaus. The two largest macros are marked as fixed by black squares. The picture demonstrates that the six high macros are

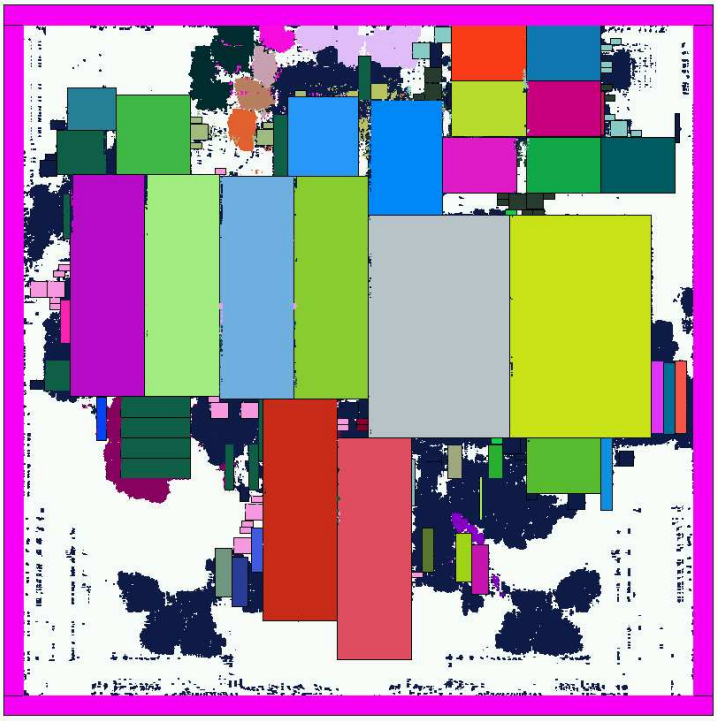


Figure 7.2: The placement for the chip Klaus produced by BonnPlace without macro preprocessing and without preplacing any macros.

on the border of what we can handle with our approach. In the lower left part of the chip area, the fragment sets of some of these macros are mixed in a way that makes it quite hard to decide which of them should be placed where. For the smaller macros, the fragments are placed in much "nicer" shapes.

We compute for each macro the average position of its single parts and ask for a legal placement of the macros minimizing the distance to these average positions, where we choose the costs for moving a circuit to be proportional to the square root of its size. To this end, we legalize the macros in groups of size at most four. To compute these groups, we iteratively take the largest macro that has not been fixed yet and put it together with up to three circuits nearby into a new group. Then, we place each group in one step. For the placement of a group, we make use of the fact that if we fix for each pair of macros (preplaced or movable) their relative positions (so, we introduce the constraint that one of them has to be placed on top or to the left of the other one), a placement that mimizes the bounding-box netlength for any netlist can be computed efficiently by reduction to a minimum cost flow problem (see Vygen [1996]). We run a branch-and-bound algorithm (branching by fixing one of the four different relative positions of two macros) to compute a placement of minimum movement for the circuits of each group. Such a branch-and-bound method has already been proposed by Onodera, Tanuguchi, and Tamaru [1991]. Lower bounds can also be computed using the minimum cost flow formulation. For an upper bound, we initially compute circuit locations by greedily placing the macros in the group, i.e., we traverse the group and place each macro to the nearest free position. Enumerating placements for small groups of macros by a branch-and-bound algorithm seems to be a reasonable compromise

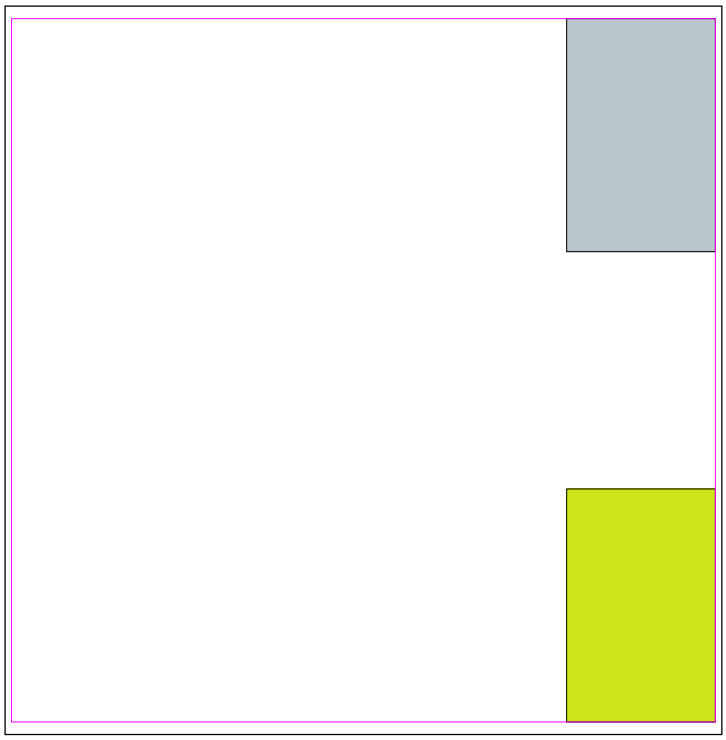


Figure 7.3: The two largest macros of the chip Klaus, placed in phase 1 of our algorithm.

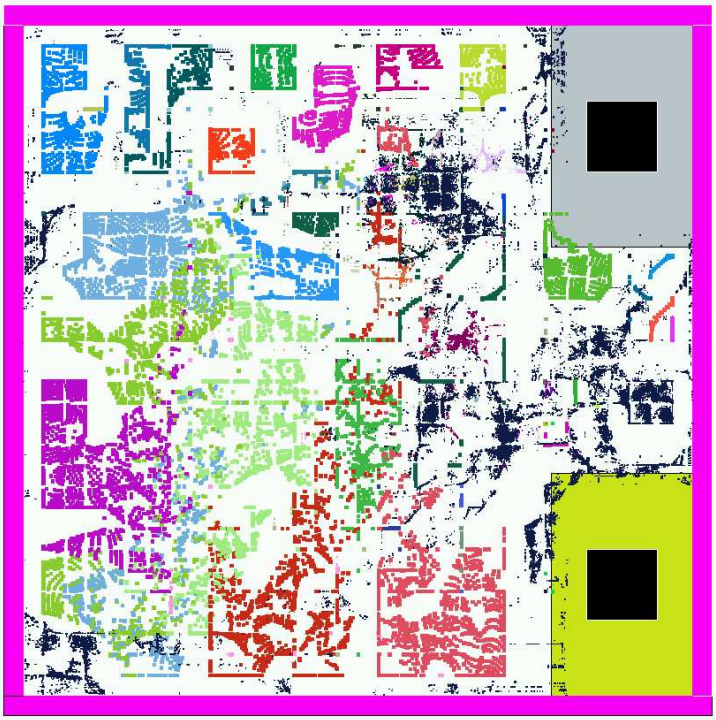


Figure 7.4: The placement of the shredded circuit list of the chip Klaus.

between simple greedy legalization and time-consuming global enumeration. We also allow macros to be several times a member of the group of size four. More precisely, our strategy for finding a legal placement of the macros of class 2 can be described as follows:

MACRO LEGALIZATION ALGORITHM

Input: A set C of circuits.
A placement area A.
A set B of rectangular blockages.
A position $(x_0(c), y_0(c))$ for the center of each circuit $c \in C$.

Output: A legal placement for the macros or the answer that such a placement could not be found.

① Let (x_c, y_c) be the center of the placement area A.
Set $\text{label}(c) := 0$ for all $c \in C$.
Let $c_1, \ldots, c_l$ be an ordering of the elements of C such that $\text{size}(c_i)\left((x_0(c_i) - x_c)^2 + (y_0(c_i) - y_c)^2\right) \geq \text{size}(c_{i+1})\left((x_0(c_{i+1}) - x_c)^2 + (y_0(c_{i+1}) - y_c)^2\right)$ for $i \in \{1, \ldots, l-1\}$.

② IF (there is no $c \in C$ with $\text{label}(c) = 0$)
STOP. // Placement is legal.
Let $i_0 = \min\{j \in \{1, \ldots, l\} \mid \text{label}(c_j) = 0\}$.
Let $\{c \in C \setminus \{c_{i_0}\} \mid \text{label}(c) < 2\} = \{c_1^*, \ldots, c_h^*\}$ such that $\frac{(x(c_i^*) - x(c_{i_0}))^2 + (y(c_i^*) - y(c_{i_0}))^2}{\text{size}(c_i^*)} \leq \frac{(x(c_{i+1}^*) - x(c_{i_0}))^2 + (y(c_{i+1}^*) - y(c_{i_0}))^2}{\text{size}(c_{i+1}^*)}$ for $i = 1, \ldots, h-1$.
Let $i^* = \min\left\{3, \max\left\{i \in \{1, \ldots, h\} \;\middle|\; \frac{(x(c_i^*) - x(c_{i_0}))^2 + (y(c_i^*) - y(c_{i_0}))^2}{\text{size}(c_i^*)} < 9\right\}\right\}$.
Let $C^* := \{c_{i_0}\} \cup \{c_1^*, \ldots, c_{i^*}^*\}$.
FOR ($c \in C^*$ with $\text{label}(c) = 1$)
Remove the rectangle corresponding to c from B.

③ Place the elements of C^* legally and disjointly from the blockages in B by a branch-and-bound method minimizing the distance to the locations $(x_0(c), y_0(c))$. If no legal placement can be found, then stop with the output: "We could not find a legal placement".

④ FOR ($c \in C^*$)
Add the rectangle covered by c to B.
Let c^* be an element of C^* maximizing the value $\text{size}(c^*)\left((x_0(c^*) - x_c)^2 + (y_0(c^*) - y_c)^2\right)$.
Set $\text{label}(c^*) := 2$ and $\text{label}(c) := 1$ for $c \in C^* \setminus \{c^*\}$.
GO TO ②.

Here, $\text{label}(c)$ is either 0 (if c is not placed at all), or 1 (if c is placed but not yet fixed), or 2 (if c is fixed). Only one of the up to four elements of the group C^* is fixed in each pass of the main loop, the other ones can still be moved. The circuits to be placed and fixed are chosen in such a way that we start with large macros near the corners of the chip area. We

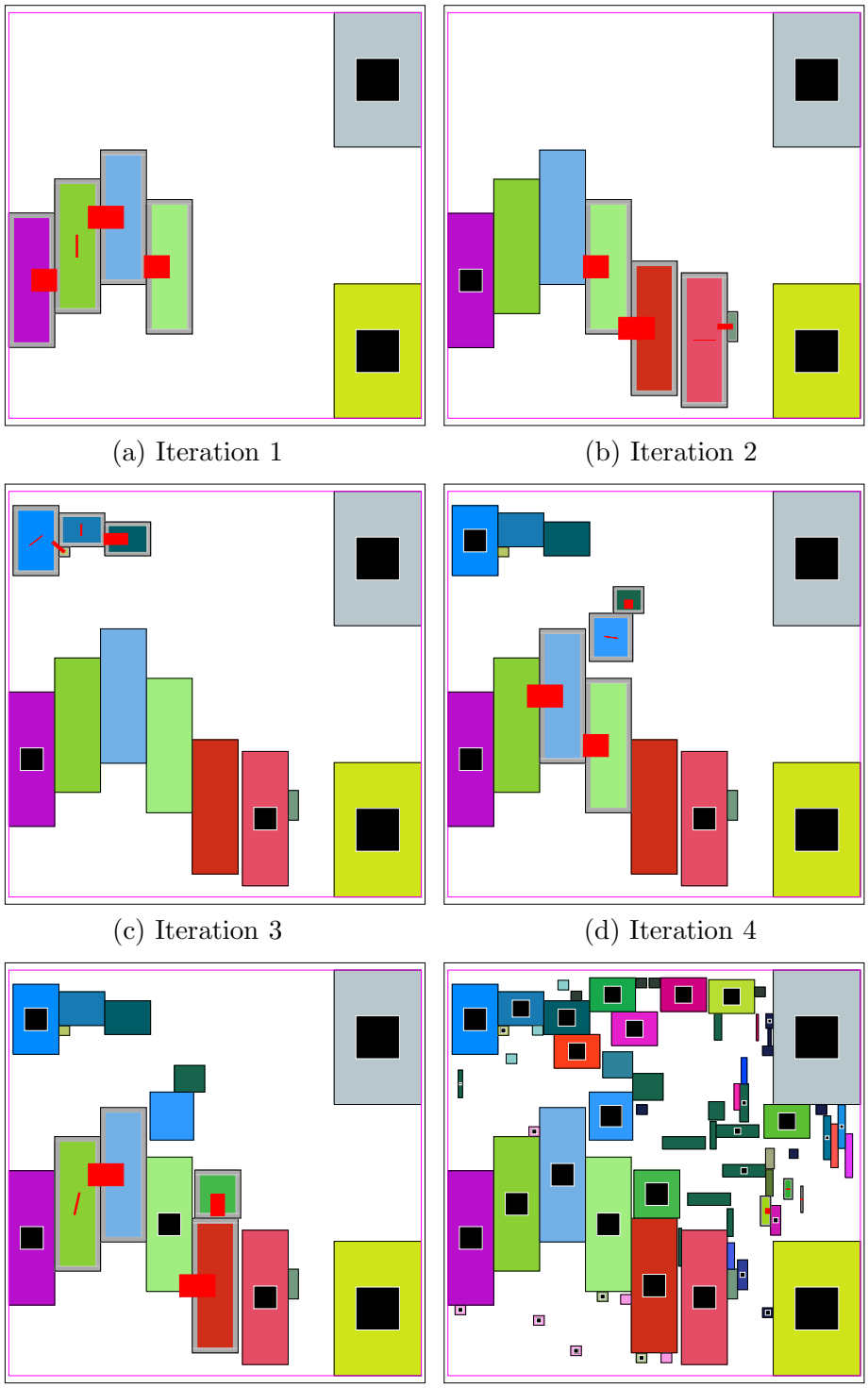
fix one of the macros in each iteration because it will hardly be possible to move very large macros in the final iterations, so we should avoid selecting a large macro several times as a member of the set C^* although it cannot be moved anymore (at least, without moving many other circuits).

Though we bound the number of circuits that we place simultaneously, the branch-and-bound algorithm can sometimes be too time-consuming (for example, if there are many fixed objects around). In order to bound the running time, we therefore do not always run a complete branch-and-bound computation but break off the enumeration if we did not find an optimum solution after a certain amount of time. In that case, we take the best placement we have found so far. Note that the result of the branch-and-bound algorithm is often not yet legal as the macros are generally not placed at grid coordinates. However, since the fixed objects are also aligned the grid coordinates and the macro widths and heights are integer multiples of x_δ or y_δ, respectively, it will most of the times be possible to find legal locations close to the branch-and-bound solution.

Figure 7.5 presents some intermediate pictures of phase 2 for the chip Klaus. The pictures show the macro placement at the end of an iteration of the main loop of the Macro Legalization Algorithm. The elements of C^* for the corresponding iteration are marked by a grey frame. For each $c \in C^*$, its center is connected to its desired position $(x_0(c), y_0(c))$ by a red line. The widths of the lines correspond to the movement costs of the circuits. Note that the placements of the macros with the grey frame are either the result of a branch-and-bound computation (possibly on off-grid coordinates) or the result of an initial greedy placement (with coordinates on the grid) if the branch-and-bound method could not improve this initial placement. Therefore, circuits that are several times member of the group C^* may have slightly different coordinates at the end of the iterations even though their neighbouring macros did not move. All circuits without a grey frame shown in the pictures are placed legally. The macros that have already been fixed at the beginning of the iteration (i.e., the macros that have been preplaced in phase 1 or that have been fixed in earlier iterations of phase 2) are marked by a black square at their centers. After 34 iterations of phase 2, the two macro in class 1 and the 70 macros in class 2 are placed, and they are fixed at their positions for the rest of the macro placement process. The remaining 51 macros of class 3 will be handled in phase 3.

7.2.3 Phase 3: Placing Small Macros

After phase 2, at most the very small macros are still movable. To place these macros, we run in phase 3 our global placer that fixes the remaining macros as soon as they are too big compared to the partition windows. Since they are small enough, they will stay movable during the first levels of the placement and therefore have reasonable positions when they are fixed. One could apply local enumeration techniques as described for phase 2, but at least if the macros are not placed too densely, they can be legalized greedily without causing very large movements. Figure 7.6 shows the placement at the end of phase 3 for the chip Klaus.



(a) Iteration 1 (b) Iteration 2

(c) Iteration 3 (d) Iteration 4

(e) Iteration 5 (f) Iteration 34

Figure 7.5: Snapshots of the macro legalization in phase 2 for the chip Klaus.

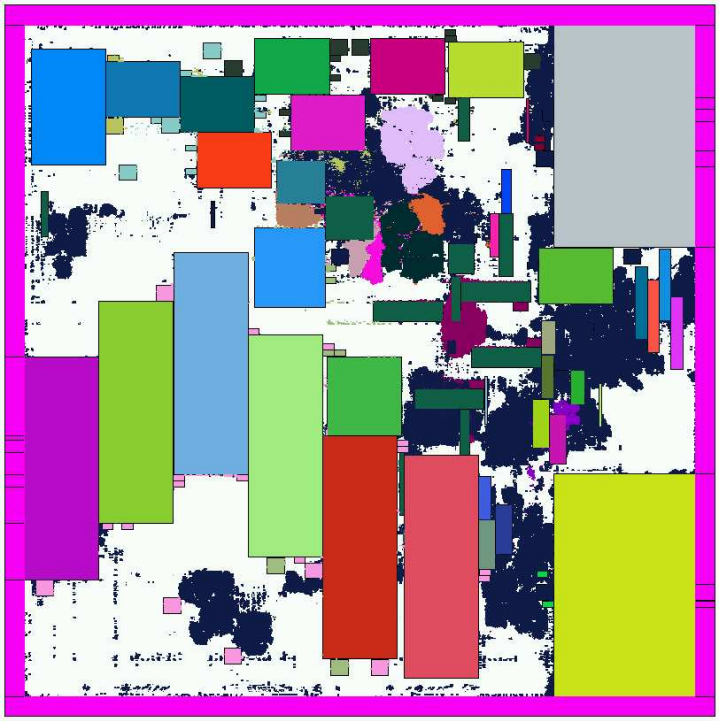


Figure 7.6: The placement of he chip Klaus at the end of our macro placement algorithm.

7.3 Interaction with the Designer

The macro placer proposed in this chapter can be used as a fully automatic tool that can handle placement instances of mixed sizes, and for most of our experiments we will run it in that way. But it should be noted that to get very good results in practice there are still situations when an intervention of the designer is useful or even necessary. Besides the changing of some standard global placement parameters, like the maximum allowed density in a region, the designer can improve the results in the following ways:

- It can be useful to preplace some of the largest macros manually. As phase 1 is only a simple heuristic with many arbitrary decisions, it may be reasonable to preplace macros that would be placed in this phase. This is especially important, if there are large macros for which the assumption that they can be placed anywhere where they do not disturb the rest of the circuits is wrong.

- The designer may predescribe which macros should be placed in phase 1. In some cases, there are macros which are not that big but whose positions are not important, so the size is not necessarily the best way to detect this kind of macros.

- Often, the result of our macro placer will be locally suboptimal. In some cases, it may by necessary to "align" a group of circuits, while in other cases such an alignment may cause routing problems. Similarly, in some situations, it may be desirable to have small slots between macros (for the insertion of buffers or inverters during timing

optimization or in order to place some standard circuits close to a macro they are strongly connected to), while in other cases such slots may cause problems because standard circuits that are placed there may make routing connections over macros necessary.

- The minimization of netlength does not make sense for all kinds of circuits. There are circuits, e.g., de-coupling capacitors (decaps), that must be placed at certain positions near other circuits without being connected to them by nets. Obviously, our macro placer cannot handle this kind of circuits reasonably, so it is up to the designer to insert them afterwards. In order to keep a sufficient amount of free space for them, it is possible to prescribe a rectangular area around each macro that has to stay empty during placement.

7.4 Experiments

We tested our algorithm on several ASICs from our testbed. To compare the quality of different macro placements, we considered the bounding-box netlength that BonnPlace could achieve with the given macro preplacement. Of course, netlength is not the only optimization goal in placement, and a macro placement with very short netlength can be useless if it contains, for example, single regions that are not routable. On the other hand, the experiments are not intended to show that our macro placer can produce excellent macro placements in a fully automatic way, but they should show that the macro placements found by our algorithm are a reasonable starting point for a designer to work with. Then, as mentioned above, in most cases some local changes have to be done manually. But to show that our placements are useful for that purpose, it seems to be sufficient to consider netlength and to have a look at the picture of the placements.

For a comparison, we used BonnPlace without our macro preprocessing, so all macros are floating until they are too big compared to the sizes of the partitioning windows. In addition, we compared our results to macro placements that were found by experienced designers. On some chips, not all of the macros have been preplaced, but at least the bigger ones have been fixed. The remaining smaller macros have been placed by our global placer.

In our experiments, we distinguish two sorts of chips depending on the positions of the IO-pins. On the first group of chips, the *wire-bond chips*, the IO-pins are placed outside the area that can be used for placement at the border of the chip area (*boundary-IOs*). On the other chips, the *flip-chips*, the IO-pins are spread over the placement area (*area-IOs*). The area-IO-pins create some additional difficulties for macro placement. They are placed in a gridlike structure on the chip area, and close to each IO-pin a special IO-circuit, that is often realized by a macro, has to be placed. The IO-circuit is connected to the IO-pin by a two-terminal net whose resistance may not be larger than a prescribed value. Hence, for each IO-circuit, only a very restricted area is allowed. We can handle this restriction in our placer by introducing movebounds for each IO-circuit (and, in phase 2, for each fragment of an IO-circuit). In the partitioning steps, we drastically increase the costs for moving a circuit into a window that does not intersect its allowed area. However, we cannot guarantee that we meet *all* the movebounds, so this is another situation when our macro

placement has to be corrected by a designer. Another problem with area-IOs is the fact that it is mostly necessary to place a decap circuit next to each IO-circuit. As mentioned above, we cannot handle such circuits directly with our approach, but if these decaps are macros, we can block some free area next to each IO-circuit, where the designer can afterwards insert the decaps.

Table 7.1 and Table 7.2 summarize the results of our macro placement experiments. We ran our experiments on an IBM 650 with 8 processors of 1.45 GHz. All running times are wallclock running times for parallel runs on four processors.

Table 7.1 presents the results for the chips with boundary-IOs. For each chip, we report the total number of all circuits, except the IO-circuits, and the number of all macros in the different classes. The last six columns contain the results of our experiments. Columns six and seven show the bounding-box netlengths and the running times of our experiments with manually preplaced macros. On some chips, we did not have a preplacement for all macros. In that case, we first ran a complete placement in order to place the remaining macros, and then, we used this placement as an input for a second placement run with all macros fixed. So, for these chips, two running times are given in column seven, and the netlength in the table refers to the result of the second run. For the experiments with movable macros but without our macro placing algorithm, we always ran two placements, where the first one was only intended to compute a macro placement. Column eight shows the netlength after the second placement run, and column nine the running times for the two runs. Columns ten and eleven contain the corresponding numbers for experiments with our macro placement algorithm, so here the first run contains all three phases of the macro placer.

Table 7.2 gives an overview of our experiments on chips with area-IOs. The table has a structure similar to Table 7.1, but it contains one additional column for each set of experiments ("IO vio's"). This columns report the fraction of IO-circuits for which the predefined IO-netbounds could not be met. Note that the chip Jens, which is in fact only a part of a larger chip, does not have any IO-circuits at all. As we did not always have reliable information on the appropriate IO-netbounds for the chips, we tried to choose the bounds rather too restrictive than too weak.

On the chips Hanno, Yvonne, Alex, and Fermi, we had to block space for decap circuits next to the IO-circuits. Though some sets of IO-circuits shared a common decap circuit in the placement that has been created by a designer, we blocked this space next to each single IO-circuit because it was not clear in which cases it would be possible to avoid the decaps. Hence, we overestimated the amount of area needed for IO-circuits. Of course, these additional constraints made it more difficult to optimize netlength and to meet the IO-netbounds. On Hanno, for example, the fraction of the chip area covered by all circuits was 69.3% with the manual macro placement (including IO-circuits and decaps) but 81.2% with the additional blocks for decaps.

We sum up the results of our experiments:

- On all chips, our macro placer created a better netlength than the runs of BonnPlace without macro preplacement. On 14 of the 17 chips, we could also improve the netlength compared to the manual placements.

Chip	# Circuits	# Macros			Old BonnPlace				BonnPlace with macro preprocessing	
					Preplaced macros		Movable macros			
		Class 1	Class 2	Class 3	BB	Time	BB	Time	BB	Time
Klaus	214 865	2	70	51	44.08 m	30:21 ———	44.23 m	36:17 30:20	37.08 m	49:51 12:18
James	412 050	4	47	73	109.82 m	29:47 30:36	121.60 m	36:57 29:51	95.58 m	2:19:16 44:16
Max	521 375	0	57	0	77.79 m	35:08 32:48	84.19 m	41:09 31:03	75.93 m	2:03:11 30:53
Katrin	763 484	0	534	116	168.19 m	1:04:10 1:12:52	174.13 m	1:09:06 1:10:17	161.30 m	7:54:34 1:01:40
Dagmar	904 756	0	104	11	181.86 m	1:02:39 1:03:09	202.02 m	1:38:41 1:23:51	168.15 m	5:11:35 1:06:53
Sandra	1 317 488	0	194	71	392.79 m	2:31:11 ———	423.54 m	2:47:39 2:46:04	400.06 m	8:40:03 2:04:39
Kevin	1 497 709	0	288	26	153.96 m	2:29:10 ———	140.84 m	2:58:23 2:01:37	138.78 m	12:20:43 2:09:53
Hardy	2 056 728	0	135	136	372.90 m	2:39:02 3:10:22	374.72 m	2:35:40 3:04:00	364.26 m	10:12:52 2:35:41

Table 7.1: Macro placement results on chips with boundary IOs.

Chip	# Circuits	# Macros			Standard BonnPlace						BonnPlace with macro preprocessing		
					Preplaced macros			Movable macros					
		Class 1	Class 2	Class 3	BB (m)	Time	IO vio's	BB (m)	Time	IO vio's	BB (m)	Time	IO vio's
Jens	72 496	5	9	0	7.19	3:55 ———	———	8.12	4:31 5:13	———	6.74	11:03 3:44	———
Hartmut	158 802	0	95	642	145.14	20:36 ———	0.0 %	142.06	21:29 18:54	0.2 %	136.35	32:51 11:39	0.0 %
Christian	278 083	0	52	702	126.05	21:52 ———	0.0 %	134.28	21:18 16:26	2.2 %	125.07	1:02:22 15:00	0.1 %
Hanno	779 033	0	341	629	104.60	1:24:43 ———	0.0 %	121.09	3:08:02 1:10:33	10.6 %	116.12	5:50:34 59:50	5.9 %
Sven	825 737	0	271	1 359	238.29	2:14:31 ———	0.2 %	231.03	1:58:02 1:19:01	12.8 %	195.64	8:09:52 1:18:39	2.2 %
Yvonne	915 086	0	270	812	277.62	2:10:19 ———	0.0 %	278.10	1:50:46 2:17:08	3.0 %	269.09	6:55:31 1:35:30	1.1 %
Alex	971 113	0	844	1 672	159.10	1:48:09 ———	0.0 %	155.21	2:04:01 1:27:11	0.6 %	148.84	10:44:18 1:23:52	0.3 %
Josef	1 349 390	0	287	677	169.09	1:46:27 1:42:52	0.0 %	176.32	2:02:16 1:42:05	7.1 %	160.58	7:38:32 1:42:44	4.5 %
Fermi	3 605 741	0	215	1 181	363.76	6:12:46 ———	4.7 %	412.69	6:34:42 6:07:38	4.7 %	396.06	18:59:57 6:13:22	6.1 %

Table 7.2: Macro placement results on chips with area IOs.

- On the chips with boundary-IOs, the netlength of the placements computed with our macro placer is, on average 6.8 % smaller than the netlength of the placements with manual macro preplacements (computed by the geometric mean). On the chips with area-IOs, the corresponding number is 3.2 %. On all chips the average improvement is 4.9%.
- Compared to the runs of BonnPlace without macro preprocessing, we can even improve the netlengths by 10.4% (boundary IOs) or 7.6% (area IOs), on average. The averge netlength reduction on all chips is 9.0%.

The running time for the three phases of our macro placer are up to eight times bigger than the running times of the following placement with fixed macros. As especially the running time of phase 2 depends not only on the number of all circuits but on the number and the desired positions of the class 2 macros, this factor varies quite strongly with the different chips. Table 7.3 shows the running times for the different parts of the program. The running time for phase 2 is split into the time for the five levels of global placement on the shredded circuit list (I) and the time for the Macro Legalization Algorithm (II). For comparison, the running times of a standrd BonnPlace run with fixed macros is shown in the last column of the table.

Chip	Phase 1	Phase 2		Phase 3	Standard BonnPlace
		I	II		
Jens	45	6:02	15	4:01	3:44
Hartmut	———	19:39	1:25	10:50	11:39
Klaus	1	31:08	4:31	13:40	12:18
Christian	———	45:59	21	13:59	15:00
James	1:41	1:33:43	4:43	38:09	44:16
Max	———	1:26:28	6:08	27:11	30:53
Katrin	———	2:01:55	4:49:19	53:56	1:01:40
Hanno	———	1:41:49	3:06:36	49:39	59:50
Sven	———	4:26:56	2:12:36	1:16:03	1:18:39
Dagmar	———	3:09:04	14:42	1:43:48	1:06:53
Yvonne	———	2:15:14	3:05:30	1:32:53	1:35:30
Alex	———	3:05:45	6:13:38	1:14:53	1:23:52
Sandra	———	5:20:05	1:19:49	1:50:55	2:04:39
Josef	———	3:52:37	2:09:07	1:26:18	1:42:44
Kevin	———	6:00:08	4:08:07	1:59:44	2:09:53
Hardy	———	7:10:24	37:28	2:13:34	2:35:41
Fermi	———	11:13:45	1:14:21	6:22:50	6:13:22

Table 7.3: Running times for macro placement.

To illustrate our experiments, we show some typical outcomes of the three different macro placements that we considered for each chip:

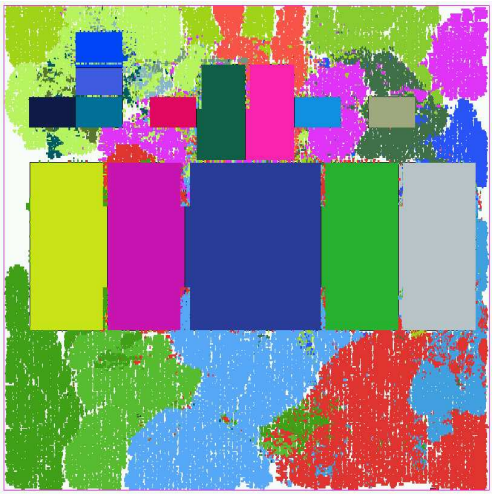
- Figure 7.7 presents the placements of the chip Jens. Again, the standard BonnPlace run with floating macros (a) placed the five large macros simply in the center

which makes the whole placement useless. Note that the placement of the five biggest macros found by phase 1 of our algorithm (picture (c)) is very similar to the manual placement (b) after mirroring on a horizontal axis. Such a mirroring can be done on Jens without changing the netlength significantly because the chip has only a few IO-pins which are located near the center of the chip area.

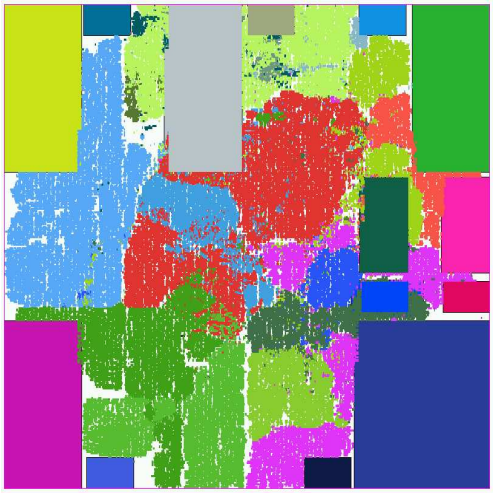
- Figure 7.8 gives an overview of the results for the chip James. Here, again, the five largest macros have been fixed in phase 1. The placements (b) and (c) are completely different but it should be noted that the netlength of placement (c) is 13% smaller.

- The chip Yvonne, shown in Figure 7.9, is an example for a chip on which much space has to be blocked for decaps. On the three pictures, one can easily see the free area around the five macros near the border and next to each of the (grey) IO-circuits that are spread over the outer regions of the chip area. Though the netlengths of placements (a) and (b) are almost equal, the dense packing of macros in placement (a) would probably cause routing and/or timing problems later in the design process.

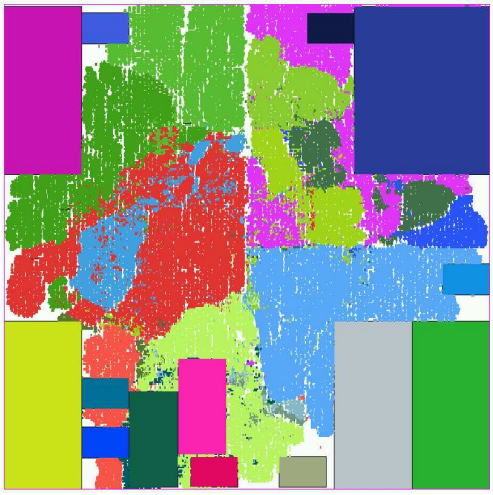
- Sandra (see Figure 7.10) is one of the chips on which the result of our macro placer was (in terms of netlength) worse than the manual preplacement (by about 1.9 %). Some of the local problems that may cause this difference can clearly be seen in placement (c), e.g., around the red macro in the upper right corner where a group of logic is divided in parts by that macro.



(a) Macros placed by a standard BonnPlace run

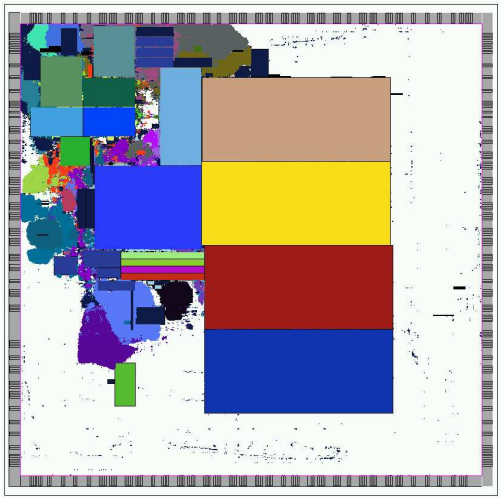


(b) Macros placed by a designer

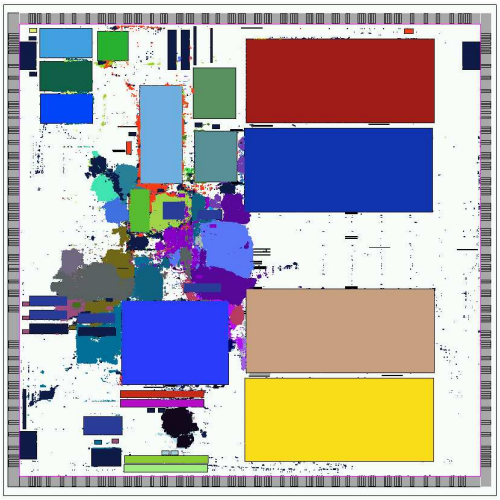


(c) Macros placed by our algorithm

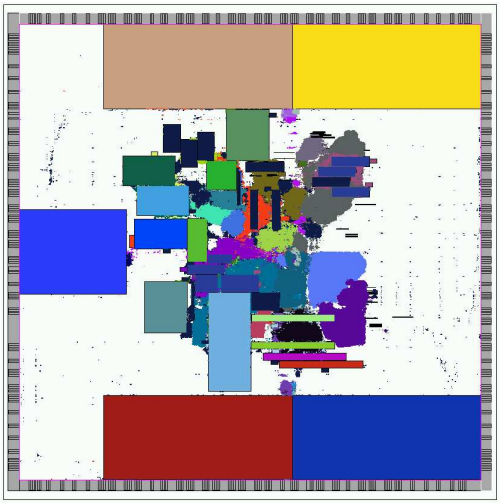
Figure 7.7: Macro placements for the chip Jens.



(a) Macros placed by a standard BonnPlace run

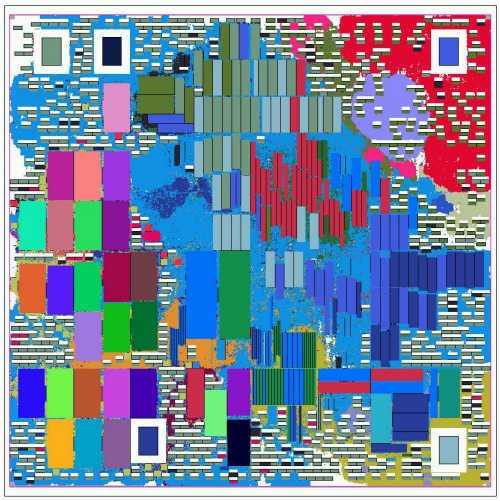


(b) Macros placed by a designer

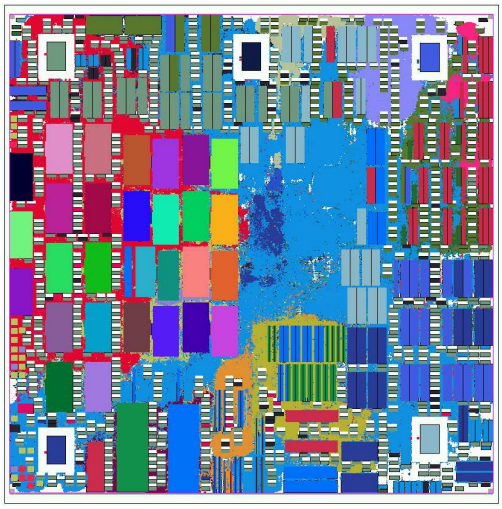


(c) Macros placed by our algorithm

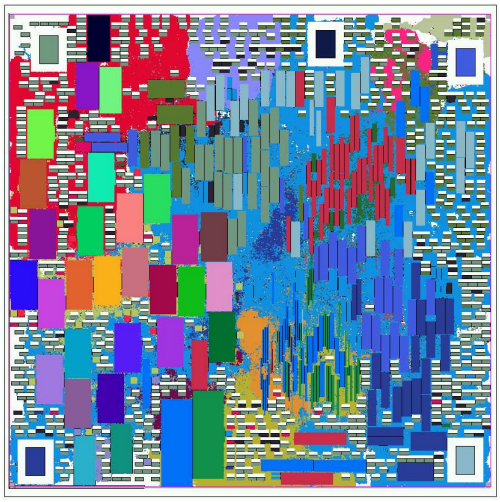
Figure 7.8: Macro placements for the chip James.



(a) Macros placed by a standard BonnPlace run

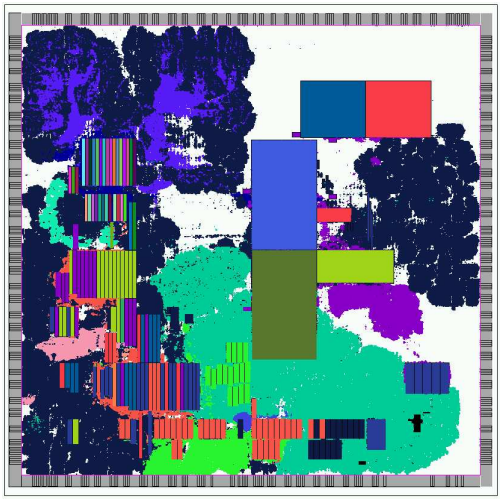


(b) Macros placed by a designer

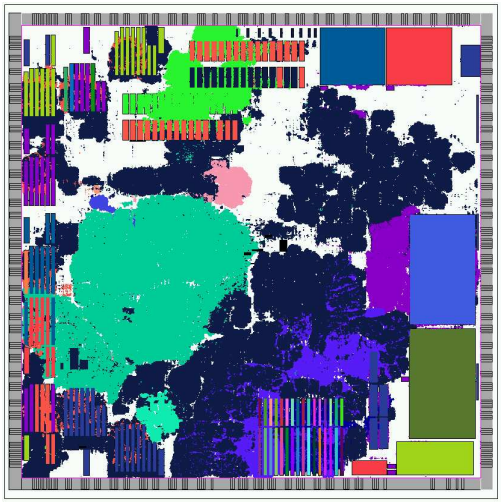


(c) Macros placed by our algorithm

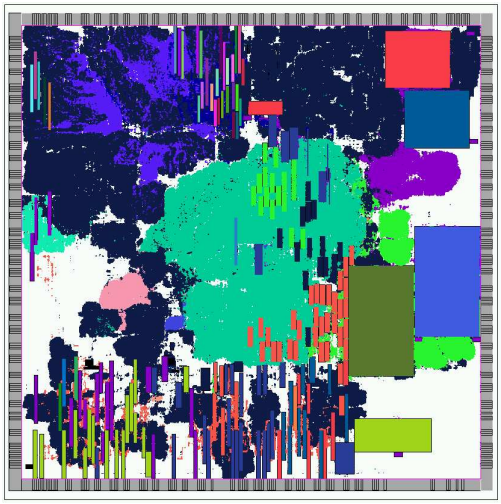
Figure 7.9: Macro placements for the chip Yvonne.



(a) Macros placed by a standard BonnPlace run



(b) Macros placed by a designer



(c) Macros placed by our algorithm

Figure 7.10: Macro placements for the chip Sandra.

Chapter 8

Congestion-Driven Placement

In the global placement algorithm described in Chapter 5, we considered minimization of weighted netlength as the main optimization goal. We have already mentioned that timing optimization can be incorporated into this approach by defining appropriate weights for nets on timing-critical paths. For routability, a short netlength is necessary, too. However, in placements with very short netlength, there are often groups of strongly connected circuits that are placed very densely. These groups can create local congestion that makes large routing detours necessary or even makes routing impossible. To avoid such problems, the placement density in the routing-critical parts of the chip has to be reduced. In BonnPlace, this can be done manually by defining a lower density for regions or for groups of circuits. However, since one does not want to rerun the complete placement when a routability problem occurs, it is desirable to detect and to solve possible routing problems during a single placement run. In this chapter, we describe a very fast method that detects accurately routing criticalities and reduces automatically the placement density in the critical regions. We incorporated the method into our placer BonnPlace and we will show by experiments that routability could be improved significantly. By using a lower density only in the critical parts of the chip, we can increase the allowed density in other regions and hence get better results in terms of (routed) wirelength. It should be noted that the method can be incorporated into any placement algorithm that is based on top-down recursive partitioning. Our congestion estimation can be applied to any given placement, even if it is not yet legal, so the part of the algorithm that analyses the congestion can be used in almost any placement algorithm.

8.1 Previous Approaches

In the last couple of years several new approaches have been developed in order to take congestion during placement into consideration. Many algorithms use a very simple version of a probabilistic global router to estimate the congestion in a region: The chip area is divided into small tiles, for each border of a tile the expected number of wires routed through this border is compared to the number of free routing tracks that cross the border. For example, Cheng [1994], Hung and Flynn [1997], Wang et al. [2000], Hou et al. [2001],

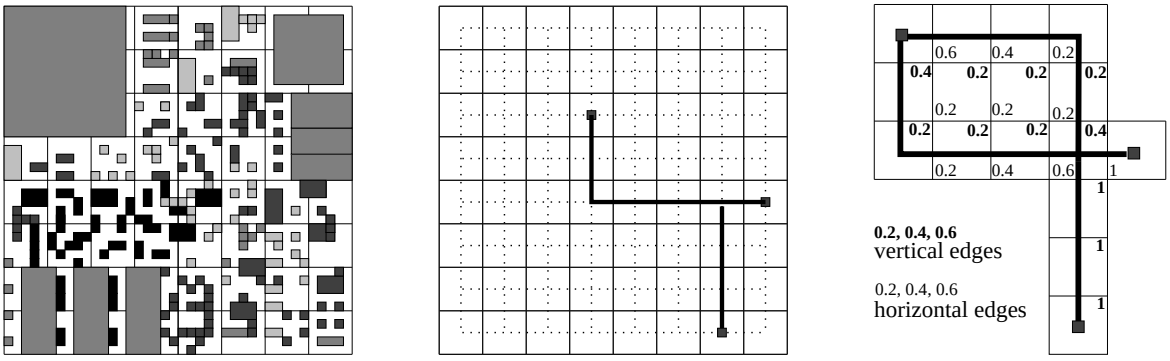
Lou, Krishnamoorthy, and Sheng [2002], Yang, Kastner, and Sarrafzadeh [2001], Kahng and Xu [2003], Yang, Choi, and Sarrafzadeh [2003], and Chang et al. [2003] follow this idea. The algorithms differ mainly in the way they handle multi-terminal nets and blockages and in their probabilistic distributions for the interconnections. A probabilistic routing estimator of this type will also be an important part of our congestion estimation. Other methods to detect congestion are described by Yang, Kastner, and Sarrafzadeh [2002] and Hu and Marek-Sadowska [2002], where Rent's Rule is used to estimate the peak congestion value. Rent's Rule is an empirical observation concerning the relation between the number of circuits and the number of external connections of a part of a chip (see Landman and and Russo [1971]).

Most authors describe not only ideas to detect congestion, but also propose ways to reduce it. Often congestion reduction is done in detailed placement or in a post-placement-optimization (see Wang and Sarrafzadeh [1999, 2000], Wang et al. [2000], Wang, Yang, and Sarrafzadeh [2000a], and Yang, Kastner, and Sarrafzadeh [2001]). Other authors incorporate the goal of congestion reduction into the global placement: Parakh, Brown, and Sakallah [1998] show how quadratic placement can be modified in order to avoid routing problems; Mayrhofer and Lauther [1990] describe a partitioning approach that works similar to min-cut partitioning, but has minimization of congestion as a goal. Xiu et al. [2004] incorporate minimization of congestion on edges between the partitioning windows into their objective function for partitioning. The partitioning-based algorithm in Hou et al. [2001] increases the estimated area consumption after a global placement for circuits in congested regions and then repeats the last levels of the partitioning while considering these modified circuit sizes. There have also been experiments where congestion is used as an optimization function for simulated annealing (see Cheng [1994], Wang and Sarrafzadeh [1999], and Wang, Yang, and Sarrafzadeh [2000a]), but the results of Wang and Sarrafzadeh [1999] and Wang, Yang, and Sarrafzadeh [2000a] imply that this approach will hardly lead to an improvement of routability. The authors achieved the best congestion results when they used netlength as objective function.

8.2 Congestion Analysis

In this section, we will describe a measurement for the congestion of a placement as it appears during the placement algorithm. Given a chip which is partitioned into $k \times k$ regions (forming a $(k+1) \times (k+1)$-placement-grid), we calculate a pin density for each region and a congestion estimation for each edge in the dual graph of the placement grid.

The core part of our algorithm is an estimation for the global routing of the current placement. Global routing tools divide the chip into a number of tiles (bins) and calculate a congestion estimation for the chip as well as a rough topology for each net. The main drawback of existing global routing tools is the running time. A global router using sophisticated methods based on multicommodity flows (Albrecht [2001]) might take several hours on a chip with millions of nets. Though just a congestion estimation with a global router is faster, it is in general too slow because we have to call such an estimator several times during the placement as a subroutine. Our simplified probabilistic global router tries to imitate a real router in the following way:



(a) placement grid (b) Steiner tree and routing grid (c) probabilistic weights

Figure 8.1: A chip with the placement grid and its dual graph and the expectation values for the routing of a net.

(a) The current status of the placement grid is used as the global routing partition. The global routing grid is defined as the dual of this partition (See, for example, Figure 8.1(a) that shows the placement grid, the dotted segments in Figure 8.1(b) show the routing grid).

(b) Multi-terminal nets are split into a set of two-terminal nets: We calculate a Steiner tree for all nets $N \in \mathcal{N}$, each connection between two pins or Steiner points in this Steiner tree is treated as a separate two-point connection (see Figure 8.1(b)).

(c) For the two-point connections in the Steiner tree of net N, we calculate the probability $p_N(e)$ of each edge e in the dual graph of the placement grid to be used in the routing of this connection. This method is similar to the algorithms proposed by Hung and Flynn [1997], Lou, Krishnamoorthy, and Sheng [2001], and Kahng and Xu [2003]. We calculate the set P of all length-optimal paths with at most two bends (vias) between the points. In Hung and Flynn [1997], also shortest connections with arbitrarily many vias are examined (where larger numbers of vias get lower probabilities), but the restriction to two bends models quite well what a real router does. Based on this, $p_n(e)$ is set to the number of paths in P that use e divided by the cardinality of P. In Figure 8.1(c), three two-point connections are considered. The connections from the Steiner point to the right and the bottom terminal are uniquely optimal, we get $p_n(e) = 1$ for these edges. The connection of the left terminal with the Steiner point can be realized by five different paths with at most two vias. The usage probabilities for the vertical routing edges are shown in bold, the other probabilities with the normal font.

(d) For each edge $e \in E(G)$ in the dual of the placement grid, we calculate its expected usage $p(e) := \sum_{N \in \mathcal{N}} p_N(e)$ and the capacity value $cap(e)$. The fraction $cong(e) := \frac{p(e)}{cap(e)}$ is our estimated congestion on e. The capacity $cap(e)$ depends (via a user parameter) on the number of routing channels of the three-dimensional routing grid that cross e. On edges over preplaced macros that block some routing layers, the capacity might by reduced (also controlled by a user parameter). However, according to our experience such a reduction is often unnecessary. In the algorithm, routing problems

are reduced by spreading parts of the standard logic that are close to routing-critical edges. After some steps of a partitioning based placer, we may assume that in the center region of large macros there are no standard circuits any more, so the edges crossing these macros are not that important. Of course, to handle smaller macros or to get a good estimation on all parts of a chip one can reduce the estimated routing capacity on macros.

The measurement described above just considers the congestion on the edges of the (quite coarse) placement grid that is used as a global routing grid. Nets that are completely contained in one tile of the placement partition are not considered at all. In order to take also routability problems inside the tiles into account, we use the pin density *pin-dens*(r) inside a region r as a second measurement. The number *pin-dens*(r) is calculated in a straight-forward way: Given the circuits $C(r)$ inside the region, we divide the total number of pins in $C(r)$ by the number of routing grid nodes in r.

8.3 Usage of Congestion Data

8.3.1 Calculation of Inflation Values

Once we have found a routing-critical region during the placement process, we have to use this knowledge to remove or at least reduce the congestion in this region. We will show how our estimation can be used in the placement tool BonnPlace. In our approach, we follow the strategy of many chip designers and try to use a lowered density for groups of circuits that tend to create routing problems. We handle such groups of circuits by *inflating* them, i.e., we increase their estimated area usage in the partitioning step. This means that we do not use $s(c) := w(c) \cdot h(c)$ as the size of the circuit c in this step, but a number $(1+b(c)) \cdot s(c)$ (with $b(c) \geq 0$). Of course, when partitioning the set $C(r)$ (for a region r) into subsets $C(r_1), \ldots, C(r_k)$, we have to ensure that the condition $\sum_{c \in C(r_i)} (1 + b(c)) \cdot s(c) \leq s(r_i)$ is fulfilled (for $i = 1, \ldots, k$).

The numbers $b(c)$ depend on an input parameter $\tau \geq 0$ that specifies how much we want to allow the algorithm to increase the circuit sizes. Before the placement starts, each circuit c gets an initial value $b(c)$ that is proportional to the number of pins of c divided by $s(c)$. This is motivated by the observation that small circuits with many pins often cause routing problems if they are placed densely. The initial numbers $b(c)$ are scaled with the parameter τ such that $\sum_{c \in C} b(c) \cdot s(c) = \frac{\tau}{4} \sum_{c \in C} s(c)$. This ensures that the total size of the circuits grows initially by a factor of $(1 + \frac{\tau}{4})$. During the placement run, the number $b(c)$ is updated according to the congestion estimation of the region r the circuit c is currently placed in. Let $e_1, \ldots, e_4$ be the four edges of the global routing grid that are incident to r. If *cong*$(e_i) \geq 1$ we increase $b(c)$ by $\max\{0, \min\{1, 2(\textit{cong}(e_i) - 1)\}\} \cdot \frac{\tau}{5}$ (for $i = 1, \ldots, 4$). This way, each of the edges can cause an increment of $b(c)$ by at most $\frac{\tau}{5}$. This maximum is attained for *cong*$(e_i) \geq 1.5$. The number $b(c)$ is also increased by adding a number proportional to *pin-dens*(r) (but at most $\frac{\tau}{5}$), if *pin-dens*(r) is bigger than some fixed threshold value. Therefore, it is guaranteed that τ is an upper bound for the total increment of $b(c)$ in each level, and that the increment due to congested global routing

edges is the dominating factor. A typical value for τ is 0.2.

Moreover, we also decrease the numbers $b(c)$ if *both* the congestion estimation on the routing grid edges and the pin density are far away from critical numbers. In this case, the number subtracted from $b(c)$ is also proportional to τ. This ensures that unnecessarily high values $b(c)$ (especially the inflating values computed in the initialization) can be corrected during the placement run.

8.3.2 Spreading Inflated Circuits

Changing the estimated area usage for circuits during the placement run would not be useful if we could not find a placement that respects these larger circuit sizes. One way to find such a placement is to calculate (but not use) the inflating numbers during a first placement run and then use them in a post-placement optimization or in a new placement run (that may consist only of the last partitioning steps, see Hou et al. [2001]). An important feature in our approach is that we do not have to repeat parts of the placement, but we use the repartitioning method to move circuits out of regions that are too full (with respect to the sizes $(1+b(c))\cdot s(c)$). While the repartitiong method was invented to improve the netlength of a placement, we use it in addition to reduce the number of overcrowded regions. Similar ideas are used in the iterative partitioning strategy in Section 5.2.2. In the additional repartitioning step, we consider $m \times m$-windows with $m \in \{2, 3\}$. As in the repartitioning for the reduction of netlength, choosing $m = 2$ leads to a faster program while setting $m = 3$ produces better results. We collect all these $m \times m$-windows in a heap. Let $\{r_1, \ldots, r_{m^2}\}$ be the set of regions of a $m \times m$-window in the heap, and let $\{C(r_1), \ldots, C(r_{m^2})\}$ be the set of the corresponding circuit sets. Then, the key for this element in the heap is

$$\left(\sum_{i=1}^{m^2} \sum_{c \in C(r_i)} (1 + b(c))s(c)\right) / \left(\sum_{i=1}^{m^2} s(A_i)\right) - \max_{i=1,\ldots,m^2} \left(\sum_{c \in C(r_i)} (1 + b(c))s(c)/s(A_i)\right).$$

Note that all keys are non-positive because the maximum density in a region is never smaller than the average density. To reduce overloads of regions we take a heap element with the smallest key, so we first consider $m \times m$-windows with large overloads in a single region but small average densities. In these windows, the greatest reduction of overloads is possible. As in standard repartitioning, we compute a new placement in the window by solving a QP, computing a partitioning of the circuits, and solving a second QP that forces the cicuits to be placed in their subwindows. The new placement is accepted if the balance of the regions is improved (i.e., the maximum overcrowding is reduced), even if the netlength gets slightly worse. The optimization function is a weighted sum of the maximum overload of a region and the netlength. Of course, after accepting a new placement in a $m \times m$-window, we have to update the keys for the $m \times m$-windows intersecting with it. We repeat this until all overloads are removed or the minimal key of an heap element is larger than a certain constant $-\epsilon$. The method is used as the function Reduce Overloads in the congestion-driven version of BonnPlace. The congestion-driven version of BonnPlace can be described as follows (additional steps compared to a standard BonnPlace run are indicated by a red color):

CONGESTION-DRIVEN BONNPLACE

Input: An instance of the placement problem.

Output: A placement.

① INITIALIZATION:
window_set := $\{r_0\}$;
$C(r_0) := C$;
• FOR (each $c \in C$)
Increase $b(c)$ proportional to #pins(c)/size(c);

② Solve a QP to minimize quadratic net-length;

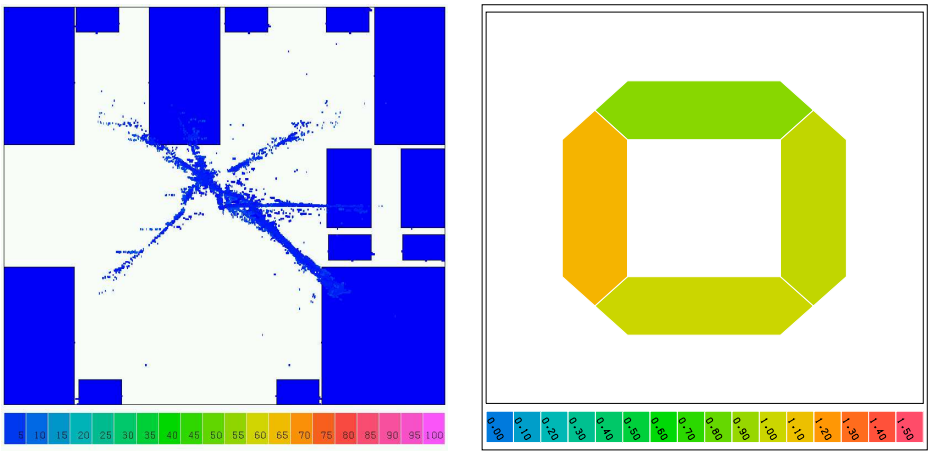
③ WHILE (window size is big enough)
{
FOR (each window r in window_set)
{
Solve constrained QP;
PARTITIONING$(r, C(r))$;
• Compute congestion on the edges of the placement grid;
• FOR (each $c \in C$)
Update $b(c)$ according to local congestion;
• PARTITIONING$(r, C(r))$;
Solve a QP (with terminal propagation at the subwindow borders);
}
• REDUCE OVERLOADS;
REPARTITIONING;
}

④ LEGALIZATION;

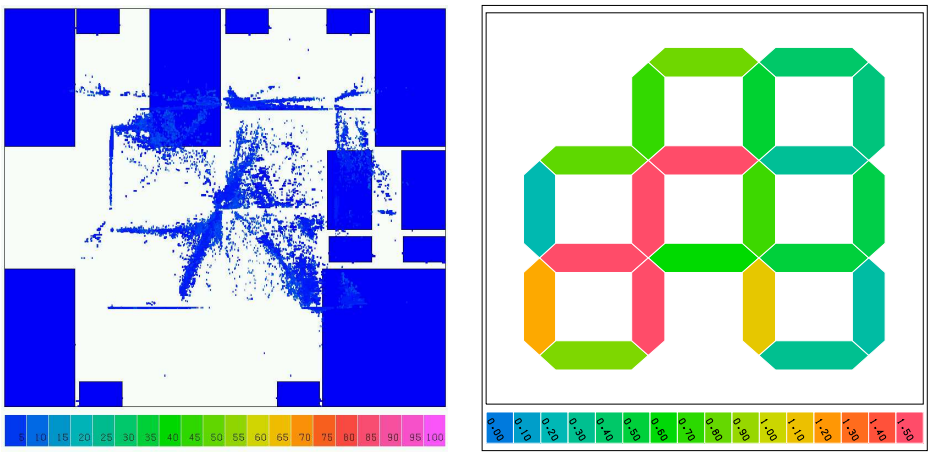
The second PARTITIONING step after the congestion estimation (with respect to the new inflation numbers) also helps to reduce overloads of regions. However, note that it is in general still necessary to call the function REDUCE OVERLOADS.

Since we decrease the estimated area usage for circuits in non-critical regions and we increase the maximal allowed density $d(r)$ for every region r in each placement level a little bit (typically 1% per level), there is normally enough free capacity to move circuits away from crowded regions. Note that the schematic description above is a simplification: apart from the initialization, the inflating values $b(c)$ are changed only during the levels 3 to 7. Experience shows that during earlier levels the congestion estimation does not really provide accurate information, while in the last levels it will hardly be possible to change the placement such that new inflating values could be taken into account.

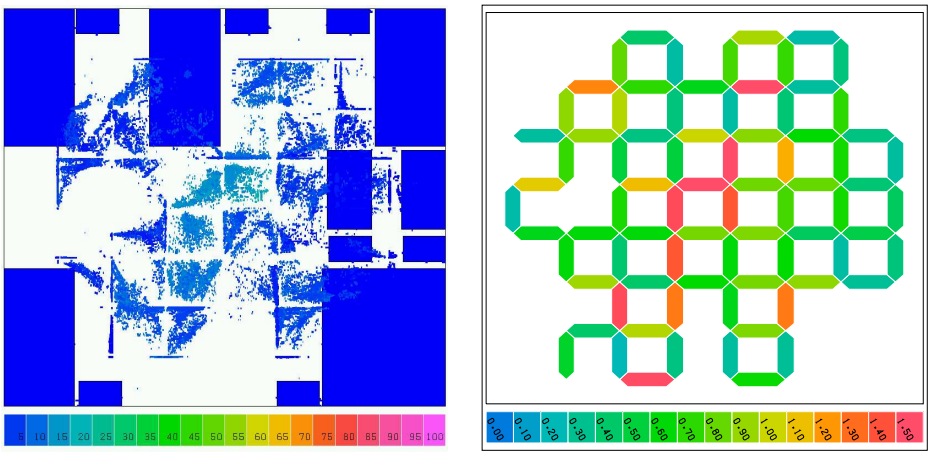
Figures 8.2, 8.3, and 8.4 illustrate a congestion-driven BONNPLACE run on the chip Jens. For each level, the placement and the congestion estimation at the at the end of the level are shown. The circuits are colored according to their inflating values $b(c)$: dark blue



(a) Level 1

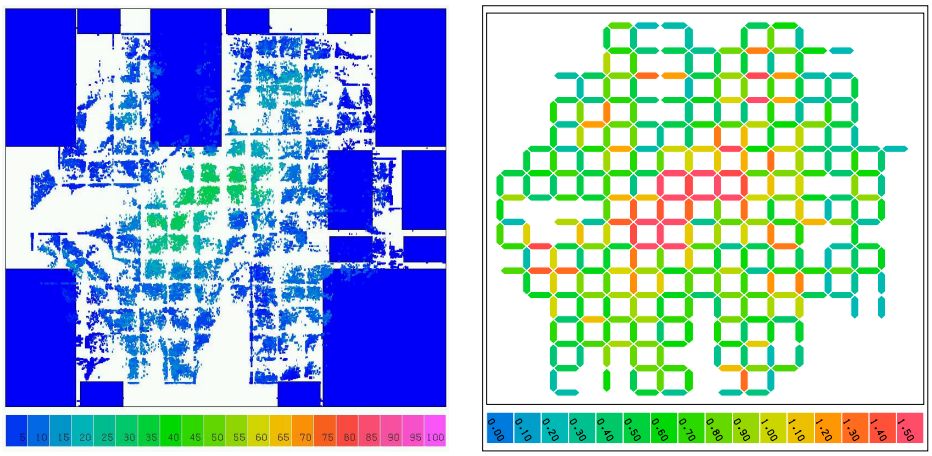


(b) Level 2

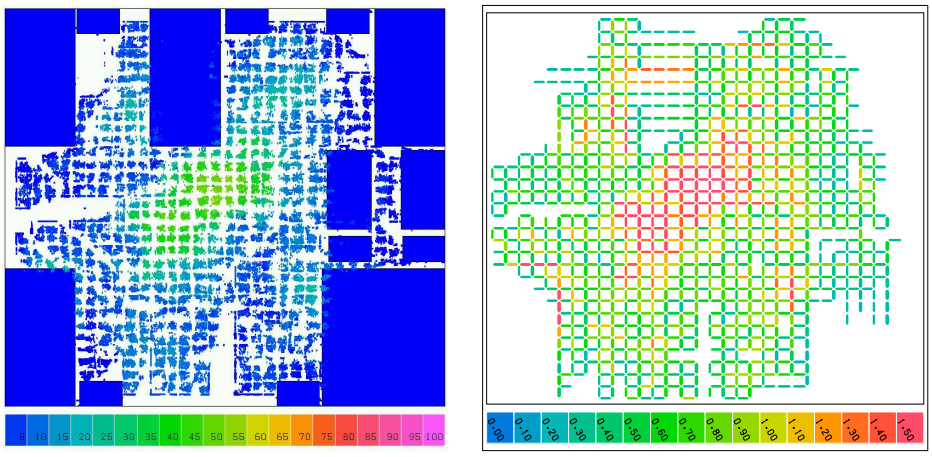


(c) Level 3

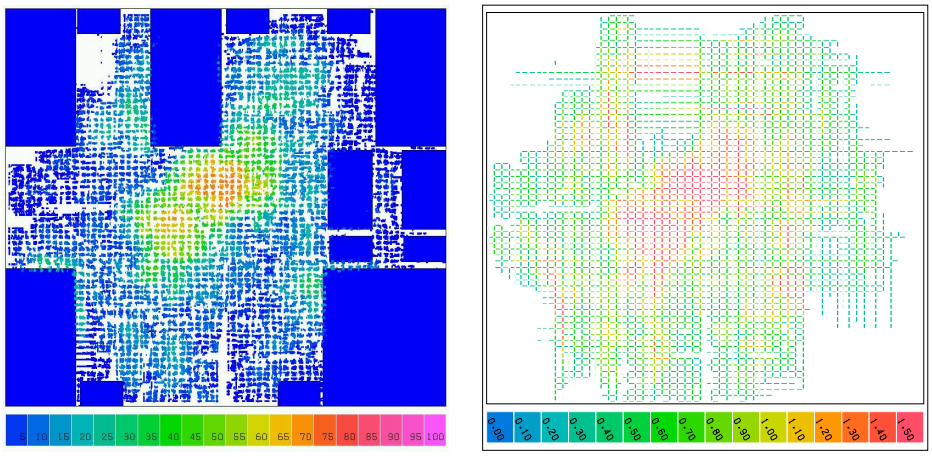
Figure 8.2: Levels 1 to 3 of a congestion-driven BONNPLACE run.



(a) Level 4

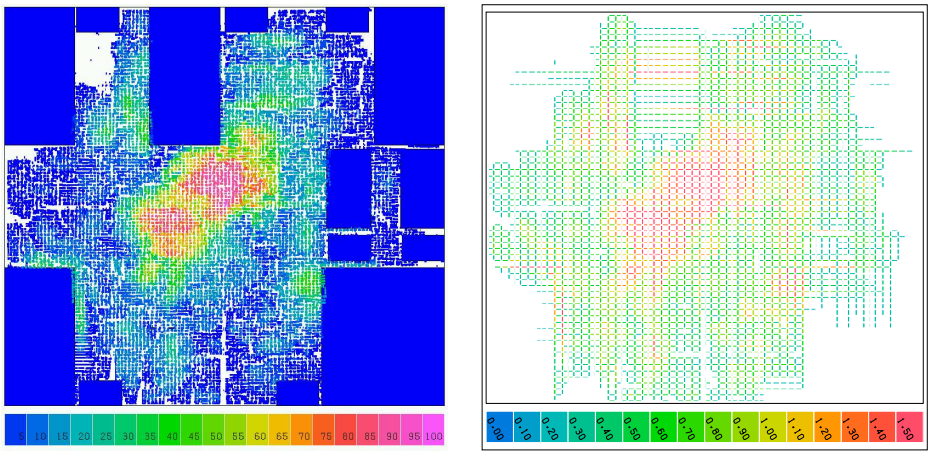


(b) Level 5

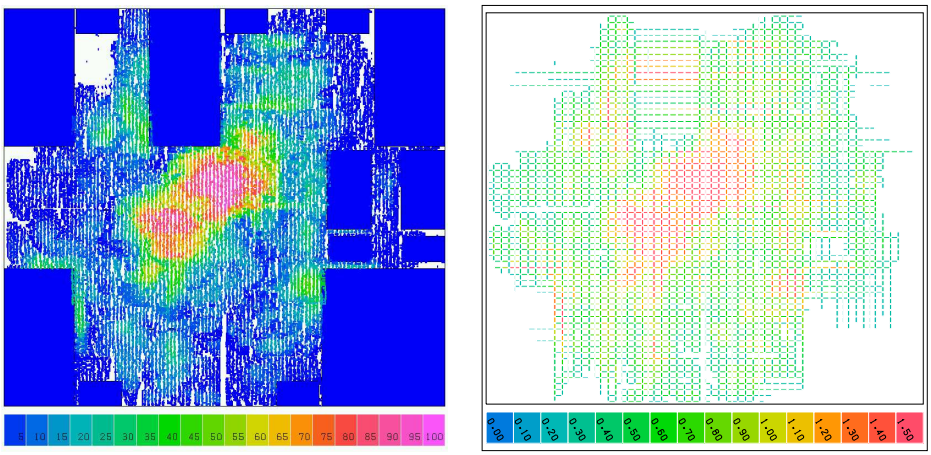


(c) Level 6

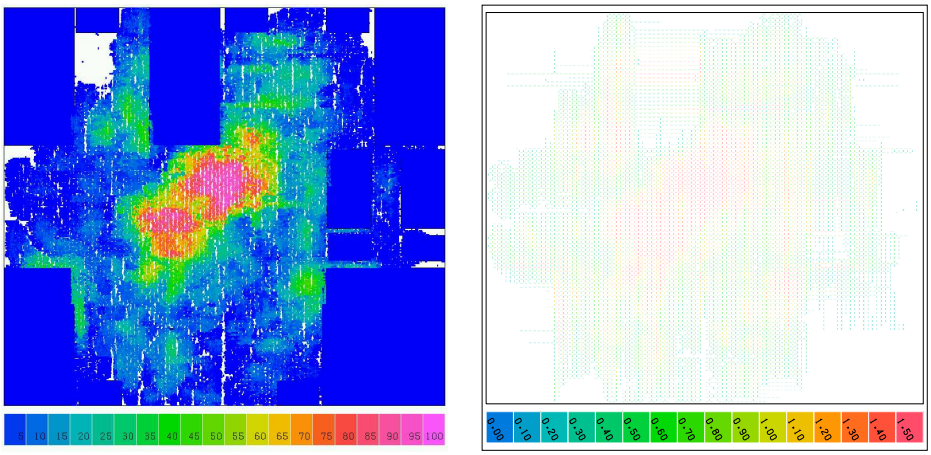
Figure 8.3: Levels 4 to 6 of a congestion-driven BONNPLACE run.



(a) Level 7

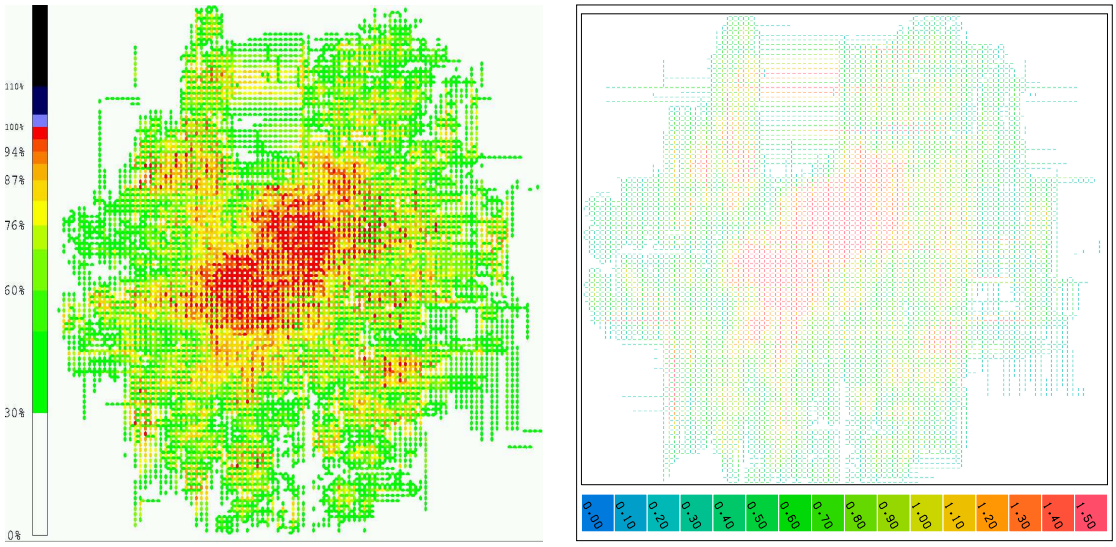


(b) Level 8



(c) Legalized Placement

Figure 8.4: Levels 7 and 8 of a congestion-driven BONNPLACE run and the result of the legalization.



(a) (b)

Figure 8.5: Comparison of congestion maps produced by the global router BonnRouteGlobal and by our congestion estimator. Both pictures clearly show the same big congested area in the center of the chip. Also on the other parts of the chip, both tools see almost the same routing criticalities.

means $b(c) = 0$ while hot pink means $b(c) \geq 1$. In the congestion pictures, the colors of the routing edges reflect their criticality. White means that the edge is not used at all, blue means a usage of about 10 %, while yellow indicate that $cong(e)$ is about 1. Edges with $cong(e) \geq 1.5$ are red, they causes the highest possible inflating values. Figure 8.4 (c) shows the placement after legalization and the corresponding congestion map. Note that the congestion estimation in level 1, 2, and 8 and after the legalization are computed only for this illustration as they are not used to change the inflating values $b(c)$.

8.4 Computational results

In this section, we will evaluate the quality of the congestion estimation and the performance of the congestion-driven BonnPlace. In order to test the accuracy of our congestion estimation and to analyze the routability of the placements, we used BonnRouteGlobal, a global router based on a multicommodity flow algorithm (see Albrecht [2001] and Müller [2002]). All runs were performed on an IBM 650 with 8 processors of 1.45 GHz.

8.4.1 Congestion Analysis vs. Global Routing

As a first step, we will examine the accuracy of our congestion estimation routine by comparing the output of BonnRouteGlobal with our congestion estimator. In Figure 8.5, we show the two congestion estimations for a placement of Jens that was produced by a

congestion-drivem BonnPlace run (with manually placed macros). It can be seen nicely that the very fast congestion estimation used by BonnPlace matches the output of BonnRouteGlobal quite well: The main congestion problem is in the middle of the chip. All other chips of the test suite show a similar behavior: The important congested spots are identified by our congestion estimator, sometimes the tool is a little bit too pessimistic.

To check if the algorithm identifies the routing hot-spots correctly we ran our estimator and BonnRouteGlobal (with the same routing grid) on legalized placements and compared the sets of the most critical edges. To avoid wrong estimations on edges above large macros, we reduced the routing capacities on blockages by a factor of 0.8. As described above, such edges on macros are not very important for our algorithm but for such tests macros cannot be ignored. The Tables 8.1 and 8.2 show the results of the tests. In Table 8.1, we consider the 20% most critical routing edges calculated by BonnRouteGlobal. We examine how critical they are in the estimation we use during placement. For different values of α, we count how many of these critical edges belong to the α% most critical edges in the BonnPlace estimator. We see that 75% up to 88% of them belong to the 20% most critical edges of the BonnPlace estimator. Some of them are not assumed to be that critical by our estimator, but only a small fraction of them does not belong to at least the 50% most critical edges of the estimator. So Table 8.1 shows that our estimation is not too bad in critical areas.

α	Jens	Heidrun	James	Sandra	Ulrich
20	77.0 %	75.6 %	88.2 %	80.2 %	76.3 %
30	91.9 %	86.0 %	94.3 %	92.1 %	92.2 %
40	96.7 %	93.6 %	96.0 %	95.8 %	98.0 %
50	99.0 %	94.8 %	97.4 %	96.8 %	99.9 %

Table 8.1: Percentage of the 20% most critical edges of BonnRouteGlobal that belong to the α% most critical edges of the BonnPlace congestion estimation.

For Table 8.2, we made the test the other way round. We considered the 20% most critical edges according to our estimator and checked if they are really critical (in the estimation of BonnRouteGlobal). The table shows how many of the edges belong to the α% most critical edges of BonnRouteGlobal. We see that a big part of the edges that are assumed to be critical by our estimator are really critical.

α	Jens	Heidrun	James	Sandra	Ulrich
20	77.0 %	75.6 %	88.2 %	80.2 %	76.3 %
30	93.8 %	93.2 %	98.2 %	96.9 %	96.3 %
40	98.6 %	98.5 %	99.5 %	99.9 %	98.7 %
50	99.9 %	99.1 %	100.0 %	100.0 %	98.9 %

Table 8.2: Percentage of the 20% most critical edges of the BonnPlace congestion estimation that belong to the α% most critical edges of BonnRouteGlobal.

The Tables 8.1 and 8.2 together show that our estimation of the most critical edges is quite accurate. Only for a very small percentage of the edges the estimation is completely wrong.

Most of the critical edges are detected correctly. Note that we compared our estimation to the final congestion computation of BonnRouteGlobal after all optimization steps of the router. This means that some routing criticalities are solved at that time by making detours while other routing edges get more critical as they are used for detours. Compared to the initial routing estimation of BonnRouteGlobal, our congestion estimator is even more accurate.

While the results are quite similar, the running times of the tools differ a lot: For our largest test-case, Ulrich, BonnRouteGlobal needs between three and four hours, while our internal congestion estimator runs less than a minute.

8.4.2 Congestion-Driven Placement

Finally, we will analyze the behavior of congestion-driven BonnPlace compared to standard BonnPlace. On many chips, even the standard version of BonnPlace produces a routable placement if the allowed density is small enough. Therefore, we will run BonnPlace with different density parameters and compare routability and (routing) netlength on these placements.

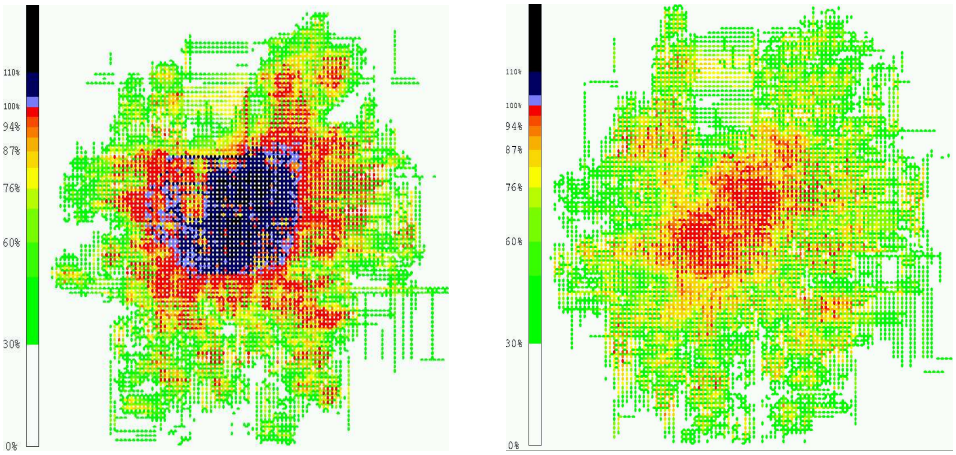
Table 8.3 and Table 8.4 give an overview of our runs with standard and congestion-driven BonnPlace. We used five different routing-critical ASICs for the experiments. Each row in the table corresponds to a run of BonnPlace followed by a run of BonnRouteGlobal. For each run, the second columns of the tables report the maximum allowed density at the beginning of the placement run. In each level, this density is increased by 1 %. For the congestion-driven runs, we have set $\tau := 0.2$. Columns four and five contain the bounding-box netlength after placement and the wallclock running time for the BonnPlace run, performed in parallel on four processors. The average inflation value at the end of the placement run is shown in column six, so the number in this column is the percentage by which $\sum_{c \in C}(1 + b(c))s(c)$ is bigger than $\sum_{c \in C} s(c)$. Columns seven to ten summarize the results of the BonnRouteGlobal runs on the different placements. Column seven reports the total length of all Steiner trees computed by the router, and columns eight the wallclock running time for the global routing, also run in parallel on four processors of the same machine as the placement runs. Columns nine indicates if the router succeded on these instances: the numbers are the sums of the overloads on all edges of the global routing graph. Here, the load of an edge is the number of Steiner trees using this edge, and if its load is bigger than its capacity, the overload of an edge is the difference between the load and the capacity (it is zero otherwise). Hence, a non-zero number in this column means that the global router failed on the instance. The last columns gives an impression of the distribution of the routing criticalities. To produce this "magic number", BonnRouteGlobal computes a 20 % median on the edges of the global routing graph weighted by their load, so the program considers a set of edges whose load is not smaller than the load on the edges outside the set such that the total load on the edges in this set is (approximately) 20 % of the total load on all edges. The number in the column is the average relative load on these edges, so it is the average of the percentages of the edge loads compared to the edge capacities. A higher "magic number" means generally a more difficult routing instance.

Chip	Placement					Routing			
	Initial density	Congestion driven	BB netlength	Running time	Inflation	Netlength	Running time	Overflows	20 % median
Jens	77.0 %	no	6.76 m	3:54	-	8.80 m	2:10	1 725	105.5 %
	82.0 %	no	6.72 m	3:21	-	8.73 m	2:20	4 543	115.8 %
	82.0 %	yes	7.10 m	6:52	16.3 %	8.28 m	1:49	0	96.0 %
Heidrun	58.0 %	no	37.51 m	24:26	-	39.80 m	16:41	0	90.0 %
	60.0 %	no	38.04 m	27:51	-	40.75 m	17:02	2	91.3 %
	65.0 %	no	35.32 m	25:35	-	38.10 m	17:51	1 542	96.4 %
	70.0 %	yes	36.87 m	37:47	8.8 %	38.48 m	15:18	0	89.8 %
	75.0 %	yes	35.74 m	37:00	9.2 %	37.85 m	16:49	0	90.4 %
	80.0 %	yes	34.50 m	44:31	9.9 %	37.94 m	19:34	3 035	98.6 %
James	50.0 %	no	116.93 m	28:17	-	128.30 m	47:02	0	86.7 %
	55.0 %	no	113.80 m	25:58	-	125.36 m	46:22	0	87.6 %
	60.0 %	no	114.81 m	27:40	-	129.76 m	49:27	0	93.6 %
	65.0 %	no	111.99 m	26:23	-	129.59 m	1:40:27	2 051	98.2 %
	70.0 %	no	109.74 m	26:03	-	128.21 m	2:03:44	19 396	101.9 %
	70.0 %	yes	114.89 m	48:49	22.0 %	124.04 m	45:32	0	86.5 %
	75.0 %	yes	113.08 m	48:57	24.2 %	122.89 m	45:17	0	87.1 %
	80.0 %	yes	114.68 m	51:12	24.6 %	127.44 m	1:11:11	0	89.2 %

Table 8.3: Results of placement and routing runs for the chips Jens, Heidrun, and James.

Chip	Placement					Routing			
	Initial density	Congestion driven	BB netlength	Running time	Inflation	Netlength	Running time	Overflows	20 % median
Sandra	36.0 %	no	423.79 m	1:59:14	-	493.40 m	2:12:11	0	85.1 %
	38.0 %	no	411.36 m	1:49:25	-	483.28 m	2:06:19	1	89.8 %
	40.0 %	no	414.51 m	1:49:46	-	485.67 m	2:27:57	6 841	92.5 %
	45.0 %	no	397.88 m	1:38:57	-	466.24 m	9:17:42	112 325	104.4 %
	60.0 %	yes	397.26 m	3:36:23	40.2 %	465.14 m	2:28:22	0	86.6 %
	65.0 %	yes	366.78 m	3:03:16	37.5 %	433.24 m	2:32:30	13	90.0 %
	70.0 %	yes	360.65 m	4:16:54	41.3 %	429.11 m	4:34:37	3 806	98.3 %
Ulrich	40.0 %	no	611.53 m	4:23:53	-	668.16 m	2:46:11	0	86.1 %
	50.0 %	no	563.52 m	3:54:11	-	642.07 m	3:25:04	0	89.2 %
	52.0 %	no	553.70 m	4:06:46	-	646.44 m	3:24:32	0	89.6 %
	53.0 %	no	560.32 m	4:12:53	-	647.27 m	3:05:25	0	90.3 %
	54.0 %	no	548.27 m	4:04:17	-	630.69 m	2:58:18	1 806	90.2 %
	55.0 %	no	546.98 m	4:05:07	-	637.05 m	3:21:10	530	91.0 %
	60.0 %	no	526.50 m	3:58:35	-	616.35 m	4:19:50	4 074	98.0 %
	53.0 %	yes	593.39 m	6:24:17	19.3 %	646.74 m	3:34:57	0	86.7 %
	60.0 %	yes	569.02 m	5:53:50	20.8 %	647.56 m	3:08:40	0	88.6 %
	65.0 %	yes	552.73 m	6:00:56	18.0 %	649.09 m	4:30:12	0	92.7 %
	67.0 %	yes	547.84 m	5:55:48	18.9 %	629.71 m	3:11:32	0	89.8 %
	70.0 %	yes	543.75 m	6:40:01	30.0 %	626.96 m	4:19:20	118	93.2 %

Table 8.4: Results of placement and routing runs for the chips Sandra and Ulrich.



(a) Standard BonnPlace (b) Congestion-driven BonnPlace

Figure 8.6: Comparison of BonnRouteGlobal congestion estimations on Jens. The normal BonnPlace placement (a) is not routable (the blue or black edges have an overload, and the red edges are critical). The congestion-driven BonnPlace (b) creates a routable placement.

The results of our experiments can be summarized as follows:

- The congestion-driven BonnPlace is much slower than the standard version. It may take 50 % up to 100 % more running time.
- The running time of BonnRouteGlobal may increase drastically when the placement gets routing-critical.
- In the congestion-driven mode, we can run BonnPlace with a significantly higher allowed density than in the standard mode and still get a routable result. This makes the usage of the program easier as the designer can always use quite a high density and does not have to search the right density in several runs.
- On the chip Jens, it was even impossible to get a routable placement without using the congestion-driven mode.
- As only the circuits in routing-critical areas are inflated, we can achieve better netlength with congestion-driven placements than with standard placement runs. Standard BonnPlace succeded to produce a routable placement on four chips. On these chips, the best routable placement computed by congestion-driven BonnPlace has, on average, a 3.2 % shorter bounding-box netlength and a 3.6 % shorter global routing wirelength than the best routable placement computed by standard BonnPlace.

To illustrate the effect of the congestion-driven placement mode, Figure 8.6 compares for the chip Jens BonnRouteGlobal congestion pictures for placements computed with standard and congestion-driven BonnPlace.

Chapter 9

Further Experiments

In the experiments presented in the previous chapters, we considered single aspects and features of our placement algorithm. In this chapter, we want to test our placement tool as a whole and compare it with previous appproaches. As testcases, we will use again some of the IBM ASICs presented in Chapter 2 and some publicly available benchmarks.

9.1 Experiments on Real-World Chips

In our experiments on the IBM ASICs we applied the iterative partitioning (up to level 7) and the 2×3-partitioning in the last levels to reduce the number of levels. In addition to the standard method, we made tests with 3×3-windows for repartitioning and the overload reduction. For a comparison, we used the old version of BonnPlace that combined the global placement algorithms presented by Vygen [1997] and the legalization approach described by Vygen [1998]. The tests were made on an IBM 650 with eight processors of 1.45 GHz. Table 9.1 contains the results of our experiments. For each run, the table shows the bounding-box netlength in meters ("BB"), the wallclock running time, i.e., the difference between the time when the program stopped and the time when it started in hours, minutes, and seconds ("Time"), and the maximal memory allocated during the run ("Memory") in megabytes. The results for the old BonnPlace are shown in the columns two to four. We ran the new BonnPlace with standard optimization parameters on a single processor (columns five to seven) and in parallel on four processors (columns eight to ten). The runs on four processors with 3×3-windows in repartitioning and in the reduction of overloads are shown in the last three columns. Note that all versions of BonnPlace had to meet the same constraints, especially the same density constraints. The experiments clearly demonstrate that the new BonnPlace is much faster than the old version (roughly by a factor of 2 for the sequential version and by more than a factor of 4 for the parallel version) and that it is able to produce placements with signficantly shorter netlength using the extended repartitioning method (without being slower than the old BonnPlace version).

Figure 9.1 visualizes the running times presented in Table 9.1. The diagram shows the running times of the different versions of BonnPlace (y-coordinate) for different instance

Chip	Old BonnPlace			New BonnPlace (standard)						New BonnPlace		
				1 Processor			4 Processors			3×3 repart, 4 Processors		
	BB	Time	Memory	BB	Time	Memory	BB	Time	Memory	BB	Time	Memory
Jens	6.92	0:10:05	187	6.77	0:07:15	118	6.76	0:03:10	123	6.53	0:09:18	193
Hans	7.43	0:14:12	319	7.53	0:08:18	170	7.56	0:03:53	167	7.18	0:10:40	189
Christian	166.44	1:04:02	1 749	166.14	0:32:01	858	166.04	0:17:48	848	156.45	0:34:40	884
James	108.34	1:38:21	4 023	109.57	0:53:48	1 873	109.80	0:26:55	1 849	100.88	0:56:27	1 929
Paul	28.81	1:27:47	1 678	27.89	0:53:55	921	27.88	0:27:43	935	26.83	0:54:51	1 040
Sven	253.07	3:28:56	5 469	254.23	2:09:21	2 653	252.85	1:02:28	2 662	246.16	1:45:16	2 736
Alex	207.98	4:43:44	5 895	201.53	2:57:09	2 900	200.99	1:19:15	2 950	197.59	2:19:14	3 061
Sandra	340.55	7:19:11	4 856	328.65	3:28:46	2 636	328.42	1:32:21	2 689	318.36	3:29:17	2 984
Reinhardt	366.59	6:29:57	5 042	360.65	3:20:25	2 752	360.96	1:36:39	2 801	355.23	3:04:54	3 198
Nadine	375.73	9:38:13	7 690	379.40	4:34:48	4 009	382.42	2:11:11	4 036	364.05	4:11:10	4 380
Hardy	353.50	9:45:39	6 131	365.71	4:43:04	3 413	363.24	2:15:40	3 542	341.05	4:32:14	4 277
Wolf	467.09	13:34:36	16 647	505.13	6:10:37	7 946	501.81	3:16:23	7 947	444.97	6:18:45	8 296
Ulrich	505.06	15:35:06	17 368	504.38	7:57:10	8 223	506.77	3:50:31	8 268	490.20	6:52:41	8 738
Fermi	378.52	23:21:41	19 167	368.98	12:17:50	10 151	368.00	6:55:38	10 182	355.51	9:08:56	10 563

Table 9.1: Experimental results on the IBM ASICs.

sizes (x-coordinate). As one can easily see, the running time of the new BonnPlace grows only slightly faster than linearly in the number of circuits. It should be noted that on our fastest machines (with 2.6 GHz Opteron processors) BonnPlace is faster by a factor of 2 compared to the results presented here. We used an older and slower machine in the experiments as we wanted to compare to the old BonnPlace which does not run on the Opteron machines. On the Opteron machines, we can place Fermi, the largest chip in our testsuite, in three hours and a half (on four processors and with standard repartitioning), so BonnPlace can place more than 1 000 000 circuits in one hour. This is fast enough to handle even the largest industrial instances.

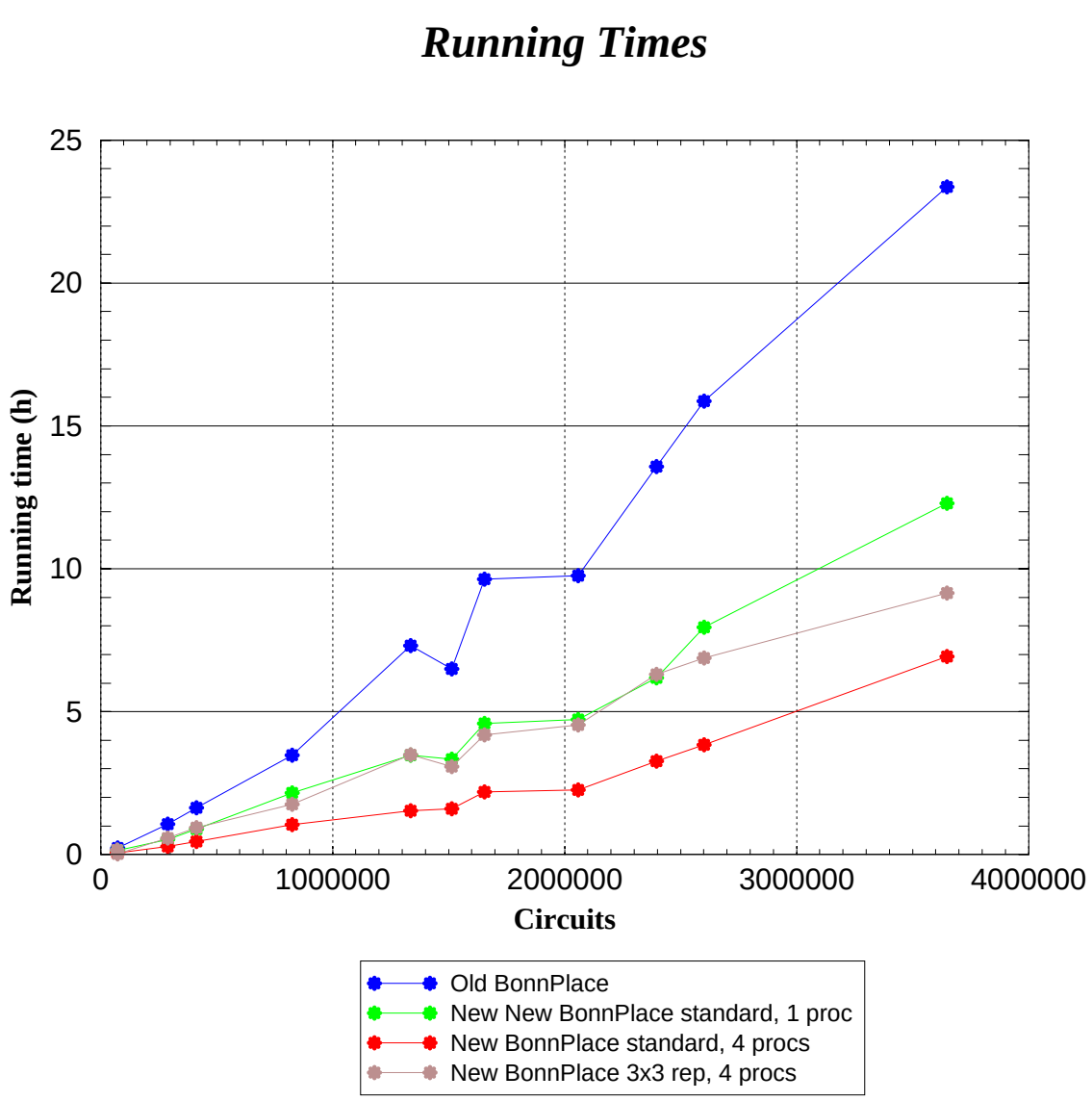


Figure 9.1: Running times of the different version of BonnPlace.

9.2 Experiments on Benchmarks

9.2.1 ISPD 2002 Benchmarks

The first set of publicly available testcases we used for our experiments with the new BonnPlace are the ISPD '02 benchmarks (see Adya and Markov [2002, 2005], and the website http://vlsicad.eecs.umich.edu/BK/ISPD02bench). These instances were generated from real-world chips (see Alpert [1998]) and hence are quite realistic, although they are relatively small and, by now, outdated. Table 9.2 summarizes our experiments on these instances. Column two contains the total number of circuits of the chips. For a comparison, we show in columns three and four the results of the placer Feng Shui 2.4 as reported by Khatkate et al. [2004]. So far, the Feng Shui-results were the best published placements on the ISPD '02 benchmarks. We cite the bounding-box netlength ("BB") and the running time on an 2.5 GHz Pentium 4 workstation ("Time"). Columns five to seven contain the results for a four-processor run of BonnPlace with 3×3-repartitioning. The bounding-box netlength ("BB"), the difference to the Feng Shui-results ("Gap") and the wallclock running time ("Time") are shown. Again, we ran BonnPlace on up to four 1.45 GHz processors of an IBM 650 machine. The numbers demonstrate that we can improve the Feng Shui-results on 16 of the 18 benchmarks. The average improvement (computed via the geometric mean of the ratios) is 5.1 %.

Chip	# Circuits	Feng Shui 2.4		BonnPlace		
		BB	Time	BB	Gap	Time
IBM01	12 506	2.41	0:03:00	2.26	- 6.2 %	0:06:02
IBM02	19 342	5.34	0:05:00	4.93	- 7.7 %	0:09:39
IBM03	22 853	7.51	0:06:00	7.01	- 6.7 %	0:11:27
IBM04	27 220	7.96	0:07:00	8.23	3.4 %	0:13:14
IBM05	28 146	10.10	0:08:00	10.02	- 0.8 %	0:12:38
IBM06	32 332	6.82	0:10:00	6.55	- 4.0 %	0:12:39
IBM07	45 639	11.71	0:13:00	10.41	-11.1 %	0:19:30
IBM08	51 023	13.60	0:16:00	12.68	- 6.8 %	0:30:27
IBM09	53 110	13.83	0:15:00	13.27	- 4.0 %	0:34:51
IBM10	68 685	37.48	0:22:00	32.92	-12.2 %	0:32:17
IBM11	70 152	19.96	0:21:00	19.15	- 4.1 %	0:37:02
IBM12	70 439	35.57	0:23:00	31.90	-10.3 %	0:48:18
IBM13	83 709	24.95	0:16:00	24.31	- 2.6 %	0:48:26
IBM14	147 088	38.48	0:52:00	37.82	- 1.7 %	1:00:05
IBM15	161 187	52.14	1:27:00	49.31	- 5.4 %	1:25:08
IBM16	182 980	61.33	1:16:00	57.88	- 5.6 %	1:49:33
IBM17	184 752	70.60	1:44:00	66.65	- 5.6 %	3:19:05
IBM18	210 341	45.05	1:54:00	45.74	1.5 %	1:29:05
Average					- 5.1 %	

Table 9.2: The results for ISPD '02 testsuite.

9.2.2 PEKO Benchmarks

The second benchmark testsuite we used are the PEKO ("**P**lacement **E**xample with **K**nown **O**ptimal wirelength") chips, a set of benchmarks that are extracted from the same instances as the ISPD '02 benchmarks. However, they were modified in such a way that an optimum placement is known (see Chang et al. [2003]). Especially, all macros and non-local nets have disappeared in that modification. There are four sets of PEKO benchmarks. The first two sets PEKO1 and PEKO2 have no IO-connections. Since BonnPlace needs at least one preplaced circuit or IO-pin, we used the testsuites PEKO3 and PEKO4 with boundary IO pins for our experiments. PEKO3 and PEKO4 are generated by the same algorithm but each chip in PEKO4 contains ten times the number of circuits of the corresponding chip in PEKO3. Very recently, Chan, Cong, and Sze [2005] published results on the testsuites PEKO3 and PEKO4 for a placer that combines force-directed placement and multilevel partitioning. They obtained netlengths that were approximately 20 % away from the optimum. All previous approaches produced netlength that were at least 40 % away from the optimum.

Tables 9.3 and 9.4 give an overview of our experiments with the PEKO benchmarks. In both tables, the first column contains the name of the chip, the second column the total number of circuits, and the third number is the BB netlength of an optimal placement (divided by 1 000 000, the same scaling is used for the results of our placer). For all our experiments, we allowed the program to use up to four processors. We ran BonnPlace with three different parameter settings. For a very fast placement, we ran BonnPlace with a reduced accuracy in the QP solution, a reduced number of constrained QPs before partitioning, and the restriction to one repartitioning loop in each level (columns "BonnPlace FAST"). The runs with the standard parameters are shown in the columns labelled with "BonnPlace NORMAL". We also ran BonnPlace allowing up to five repartitioning loops and computing the repartitiong on 3×3-windows ("BonnPlace BEST"). For each version, we report the bounding-box netlength ("BB"), the gap between our result and the bounding-box netlength of an optimal solution ("Gap"), and the wallclock running time of the whole placement process ("Time").

The tables show that the fast version of BonnPlace is able to produce placements that are on average about 30 % away from the optimum. Even on the largest instances this version does not need more than two and a half hour for the complete placement. The standard version is slightly slower but yields placement that differ only by about 25 % from the optimum. If we run BonnPlace with an exhaustive use of the repartitioning method, we can even produce placements whoose netlength is within 17 % of the optimum, so these are the best known results of a placer on these instances. However, these superior results are paid by significantly bigger running times. It depends on the situation which parameter setting for BonnPlace is best. For an early attempt, the fast mode may be acceptable, while for critical chips (especially if they are not too big), the increased repartitioning seems to be recommendable.

It should be noted that we cannot assume that our placements for real-world chips are as close to the optimum as the placements on the PEKO instances. All circuits in the PEKO benchmarks are squares of the same size, so, there are no macros or blockages to handle. Moreover, the boundary IO-pins on three borders of the chip (the right-hand-side does not

Chip	Circuits	Opt	BonnPlace FAST			BonnPlace NORMAL			BonnPlace BEST		
		BB	BB	Gap	Time	BB	Gap	Time	BB	Gap	Time
Peko01	12 506	0.82	1.07	30.4 %	0:00:22	1.04	26.8 %	0:00:24	0.97	18.3 %	0:00:53
Peko02	19 342	1.27	1.63	28.1 %	0:00:39	1.59	25.0 %	0:00:48	1.48	16.7 %	0:01:47
Peko03	22 853	1.51	1.95	28.9 %	0:00:38	1.90	25.6 %	0:00:52	1.78	17.8 %	0:01:34
Peko04	27 220	1.76	2.30	30.5 %	0:00:52	2.21	25.7 %	0:00:57	2.05	16.5 %	0:02:23
Peko05	28 146	1.95	2.50	28.3 %	0:00:49	2.44	25.0 %	0:00:55	2.27	16.2 %	0:02:07
Peko06	32 332	2.07	2.67	29.2 %	0:00:53	2.61	26.2 %	0:00:57	2.43	17.5 %	0:02:14
Peko07	45 639	2.89	3.67	26.8 %	0:01:07	3.61	24.8 %	0:01:24	3.37	16.7 %	0:02:54
Peko08	51 023	3.15	4.13	31.1 %	0:01:30	3.94	25.0 %	0:01:57	3.71	17.5 %	0:04:21
Peko09	53 110	3.65	4.69	28.4 %	0:01:35	4.55	25.6 %	0:02:01	4.24	16.2 %	0:04:00
Peko10	68 685	4.75	6.05	27.4 %	0:02:09	5.93	24.7 %	0:02:36	5.50	15.7 %	0:07:09
Peko11	70 152	4.72	6.04	27.9 %	0:02:06	5.88	24.5 %	0:02:19	5.49	16.2 %	0:05:42
Peko12	70 439	5.02	6.51	29.6 %	0:02:25	6.31	25.7 %	0:02:48	5.84	16.3 %	0:06:05
Peko13	83 709	5.89	7.56	28.3 %	0:02:33	7.36	24.9 %	0:03:09	6.85	16.3 %	0:07:01
Peko14	147 088	9.03	11.54	27.8 %	0:04:15	11.25	24.6 %	0:05:23	10.56	16.9 %	0:10:35
Peko15	161 187	11.60	15.12	30.3 %	0:04:43	14.80	27.6 %	0:06:19	13.46	16.0 %	0:13:15
Peko16	182 980	12.50	16.08	28.7 %	0:07:07	15.66	25.3 %	0:08:37	14.63	17.0 %	0:17:29
Peko17	184 752	13.50	17.46	29.4 %	0:06:31	16.93	25.4 %	0:09:39	15.72	16.4 %	0:17:04
Peko18	210 341	13.20	16.76	27.0 %	0:08:20	16.31	23.6 %	0:10:19	15.39	16.6 %	0:17:45
Average				28.8 %			25.3 %			16.7 %	

Table 9.3: The results for the PEKO3 testsuite.

Chip	Circuits	Opt BB	BonnPlace FAST BB	BonnPlace FAST Gap	BonnPlace FAST Time	BonnPlace NORMAL BB	BonnPlace NORMAL Gap	BonnPlace NORMAL Time	BonnPlace BEST BB	BonnPlace BEST Gap	BonnPlace BEST Time
Peko01	125 060	8.2	10.5	27.9 %	0:03:58	10.3	25.4 %	0:04:40	9.6	17.2 %	0:09:20
Peko02	193 420	12.7	16.3	28.7 %	0:08:05	15.9	25.2 %	0:10:24	14.9	17.0 %	0:18:49
Peko03	228 530	15.1	19.3	27.9 %	0:08:52	18.8	24.8 %	0:10:41	17.7	17.2 %	0:19:29
Peko04	272 200	17.6	22.7	29.0 %	0:11:23	21.9	24.3 %	0:12:10	20.6	17.1 %	0:32:05
Peko05	281 460	19.5	24.8	27.0 %	0:12:53	24.2	23.9 %	0:15:04	22.6	15.9 %	0:30:41
Peko06	323 320	20.7	26.8	29.6 %	0:13:04	25.9	25.0 %	0:15:32	24.1	16.6 %	0:33:45
Peko07	456 390	28.9	36.9	27.6 %	0:20:52	36.0	24.5 %	0:23:49	33.6	16.1 %	0:44:47
Peko08	510 230	31.5	43.1	37.0 %	0:23:52	39.6	25.7 %	0:27:50	37.0	17.4 %	0:56:15
Peko09	531 100	36.5	47.6	30.5 %	0:23:39	45.4	24.4 %	0:29:14	42.5	16.5 %	0:58:28
Peko10	686 850	47.5	64.4	35.5 %	0:34:12	59.9	26.1 %	0:40:50	55.6	17.1 %	1:21:22
Peko11	701 520	47.2	60.7	28.6 %	0:32:33	59.4	23.7 %	0:40:55	54.8	16.2 %	1:10:32
Peko12	704 390	50.2	65.4	30.3 %	0:35:51	62.9	25.3 %	0:42:44	58.8	17.1 %	1:23:34
Peko13	837 090	58.9	75.8	28.6 %	0:41:18	74.0	25.6 %	0:52:45	68.4	16.1 %	1:43:53
Peko14	1 470 880	90.3	123.8	37.0 %	1:20:23	113.5	25.7 %	1:39:59	105.4	16.7 %	3:13:34
Peko15	1 611 870	116.0	150.0	29.3 %	1:41:59	143.8	24.0 %	2:04:53	133.6	15.1 %	3:46:17
Peko16	1 829 800	125.0	170.8	36.6 %	2:04:44	155.5	24.4 %	2:28:35	146.6	17.3 %	4:49:48
Peko17	1 847 520	135.0	183.5	36.0 %	2:10:12	171.3	26.9 %	2:41:54	156.9	16.2 %	5:05:40
Peko18	2 103 410	132.0	176.7	33.8 %	2:24:38	165.7	25.5 %	2:37:15	153.5	16.3 %	5:02:36
Average				31.1 %			25.0 %			16.6 %	

Table 9.4: The results for PEKO4 testsuite.

contain any pins) lead, together with the fact that there are only local nets in the optimum solution, to a good circuit spreading even in the first QP solution. This makes the life quite easy for any QP-based placer. In fact, Liu and Marek-Sadowska [2004] constructed benchmarks similar to the PEKO benchmarks but with IO-pins on all four borders of the chip area (so they had a even better spreading in the QP solution) and demonstrated that an algorithm that just considers a QP solution and then legalizes the placement by simulated annealing produces results that are only 12 % away from the optimum.

9.2.3 ISPD 2005 Placement Contest Benchmarks

The most realistic public benchmarks that are available are the chips from the ISPD 2005 testsuite. These instances contain the placement data of recent IBM ASICs. They were published for a contest of (academic) placement tools at the International Symposium on Physical Design 2005 (see Nam et al. [2005]). The goal in this contest was only the minimization of bounding-box netlength (without regaring any density constraints), hence the benchmarks do not contain any information concerning routability or timing. Table 9.5 gives an overview of the chips used for the contest. The numbers of circuits, nets, and pins include both preplaced and movable objects. The density describes the size of all circuits (preplaced or movable) divided by the total size of the placement area.

Chip	# Circuits	# Nets	# Pins	Density
adaptec2	211 447	221 142	944 053	75.7 %
adaptec4	255 023	266 009	1 069 482	78.6 %
bigblue1	278 164	284 479	1 144 691	54.2 %
bigblue2	557 866	577 235	2 122 282	61.8 %
bigblue3	1 096 812	1 123 170	3 833 218	85.7 %
bigblue4	2 177 353	2 229 886	8 900 078	65.3 %

Table 9.5: The instances of the ISPD '05 placement contest.

In the contest, every participant had five days to compute the placement with the best possible netlength. Except for that upper bound of five days, there was no restriction on the running time, on the number of attempts or on the amount of postoptimization that was applied to the placements. Table 9.6 sums up the results of the contest as they were presented during the ISPD. The results are not published in the conference proceedings, so we cite them according to the slides on the conference web page www.ispd.cc/ispd/main.htm. For each participating placer, we cite a short paper summarizing its main features by a reference number and show the netlengths on the different instances in a column. In the last row, we present the average gap between the results of the specific placer and APlace, the winner of the contest (computed by the geometric mean of the ratios).

In Table 9.7, we demonstrate how BonnPlace performed on the placement contest instances. We chose the user parameters as if running time was not an issue and focused on netlength only. Hence, we applied several 3×3-repartitioning iterations in every level and

Chip	APlace [61]	mFar [55]	Dragon2005 [100]	mPL6 [25]	FastPlace [106]	Capo 9.1 [89]	NTUplace [30]	Feng Shui 5.0 [6]	Kraftwerk [81]
adaptec2	87.31	91.53	94.72	97.11	107.86	99.71	100.31	122.99	157.65
adaptec4	187.65	190.84	200.88	200.94	204.48	211.25	206.45	337.22	352.01
bigblue1	94.64	97.70	102.39	98.31	101.56	108.21	106.54	114.57	149.44
bigblue2	143.82	168.70	159.71	173.22	169.89	172.30	190.66	285.43	322.22
bigblue3	357.89	379.95	380.45	369.66	458.49	382.63	411.81	471.15	656.19
bigblue4	833.21	876.28	903.96	904.19	889.87	1 098.76	1 154.15	1 040.05	1 403.79
Average Gap	0.0 %	6.3 %	8.3 %	8.9 %	15.2 %	16.4 %	20.1 %	46.8 %	82.6 %

Table 9.6: The results of the ISPD '05 placement contest.

we allowed a very high density (more than 93 % at the end of global placement). Note that we cannot use 100 % density in global placmenent because on such instances larger movements in legalization are necessary that would deteriorate the netlength. To compare our results, we present the APlace netlengths and running times in columns two to four of the table. The numbers are taken from the slides of the APlace presentation at the ISPD 2005 as published on the conference web page. The numbers in column two are the bounding-box netlengths of APlace before postoptimization, while column three shows the lengths after postoptimization. We cite the running times (in hours) in the next column. We do not know exactly which platform was used for the single runs, but according to the slides the authors had access to a computer pool that consists mainly of 2.4 GHz and 2.8 GHz Xeon and 1.6 GHz Opteron processors. Columns five and six of the table contain the netlengths and the wallclock running times (h:mm:ss) of the BonnPlace runs (on up to four 2.6 GHz Opteron processors). As most of the netlengths shown in Table 9.6 are the result of a postoptimization, we applied a primitive postoptimization to our placements, too: We first ran the Clumping Algorithm (see Chapter 6) with netlength minimization as optimization goal (which is much more time-consuming than the standard version that minimizes quadratic movement). Then, we applied a greedy heuristic that swaps neighbouring circuits in the same zone and accepts the new arrangement of the circuits if netlength could be improved. We ran this swapping heuristic three times on the whole chip. The numbers in columns seven and eight are the final netlength and the running time (h:mm:ss) of the postoptimization. The last column contains the difference between our netlengths and APlace (both including postoptimization).

Chip	APlace			BonnPlace				
	Placement	Postopt		Placement		Postopt		Gap
	BB	BB	Time (h)	BB	Time	BB	Time	
adaptec2	92.18	87.31	3	95.53	0:48:04	94.16	0:01:53	7.8 %
adaptec4	194.75	187.65	13	199.54	1:32:17	197.06	0:08:06	5.0 %
bigblue1	97.85	94.64	5	103.31	0:48:21	101.64	0:00:45	7.4 %
bigblue2	147.85	143.82	12	158.77	1:20:13	156.02	0:08:47	8.5 %
bigblue3	407.09	357.89	22	357.86	3:29:09	352.19	0:20:17	- 1.6 %
bigblue4	868.07	833.21	50	852.11	8:23:24	839.12	1:02:48	0.7 %
Average								4.6 %

Table 9.7: The BonnPlace results on the ISPD '05 placement contest benchmarks compared to the APlace results.

We can summarize the results on the contest benchmarks as follows:

- On average, our netlengths were 4.6 % bigger than the netlength of the contest's winner, APlace.
- On the two largest instances, we produced placements with netlengths that are very close to the APlace netlengths. For bigblue3, we can even improve the best known placement.

- BonnPlace is significantly faster than APlace even if we assume that the machines used in the APlace runs were somewhat slower than our computers.
- Compared to all other participating placers, except Aplace, our results are better on average and, especially, on all instances with more than half a million of circuits.

It should be noted that for us, minimization of bounding-box netlength is not the only optimization goal (in fact it is just used to model other goals), and we normally want to avoid very high placement densities (close to 100 %) as these placements would not be routable. In contrast, most of the academic placers that took part in the contest are tuned for the minimization of netlengths. They apply extensive postoptimization and pack circuits as dense as possible. Nevertheless, though BonnPlace is designed rather for the optimization of large industrial chips than for such netlength experiments, our results on the benchmarks are competetive even to the best academic placers.

Bibliography

[1] Adolphson, D.L., Thomas, G.N. [1977]: *A linear time algorithm for a* $2 \times n$ *transportation problem.* SIAM Journal on Computing 6, 1977, 481–486.

[2] Adya, S.N., Chaturvedi, S., Roy, J.A., Papa, D.A., and Markov, I.L. [2004]: *Unification of partitioning, floorplanning and placement* Proceedings of the International Conference on Computer–Aided Design, 2004, 550–557.

[3] Adya, S.N., Markov, I.L. [2001]: *Fixed-outline floorplanning through better local search.* Proceedings of the International Conference on Computer Design, 2001, 328–334.

[4] Adya, S.N., Markov, I.L. [2002]: *Consistant placement of macro-blocks using floorplanning and standard-cell placement.* Proceedings of the International Symposium on Physical Design, 2002, 12–17.

[5] Adya, S.N., Markov, I.L. [2005]: *Combinatorial techniques for mixed-size placement.* ACM Transactions on Design Automation of Electronic Systems, 10, 2005, 58–90.

[6] Agnihotri, A.R., Ono, S., Madden, P.H. [2005]: *Recursive bisection placement: Feng Shui 5.0 implementation details.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2005, 230–232.

[7] Ahuja, R.K., Orlin, J.B., Stein, C., and Trajan, R.E [1994]: *Improved algorithms for bipartite network flow.* SIAM Journal on Computing 23, 1994, 906–933.

[8] Albrecht, C. [2001]: *Global routing by new approximation algorithms for multicommodity flow.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20, 2001, 622–632.

[9] Alpert, C.J. [1998]: *The ISPD98 circuit benchmark suite.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 1998, 85–90.

[10] Alpert, C.J., Huang, D.J.-H., Kahng, A.B. [1997]: *Multilevel circuit partitioning.* Proceedings of the 34th ACM/IEEE Design Automation Conference, 1997, 530 - 533.

[11] Armstrong, R.D., Jin, Z. [1997]: *A new strongly polynomial dual network simplex algorithm.* Mathematical Programming 78, 1997, 131–148.

[12] Arora, S., Frieze, A., Kaplan, H. [1996]: *A new rounding preocedure for the assignment problem withh applications to dense graph arrangement problem.* Proceedings of the 37th Annual Symposium on Foundations of Computer Science, 1996, 21–30.

[13] Balas, E., Zemel, E. [1980]: *An algorithm for large zero-one knapsack problems.* Operations Research 28, 1980, 1130–1154.

[14] Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E. [1973]: *Time bounds for selection.* Journal of Compute and System Science 7, 1973, 448–461.

[15] Brenner, U. [2000]: *Plazierung im VLSI-Design.* Diploma thesis, University of Bonn, 2000.

[16] Brenner, U., Pauli, A., Vygen, J. [2004] *Almost optiumum placement legalization with minimum total movement.* Proceedings of the International Symposium on Physical Design, 2004, 2–9.

[17] Brenner, U., Rohe, A. [2002]: *An effective congestion-driven placement framework.* Proceedings of the International Symposium on Physical Design, 2002, 6–11.

[18] Brenner, U., Rohe, A. [2003]: *An effective congestion-driven placement framework.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 22, 2003, 387–394.

[19] Brenner, U., Struzyna, M. [2005]: *Faster and better global placement by a new transportation algorithm.* Proceedings of the 42nd ACM/IEEE Design Automation Conference, 2005, 591–596.

[20] Brenner, U., Vygen, J. [2000]: *Faster optimal single-row placement with fixed ordering.* Design, Automation and Test in Europe, Proceedings, IEEE, 2000, 177–121.

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

[22] Brenner, U., 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.

[23] Bui, T.N., Jones, C. [1992]: *Finding good approximate vertex and edge partitions is NP-hard.* Information Processing Letters, 42, 1992, 153–159.

[24] Caldwell, A.E., Kahng, A.B., Markov, I.L. [2000]: *Can recursive bisection alone produce routable placements?* Proceedings of the 37th ACM/IEEE Design Automation Conference, 2000, 477–482.

[25] Chan, T.F., Cong, J., Romesis, M., Shinnerl, J., Sze, K., Xie, M. [2005]: *mPL6: A robust multilevel mized-size placement engine.* ACM/IEEE Proceedings of the International Symposium in Physical Design, 2005, 227–229.

[26] Chan, T., Cong, J., Sze, K. [2005]: *Multilevel generalized force-directed method for circuit placement.* ACM/IEEE Proceedings of the International Symposium in Physical Design, 2005, 185–192.

[27] Chang, Y.-C, Chang, Y.-W., Wu, G.-M., Wu, S.-W. [2000]: *B*-Trees: A new representation for non-slicing floorplans.* Proceedings of the 37th ACM/IEEE Design Automation Conference, 2000, 458–463.

[28] Chang, C.-C., Cong, J., Pan, Z., Yuan, X. [2003]: *Multilevel global placement with congestion control.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 22, 2003, 295–409.

[29] Chang, C.-C., Cong, J., Xie, M. [2003]: *Optimality and scalability of existing placement algorithms.* Proceedings of the Asia South Pacific Design Automation Conference, 2003, 621–627.

[30] Chen, T.-C., Hsu, T.-C., Jiang, Z.-W., Chang, Y.-W. [2005]: *NTUplace: A ratio partitioning based placement algorithm for large-scale mized-size designs.* ACM/IEEE Proceedings of the International Symposium in Physical Design, 2005, 236–238.

[31] Cheng, C.-L.E. [1994]: *RISA: Accurate and efficient placement routability modeling.* Proceedings of the International Conference on Computer–Aided Design, 1994, 690–697.

[32] Chung, F.R.K., Hwang, F.K. [1979]: *The largest minimal rectilinear Steiner trees for a set of n points enclosed i na rectangle with given perimeter.* Networks 38, 1979, 19–36.

[33] Cormen, T.H., Leiserson, C.E., Rivest, R.L. [1990]: *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[34] Doll, K., Johannes. F.M., Antreich K.J. [1994]: *Iterative placement improvement by network flow methods.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13, 1994, 1189–1199.

[35] Doll, K., Johannes, F.M., Sigl, G. [1991]: *DOMINO: Deterministic placement improvement with hill-climbing capabilities.* Proceedings if the IFIP International Conference on VLSI, 1991, 91–100.

[36] Eisenmann, H. [1999]: *Ein universelles Plazierverfahren für intergrierte Schaltungen.* Hieronymus, München, 1999.

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

[38] Even, G. Guha, S., and Schieber, B. [2000]: *Improved approximations of crossings in graph drawings and VLSI layout areas.* Proceedings of the 32nd Annual Symposium on the Theory of Computing, 2000, 296–305

[39] Faroe, O., Pisinger, D., Zachariasen, M. [2001]: *Local search for final placement in VLSI design.* Proceedings of the International Conference on Computer-Aided Design, 2001, 565–572.

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

[41] Ford, L.R., Fulkerson, D.R. [1962]: *Flows in Networks.* Princeton University Press, Princeton, 1962.

[42] Fredman, M.L., Tarjan, R.E. [1984]: *Fibonacci heaps and their uses in improved network optimization algorithms.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science, 1984, 338–346.

[43] Garey, M.R., Johnson, D.S. [1975]: *Complexity results for multiprocessor scheduling under resource constraints.* SIAM Journal on Computing 4, 1975, 397–411.

[44] Garey, M.R., Johnson, D.S. [1977]: *The rectilinear Steiner tree problem is NP-complete.* SIAM Journal on Applied Mathematics 32, 1977, 825–834.

[45] Garey, M.R., Johnson, D.S. [1979]: *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.

[46] Garey, M.R., Johnson, D.S., and Stockmeyer, L. [1976]: *Some simplified NP-complete graph problems.* Theoretical Computer Science 1, 1976, 237–267.

[47] Garey, M.R., Tarjan, R.E., Wilfong. G.T. [1988]: *One-processor scheduling with symmetric earliness and tardiness penalties.* Mathematics of Operations Research, vol. 13, 1988, 330–348.

[48] Goldberg, A.V., Tarjan, R.E. [1990]: *Finding minimum-cost circulations by successive approximation.* Mathematics of Operations Research, vol. 15, 1990, 430–366.

[49] Guo, P.N., Cheng, C.-K., Yoshimura, T. [1999]: *An O-tree Representation of non-slicing floorplan and its applications.* Proceedings of the 36th ACM/IEEE Design Automation Conference, 1999, 268–273.

[50] Hansen, M.D. [1989]: *Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems.* Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, 604–609.

[51] Held, S., Korte, B., Maßberg, J., Ringe, J., Vygen, J. [2003]: *Clock scheduling and clocktree construction for high performance ASICs.* Proceedings of the International Conference on Computer-Aided Design, 2003, 232–239.

[52] Hochbaum, D.S., Woeginger, G.J. [1999]: *An optimal algorithm for the bottleneck tranportation problem with a fixed number of sources.* Operations Resear Letters 24, 1999, 25–28.

[53] Hou, W., Yu, H., Hong, X., Cai, Y., Wu, W., Gu, J., Kao, W.H. [2001]: *A new congestion-driven placement algorithm based on cell inflation.* Proceedings on the 2001 Conference on Asia and South Pacific Design Automation, ACMPress, 2001, 605–608.

[54] Hu, B., Marek-Sadowska, M. [2002]: *Congestion minimization during placement without estimation.* Proceedings of the International Conference on Computer-Aided Design, 2002, 739–745.

[55] Hu, B., Zeng, Y., Marek-Sadowska, M. [2005]: *mFAR: Fixed-points-addition-based VLSI placement algorithm.* ACM/IEEE Proceedings of the International Symposium in Physical Design, 2005, 239–241.

[56] Huang, D.J.-H., Kahng. A.B. [1997]: *Partitioning-based standard-cell global placement with an exact objective.* ACM/IEEE Proceedings of the International Symposium in Physical Design, 1997, 18–25.

[57] Hung, P., Flynn, M.J. [1997]: *Stochastic congestion model for VLSI systems.* Technical Report CSL-TR-97-737, Stanford University.

[58] Hur, S.-W., Lillis, J. [2000]: *Mongrel: hybrid techniques for standard cell placement.* Proceedings of the International Conference on Computer-Aided Design, 2000, 165–170.

[59] Hwang, F.K. [1976]: *On Steiner's problem with rectilinear distance.* SIAM Journal on Applied Mathematics 14, 1976, 104–114.

[60] Johnson, D.B., Mizoguchi, T. [1978]: *Selecting the Kth element in $X + Y$ and $X_1 + X_2 + \cdots + X_m$.* SIAM Journal on Computing 7, 1978, 147–153.

[61] Kahng, A.B., Reda, S., Wang, Q. [2005]: *APlace: A general analytic placement framework.* ACM/IEEE Proceedings of the International Symposium in Physical Design, 2005, 233–235.

[62] Kahng, A.B., Tucker, P., Zelikovsky, A. [1999]: *Optimization of linear placements for wirelength minimization with free sites.* Proceedings of the Asia and South Pacific Design Automation Conference, 1999, 241–244.

[63] Kahng, A.B., Xu, X. [2003]: *Accurate pseudo-constructive wirelength and congestion estimation.* Proceedings of the International Workshop on System-Level Interconnect Prediction, 2003, 61–68.

[64] Kahng, A.B., Wang, Q. [2004]: *Implementation and extensibility of an analytic placer.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2004, 18–25.

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

[66] Karypis, G. Agarwal, R., Kumar, V., Shekhar, S. [1997]: *Multilevel hypergraph partitioning: application in VLSI design.* Proceedings of the 34th ACM/IEEE Design Automation Conference, 1997, 526 - 529.

[67] Khatkate, A., Li, C., Angihotri, A.R., Yildiz, M.C., Ono, S., Koh, C.-K, Madden, P. [2004]: *Recursive bisection based mixed block placement,* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2004, 84–89.

[68] Kleinhans, J.M., Sigl, G., Johannes, F.M., 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.

[69] Kleinschmitt, H.P. Schannath, H. [1995]: *A strongly polynomial algorithm for the transportation problem.* Mathematical Programming 68,1995, 1–13.

[70] Korte, B., Vygen, J. [2002]: *Combinatorial Optimization: Theory and Algorithms.* Springer, Berlin, 2000, second edition 2002.

[71] Kruskal, J.B. [1956]: *On the shortest spanning subtree of a graph and the traveling salesman problem.* Proceedings of the American Mathematical Society 7, 1956, 48–50.

[72] Landman, B., Russo, R. [1971]: *On a pin versus block relationship for partitions of logic graphs.* IEEE Transactions on Computers, 20, 1971, 1469–1479.

[73] Liu, Q., Marek-Sadowska, M. [2004]: *A study of netlist structure and placement efficiency.* ACM/IEEE Proceedings of the International Symposium in Physical Design, 2004, 198–203.

[74] Lou, J., Krishnamoorthy, S., Sheng, H.S. [2002]: *Estimating routing congestion using probabilistic analysis.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 21, no. 1, 2002, 32–41.

[75] Matsui, T. [1993]: *A linear time algorithm for the Hitchcock transportation problem with fixed number of supply points* Technical Report, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 1993.

[76] Mayrhofer, S., Lauther, U. [1990]: *Congestion-driven placement using a new multi-partitioning heuristic.* Proceedings of the International Conference on Computer–Aided Design, 1996, 332–335.

[77] Mo, F. Tabbara, A., Brayton, R.K. [2000]: *A force-directed macro-cell placer.* Proceeding of the International Conference on Computer-Aided Design, 2000, 404–407.

[78] Müller, D. [2002]: *Bestimmung der Verdrahtungskapazitten im Global Routing von VLSI-Chips.* Diploma thesis, University of Bonn, 2002.

[79] Murata, H., Fujiyoshi, K., Nakatake, S., Kajitani, Y. [1995]: *Rectangle-packing-based module placement.* Proceedings of the International Conference on Computer-Aided Design, 1995, 472–479.

[80] Nam, G.-J., Alpert, C.A., Villarubia, P.G., Winter, B., Yildiz, M. [2005]: *The ISPD 2005 placement contest and benchmark suite.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2005, 216–220.

[81] Obermeier, B., Ranke, H., Johannes, F.M. [2005]: *Kraftwerk - A versatile placement approach.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2005, 242–244.

[82] Onodera, H., Taniguchi, Y., Tamaru, K. [1991]: *Branch-and-bound placement for building block layout.* Proceedings of the 28th ACM/IEEE Design Automation Conference, 1991, 433–439.

[83] Orlin, J.B. [1993]: *A faster strongly polynomial minimum cost flow algorithm.* Operations Research 41, 1993, 338–350.

[84] Parakh, P.N., Brown, R.B., Sakallah, K.S. [1998]: *Congestion driven quadratic placement.* Proceedings of the 35th ACM/IEEE Design Automation Conference, 1998, 275–278.

[85] Pauli, A. *Bewegungsminimale Legalisierung von Platzierungen im VLSI-Design.* Diploma thesis, University of Bonn, 2003.

[86] Queyranne, M., [1986]: *Performance ratio of polynomial heuristics for triangke inequality quadratic assignment problems.* Operations Resear Letters 4, 1986, 231–234.

[87] Rao, S., Richa, A.W. [1998]: *New approximation techniques for some ordering problems.* Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, 211–218.

[88] Ren. H., Pan, D.Z., Alpert, C.J., Villarubia, P. [2005]: *Diffusion-based placement migration.* Proceedings of the 42th ACM/IEEE Design Automation Conference, 2005, 515–520.

[89] Roy, J.A., Papa, D.A., Adya, S.N., Chan, H.H., Ng, A.N., Lu, J.F., Markov, I.L. [2005]: *Capo: Robust and scalable open-source min-cut floorplacer.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2005, 224–226.

[90] Sait, S.M., Youssef, H. [1999]: *VLSI Physical Design Automation.* Singapore: World Scientific, 1999.

[91] Sarrafzadeh, M., Wang, M. [1997]: *NRG: global and detailed placement.* Proceedings of the International Conference on Computer-Aided Design, 1997, 532–537.

[92] Sarrafzadeh, M., Wang, M., Yang, X., [2003]: *Modern Placement Techniques.* Kluwer Academic Publishers, 2003.

[93] Sarrafzadeh, M., Wong, C.K. [1996]: *An Introduction to VLSI Physical Design.* New York: McGraw-Hill, 1996.

[94] Sechen, C. [1988]: *VLSI Placement and Global Routing Using Simulated Annealing.* Kluwer Academic Publishers, 1988.

[95] Sechen, C., Lee, K.-W. [1987]: *An improved simulated annealing algorithm for row-based placement.* Proceedings of the International Conference on Computer–Aided Design, 1987, 478–481.

[96] Sechen, C., Sangiovanni-Vincentelli, A. [1986]: *TimberWolf3.2: A new standard cell placementand global routing package.* Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, 432–439.

[97] Sherwani, N. [1998]: *Algorithms for VLSI Physical Design Automation - 3rd Edition.* Kluwer Academic Publishers, 1998.

[98] Sun, W.-J., Sechen, C. [1995]: *Efficient and effective placement for very large circuits* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 14, no. 3, 1995, 349–359.

[99] Sun, W.-J., Sechen, C. [1997]: *A parallel standard cell placement algorithm.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 16, no. 11, 1997, 1342–1357.

[100] Taghavi, T., Yang, X., Choi, B.-K. [2005]: *Dragon2005: Large-scale mixed-size placement tool.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2005, 245–247.

[101] Takahashi, T. [2000]: *A new encoding scheme for rectangle packing problem.* Proceedings of Asia South Pacific design automation design conference, 2000, 175–178.

[102] Tokuyama, T., Nakano, J. [1991]: *Geometric algorithms for a minimum cost assignment problem.* Proceedings of the Seventh Symposium on Computational Geometry, 1991, 262–271.

[103] Tokuyama, T., Nakano, J. [1992]: *Efficient algorithms for the Hitchcock transportation problem.* Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, 175–184.

[104] Tokuyama, T., Nakano, J. [1995]: *Efficient algorithms for the Hitchcock transportation problem.* SIAM Journal on Computing 24, 1995, 563–578.

[105] Tsay, R.-S., Kuh, E.S., Hsu, C.-P. [1988]: *PROUD: A fast sea-of-gates placement algorithm.* Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, 318–323.

[106] Viswanathan, N., Pan, M., Chu, C. C.-N. [2005]: *FastPlace: An analytical placer for mixed-mode designs.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2005, 221–223.

[107] Viswanathan, N., Chu, C. C.-N. [2004]: *FastPlace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2004, 26–33.

[108] Vorwerk, K., Kennings, A., Vannelli, A. [2004]: *Engineering details of a stable force-directed placer.* Proceedings of the International Conference on Computer–Aided Design, 2004.

[109] Vygen, J. [1996]: *Plazierung in VLSI-Design und ein zweidimensionales Zerlegungsproblem.* Doctoral thesis, University of Bonn, 1996.

[110] Vygen, J. [1997]: *Algorithms for large-scale flat placement.* Proceedings of the 34th ACM/IEEE Design Automation Conference, 1997, 275–278.

[111] Vygen, J. [1998]: *Algorithms for detailed placement of standard cells.* Design, Automation and Test in Europe, Proceedings, IEEE, 1998, 321–324.

[112] Vygen, J. [2000]: *Geometric quadrisection in linear time with application to VLSI placement.* Accepted for publication in "Discrete Optimization".

[113] Vygen, J. [2002a]: *New theoretical results on quadratic placement.* Report No. 02920-OR, Research Institute for Discrete Mathematics, University of Bonn, 2002. Accepted for publication in "Integration".

[114] Vygen, J. [2002b]: *On dual minimum cost flow algorithms.* Mathematical Methods of Operations Research 56, 2002, 101–126.

[115] Wang, M., Sarrafzadeh, M. [1999]: *On the behavior of congestion minimization during placement.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 1999, 145–150.

[116] Wang, M., Sarrafzadeh, M. [2000]: *Modeling and minimization of routing congestion.* Proceedings of the conference on Asia and South Pacific design automation, ACMPress, 2000, pages 185–190.

[117] Wang, M., Yang, X., Eguro, K., Sarrafzadeh, M. [2000]: *Multi-center congestion estimation and minimization during placement.* ACM/IEEE Proceedings of the International Symposium on Physical Design, 2000, 147–152.

[118] Wang, M., Yang, X., Sarrafzadeh, M. [2000a]: *Congestion minimization during placement.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2000, 19, 1140–1148.

[119] Wang, M., Yang, X., Sarrafzadeh, M. [2000b]: *Dragon2000: Standard-cell placement tool for large industry designs.* Proceedings of the International Conference on Computer-Aided Design, 2000, 260–263.

[120] Warme, D.M., Winter, P., Zachariasen, M. [2000]: *Exact algorithms for plane Steiner tree problems: a computational study.* In: Du, D.Z., Smith. J.M., Rubinstein. J.H. (Eds.): Advances in Steiner Trees, Kluwer Academic Publishers, 2000, 81–166.

[121] Wong, D.F., Leong, H.W., Liu, C.L. [1988]: *Simulated Annealing for VLSI Design.* Kluwer Academic Publishers, 1988.

[122] Xiu, Z., Ma, J., Fowler, S.M., Rutenbar, R.A. [2004] *Large-scale placement by grid-warping.* Proceedings of the 41st ACM/IEEE Design Automation Conference, 2004, 351–356.

[123] Xiu, Z., Rutenbar, R.A. [2005] *Timing-driven placement by grid-warping.* Proceedings of the 42nd ACM/IEEE Design Automation Conference, 2005, 585–590.

[124] Yang, X., Choi, B.-K., Sarrafzadeh, M. [2003]: *Routability-driven white space allocation for fixed-die standard-cell placement.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 22, 2003, 410–419.

[125] Yang, X., Kastner, R., Sarrafzadeh, M. [2001]: *Congestion reduction during placement with provably good approximation bound.* Proceedings of the International Conference on Computer–Aided Design, 2001.

[126] Yang, X., Kastner, R., Sarrafzadeh, M. [2002] *Congestion estimation during top-down placement.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2002, 21, 72–80.

[127] Yildiz, M.C., Madden, P.H. [2001]: *Improved cut seqences for partitioning based placement* Proceedings of the 38th ACM/IEEE Design Automation Conference, 2001, 776–779.

Appendix A

Notation Index

cost_f	Residual edge costs (see Section 2.1).
$\text{dist}_i((x_1, y_1), (x_2, y_2))$	For two pairs $(x_1, y_1), (x_2, y_2) \in \mathbb{R}^2$ and $i \in \mathbb{N} \setminus \{0\}$, $\text{dist}_i((x_1, y_1), (x_2, y_2)) := \left(\lvert x_1 - x_2\rvert^i + \lvert y_1 - y_2\rvert^i\right)^{\frac{1}{i}}$ is the L_i-*distance* between (x_1, y_1) and (x_2, y_2).
$\delta_G(v)$	Set of edges incident to v (see Section 2.1).
$\delta_G^+(v)$, $\delta_G^-(v)$	Set of edges in $E(G)$ leaving (entering) v (see Section 2.1).
$E(G)$	The edge set of a graph G (see Section 2.1).
$f\lvert_X$	For a function $f : A \to B$ with $X \subseteq A$, the function $f\lvert_X : X \to B$ is the restriction of f to X, so $f\lvert_X(a) = f(a)$ for $a \in X$.
$G_{f,u}$	Residual graph (see Section 2.1).
$\mathbb{N}$	The set of the non-negative integers (including 0).
NP	Class of the nondeterministically polynomially solvable descision problems, see Garey and Johnson [1979].
NP-complete, NP-hard	See Garey and Johnson [1979].
$O(f)$	"O-notation" to describe the asymptotical growth of a function, see, e.g., Cormen, Leiserson, and Rivest [1990].
$o(f)$	"o-notation" to describe the asymptotical growth of a function, see e.g., Cormen, Leiserson, and Rivest [1990].

P	Class of the polynomially solvable decision problems, see Garey and Johnson [1979].
$\mathbb{R}_{\geq 0}$	The set of all non-negative real numbers.
$\mathbb{R}_{>0}$	The set of all positive real numbers.
$\Theta(f)$	"Θ-notation" to describe the asymptotical growth of a function, see, e.g., Cormen, Leiserson, and Rivest [1990].
u_f	Residual edge capacity (see Section 2.1).
$V(G)$	The vertex set of a graph G (see Section 2.1).
2^X	The set of all subsets of the set X.
$\binom{X}{k}$	The set of all subsets of the set X with exactly k elements (for $k \in \mathbb{N}$).

Summary

Placement is a crucial step in the design process of VLSI chips as it has a strong influence on the most important optimization goals, e.g., die size, routability, and cycle time. State-of-the-art logic chips consist of several millions of modules (circuits) that have to be placed, so efficient automated tools are mandatory for that task. In this thesis, we describe new ideas for VLSI placement that are combined in the placement tool BonnPlace which has been used by IBM Microelectronis for the design of many challenging logic chips.

The main contributions of this thesis can be summarized as follows:

From a very global point of view, BonnPlace consists of three parts: a *macro placement* phase in which the largest circuits are placed, a *global placement* phase in which the remaining circuits are spread over the chip area, and a *legalization* phase in which all overlaps between the circuits are removed and all technological constraints are met.

As we follow a recursive top-down partitioning approach for global placement, we have to distribute several times a (possibly large) set of circuits to a small number of regions. This motivates the consideration of the Transportation Problem which is an uncapacitated minimum cost flow problem on a bipartite graph where one side of the bipartition contains all supply nodes and the other side all demand nodes. In Chapter 4, we present a new algorithm that solves this problem to optimality in time $O(nk^2(\log n + k \log k))$ where n and k are the numbers of elements in the two sides of the bipartition. This is the fastest known algorithm on instances with $k \log k = O(\log n)$. If k is constant (which is the case in the instances we are interested in), we improve the running time of the fastest algorithm by a factor of $\Theta(\log n)$. Moreover, the algorithm is easy to implement and performs well in practice: instances with $n = 2\,400\,000$ and $k = 4$ can be solved in less than half a minute.

Our global placement algorithm makes use of the transportation algorithm in several ways. Similar to previous approaches, we start with a placement minimizing total squared netlength but ignoring disjointness. Then, we subdivide the chip area into subregions and assign each circuit to one of them such that no subregion contains more circuits than fit into it. As we want to change the initial placement as little as possible, our main goal is to minimize total movement of the circuits when they are moved to their regions. The partitioning steps are repeated recursively for the regions until the regions are small enough. Larger objects (macros) are simply fixed at their recent positions as soon as they are too big compared to the actual region size. This idea has already been described by Vygen [1996] but his algorithm is restricted to four subregions and has to use the L_1-distances as movement costs. In contrast, we may choose the number of areas and the movement costs arbitrarily. In Chapter 5, we propose a number of ideas how this flexibility can be exploited in the

global partitioning steps. The transportation algorithm is also applied in local optimization steps on sliding windows (*repartitioning*) where we can consider bigger windows compared to Vygen [1996]. Moreover, we present an iterative partitioning approach that reduces the number of necessary repartitioning steps. Further on, we introduce a hybrid net model that combines the positive aspects of the previously used net models STAR and CLIQUE. We also describe how our placer can be parallelized. The efficiency and effectiveness of these improvements is shown by experiments at the end of Chapter 5.

After global placement, all macros are fixed and the remaining circuits (*standard circuits*) are spread over the chip area. These standard circuits have the same height and have to be placed disjointly in given rows which is done in the legalization phase. In Chapter 6, we propose a legalization method that works in three steps. As we assume that the placement given to the legalization is well optimized, we try to minimize total movement of the circuits during legalization. In the first legalization phase, the chip area is divided into small regions and a minimum cost flow problem is stated in order to move circuits from regions which are too full into regions with free capacity. In contrast to previous minimum cost flow formulations, we do not only consider single regions in the minimum cost flow instance but also sets of horizontally neighbouring regions. Considering only single regions would mean that we ask for an assignment of the circuits to the regions such that each circuit can be placed completely within its region which would cause unnessecary movement. We can show that our minimum cost flow formulation is best possible in a natural, well-defined way. Moreover, we prove that the number of the sets of regions that we have to take into account is linear in the number of regions and that these sets can be computed in linear time. In the second phase of legalization, we apply to each row a well-known linear-time algorithm that legalizes the placement of the circuits in the zone with minimum squared movement (without allowing to change the order of the circuits in the row). The last phase is a postoptimization based on dynamic programming that tries to reduce the largest movements of the circuits.

The experiments that we present in Chapter 6 demonstrate that our legalization algorithm can reduce movement drastically compared to a similar legalization algorithm that only considers single regions in its minimum cost flow formulation. Moreover, we can compare our results to lower bounds that we compute by solving a relaxation of an integer linear program formulation of the legalization problem. We can show that, at least on instances where the given placement was spread out well enough, the circuit movement in our legalization is quite close to the optimum.

Our algorithms for global placement and legalization are together a complete placement tool that could be applied to any VLSI placement instance. However, the global placer is mainly designed for instances consisting of a large number of small circuits. On instances with bigger objects it may produce weak results. A new and more sophisticated approach to macro placement is described in Chapter 7. According to their size we divide the set of macros to be placed into three classes and for each class we follow a different strategy. The first class contains at most five of the largest objects. We define some candidate positions for these objects and place them by enumerating possible assigments to the candidate positions. Afterwards, we fix them for the rest of the placement process. The second class consists of the medium-sized macros. They are "shredded" into small pieces, and these macro fragments (and all other circuits) are placed by running some steps of our global

placement algorithm. In order to keep the fragments of a single macro together, they are connected by artificial nets with high weights. The center of gravity of its fragments is used as a desired location of the macro, and we apply a branch-and-bound algorithm (considering small subsets of the macro set at a time) to place the macros close to their desired locations. After all macros of the second class are fixed, we place the remaining very small macros (third class) by a complete run of our global placer. To compare the results of our macro placer, we consider macro placements that are the result of a global placement run (as described above) and macro placements that were found manually by experienced designers. The experiments show that our new method reduces netlength on average by 9.0% (compared to the standard global placement run with floating macros) and 4.9% (compared to manual preplacement).

In our global placement routine, we mainly try to minimize the total length of all nets connecting the circuits. Nevertheless, it should be noted that after placement the terminals of the nets have to be connected by wires (*routing*) with a number of constraints, especially concerning minimum distances between the wires. Therefore, a very dense placement may not be routable though its netlength may be short. In Chapter 8, we describe how our global placer can keep routing congestion under control by using a fast but reliable method to find routing-critical regions. As soon as a routing problem is detected, the placement density is reduced in the critical area. For the density reduction we apply a local optimization algorithm similar to repartitioning. We present experiments which show that this feature allows to use much higher density in the uncritical areas and hence helps to reduce total netlength significantly.

Chapter 9 concludes this thesis with several experiments that compare our placement tool as a whole to previous approaches. We use both artificial benchmarks and real-word chips for these experiments. On the real-world chips we compare to a previous version of BonnPlace combining the global placement algorithm proposed by Vygen [1996] and a legalization method as described by Vygen [1998]. On the 14 chips used for the experiments, we can improve netlength by 4.8% on average while the running time is reduced by more than a factor of 2. With a faster version we can improve the running time even by a factor of 4 (producing results similar to the results of the old BonnPlace). The faster version allows to place a chip with 3.6 million circuits in three hours and a half on our fastest machine, so we can place more than 1 000 000 circuits per hour. In the experiment on the artificial benchmarks, we produce either the best published results (PEKO and ISPD 2002 benchmarks) or at least the second best results (ISPD 2005 benchmarks). Note that on these benchmarks (where netlength is the only optimization goal) we compare to academic placers which are tuned for placing with the highest possible density while our placer is designed to find routable placements on large industrial designs.