Algorithms for Large–Scale Flat Placement

Jens Vygen
Research Institute for Discrete Mathematics, University of Bonn
Lennestr. 2, 53113 Bonn, Germany

Abstract

This is a survey on the algorithms which are part of a program for flat placement of large–scale VLSI processor chips. The basis is a quadratic optimization approach combined with a new quadrisection algorithm. In contrast to most previous quadratic placement methods, no min–cut objective is used at all. Based on a quadratic placement, a completely new algorithm finds a four-way partitioning meeting capacity constraints and minimizing the total movement.

1 Introduction

One reason for the dominance of hierarchical methodologies in deep submicron designs is the lack of optimization algorithms which are able to cope with large–scale flat designs. In cooperation with IBM Böblingen, the Research Institute for Discrete Mathematics in Bonn has developed algorithms for the flat design of high performance processor chips. The resulting programs have been used very successfully, e.g. for the design of the chips of the IBM S/390 Parallel Enterprise Server – Generation 3 (see [5]). The purpose of this paper is to give an outline of the algorithms which go to make up the placement program.

Let us clarify some notations first. In this paper we use the term circuit for a movable object. (Other authors speak of cells, components or modules.) We assume that most of the circuits have some standard height. These are called standard cell circuits. Other circuits are called macros. Some of the circuits may be fixed in advance; they are called preplaced circuits. See Figure 3 on the last page of this paper for a typical placement.

The basis of the placement program presented here is a well–known quadratic optimization approach (Section 2) combined with a completely new quadrisection algorithm (Section 3).

Most previous placement methods based on quadratic optimization switch to a kind of min–cut objective during partitioning (see e.g. [4]). Here the placement obtained by the minimization of the quadratic netlength estimation merely serves as a starting solution for min–cut heuristics. Our method is completely different. In fact, the quadrisection does not consider the netlist at all, but partitions the set of circuits just based on the positions obtained by the quadratic optimization. Furthermore it is crucial to do a four–way partitioning (quadrisection) in one step, instead of three bipartitioning steps.

The overall structure of our placement program will be presented in Section 4, after the two main ingredients have been explained. The framework, assigning the circuits gradually to smaller and smaller regions (induced by a finer and finer grid) is commonly known. However, some new ideas are introduced. For example, the way the quadratic placement takes the current assignment of the circuits to the regions into account is different from previous methods.

Moreover, new algorithms have been developed for the detailed placement of the macros (Section 5) and the standard cell circuits (Section 6). Section 7 contains some concluding remarks, especially on the interaction with other physical design tools.

2 Quadratic Optimization

The first step consists of finding a (non–disjoint) placement which minimizes a certain quadratic netlength estimation. Let N be the set of nets. Each net n is a set of pins and has a weight \( w(n) \). (The weight of a net reflects its timing criticality; see Section 7). For each circuit c, two variables \( x(c), y(c) \) are introduced representing the coordinates of (the center of) the circuit. Of course, if the circuit is preplaced (e.g. IO–pads), these variables have fixed values. For a pin p, let \( c(p) \) denote the circuit to which p belongs, and let \( xofsp(p), yofsp(p) \) denote the relative (x– and y–) offset of the pin to the center of its circuit. (This is a simplification because usually a pin is not a single point.) We solve

\[
\min \sum_{n \in N} \frac{w(n)}{|n| - 1} \sum_{p,q \in n} sqd(p, q),
\]

where \( sqd(p, q) \) is an abbreviation for the square of the Euclidian distance of \( p \) and \( q \), i.e.

\[
(x(c(p)) + xofsp(p) - x(c(q)) - xofsp(q))^2 +
(y(c(p)) + yofsp(p) - y(c(q)) - yofsp(q))^2.
\]

So the standard clique model is used. The factor \( \frac{1}{|n| - 1} \) prevents nets of large cardinality from dominating the objective function. Very large nets are modelled by a star (one Steiner point connected to all the pins) in order to bound the number of summands in (1).
The solution of (1) yields coordinates \( x(c), y(c) \) for all circuits \( c \). Let us call this procedure a quadratic placement (QP).

One may ask why a quadratic objective function instead of a linear one is used. There are three reasons for this. First of all, (1) can be solved very efficiently with the conjugate gradient method (the problem is convex); see \([2],[4],[8]\). A linear objective function is much harder to optimize (although this can be done in a combinatorial way, see Section 5).

The second reason is that a quadratic objective function leads to very few long connections. As an example, consider an inverter \( b \) connected to placed circuits \( a \) and \( c \). Optimizing (1), \( b \) will lie at the center between \( a \) and \( c \). With a linear objective function, any position within the rectangle spanned by \( a \) and \( c \) would be optimal.

Table 1 illustrates this further, based on the placement of the PU chip of the IBM S/390 Parallel Enterprise Server – Generation 3 [5] (obtained by the program presented here). This design has a 14.6mm image and contains 180'846 nets. The total bounding-box netlength is 106.34m. More than 70% of the nets have a bounding box whose width plus height is less than 0.5mm; less than 1% are longer than 5mm. For other chips the figures are very similar.

Table 1: Bounding-box netlength distribution

<table>
<thead>
<tr>
<th>Netlength</th>
<th>Total Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0-0.5mm</td>
<td>129'624</td>
</tr>
<tr>
<td>0.5-1.0mm</td>
<td>22'926</td>
</tr>
<tr>
<td>1.0-1.5mm</td>
<td>8'980</td>
</tr>
<tr>
<td>1.5-2.0mm</td>
<td>5'999</td>
</tr>
<tr>
<td>2.0-2.5mm</td>
<td>4'127</td>
</tr>
<tr>
<td>2.5-3.0mm</td>
<td>2'877</td>
</tr>
<tr>
<td>3.0-3.5mm</td>
<td>2'050</td>
</tr>
<tr>
<td>3.5-4.0mm</td>
<td>1'456</td>
</tr>
<tr>
<td>4.0-4.5mm</td>
<td>1'086</td>
</tr>
<tr>
<td>4.5-5.0mm</td>
<td>706</td>
</tr>
<tr>
<td>5.0-6.0mm</td>
<td>933</td>
</tr>
<tr>
<td>6.0-7.0mm</td>
<td>396</td>
</tr>
<tr>
<td>7.0-8.0mm</td>
<td>162</td>
</tr>
<tr>
<td>8.0-9.0mm</td>
<td>64</td>
</tr>
<tr>
<td>9.0-10.0mm</td>
<td>28</td>
</tr>
<tr>
<td>10.0-15.0mm</td>
<td>60</td>
</tr>
<tr>
<td>15.0-20.0mm</td>
<td>6</td>
</tr>
<tr>
<td>more than 20mm</td>
<td>0</td>
</tr>
</tbody>
</table>

There is a third reason for optimizing a quadratic objective function: After optimizing (1), it empirically almost never happens that two circuits lie on exactly the same position. This is in contrast to linear programs where usually many circuits would end up with exactly the same coordinates. The property that the circuits can be distinguished by their positions only is important for the subsequent step, the quadrisection.

### 3 Quadrisection

The second step consists of partitioning the set of circuits to four subsets, each corresponding to one part of the chip area. So let two cut coordinates \( xcut, ycut \) be given which partition the chip area into four rectangular regions \( R_0 := \{(x, y) \mid x \leq xcut, y \leq ycut\}, R_1 := \{(x, y) \mid x \geq xcut, y \leq ycut\}, R_2 := \{(x, y) \mid x \geq xcut, y \geq ycut\}, R_3 := \{(x, y) \mid x \leq xcut, y \geq ycut\} \).

(See Figure 1.)

Let \( C \) denote the set of (non–preplaced) circuits in consideration. For each circuit \( c \) we denote by \( \text{size}(c) \) the area it occupies.

We compute the capacity \( \kappa_i \) of each region \( R_i \) by subtracting the area occupied by preplaced circuits from the total area of \( R_i \), and – to avoid routability problems (see Section 7) – multiplying the result by a user–defined density parameter (usually between .6 and .9). In other words, the capacity of a region is the maximum total size of circuits we want to allow in this region.

Now we look for a partition \( f : C \rightarrow \{0, 1, 2, 3\} \) (i.e. an assignment of the circuits to the regions) meeting the capacity constraints

\[
\sum_{c \in C, f(c) = i} \text{size}(c) \leq \kappa_i
\]

for \( i = 0, 1, 2, 3 \). Such a partition will be called feasible.

The question is how to measure the quality of a partition. The measure used by most previous methods is the number of cut nets, i.e. the number of nets containing pins \( p, p' \) with \( f(c(p)) \neq f(c(p')) \). Here the previous QP is used only to produce a starting solution for iterative improvement min–cut algorithms: If the position \( (x(c), y(c)) \) of a circuit \( c \) after the QP is in \( R_i \), one starts with \( f(c) = i \).

Our approach is completely different. It does not consider the netlist at all. Rather it tries to move the circuits – starting from their positions after the QP – as little as possible in order to meet the capacity constraints of the four regions.

The partition obtained by \( (x(c), y(c)) \in R_i \Rightarrow f(c) = i \) is said to have total movement zero. If it is feasible, it is considered optimum. Usually we will not be that lucky. As a measure of the quality of a partition we take the total movement

\[
\sum_{c \in C} \text{size}(c) d ((x(c), y(c)), R_{f(c)})
\]

where

\[
d ((x, y), R_i) := \min_{(x', y') \in R_i} \left( |x - x'| + |y - y'| \right)
\]

denotes the \( L_1 \)–distance.

The motivation of this unfamiliar objective function is the following: We believe that the result of the QP is a good placement, whose only problem is that circuits are overlapping. In any top–down placement approach the overlapping is gradually reduced by meeting capacity constraints for smaller and smaller regions. So there seems to be no reason to change the placement more than necessary in order to meet the capacity constraints of the current subregions.

A min–cut objective is something completely different than the minimization of some netlength estimation; often the two goals are conflicting. Therefore the common approach, which starts with a QP but then switches to a min–cut objective, appears to be a bit inconsistent.
Now that we motivated our choice of the objective function, we come to the question how to find a feasible partition with minimum total movement.

Of course, even the problem of deciding whether a feasible partition exists is NP-complete. To obtain an efficient algorithm, we relax the problem and allow fractional partitions. This means that we consider the following problem:

Find a fractional partition \( g : C \times \{0, 1, 2, 3\} \rightarrow [0, 1] \) with
\[
\sum_{i=0}^{3} g(c, i) = 1 \quad \text{for all } c \in C, \text{ meeting the capacity constraints}
\]
\[
\sum_{c \in C} \text{size}(c) g(c, i) \leq \kappa_i
\]
for \( i = 0, 1, 2, 3, \) and minimizing the total movement
\[
\sum_{c \in C} \text{size}(c) \sum_{i=0}^{3} g(c, i) d(\{x(c), y(c)\}, R_i).
\]

Solving this relaxed problem is sufficient. Namely it can be shown that, given any optimum fractional partition \( g \), another optimum fractional partition \( g' \) with at most three circuits being split up can be derived easily. For all but three circuits \( g' \) will have integral values. Thus a concluding rounding of \( g' \) yields an integral partition \( f \) which is almost feasible and whose total movement is very close to the optimum.

The problem of finding a feasible fractional partition with minimum total movement can be solved optimally in linear time. However, the algorithm is by far too complicated to be presented here; the proof of its correctness has more than 30 pages. Therefore I have to refer to [9]. The algorithm not only has the theoretical worst-case running time, but also performs very efficiently in practice. This algorithm seems to be crucial for the success of our placement program.

Note that the special case of a two-way partitioning (for example \( \kappa_2 = \kappa_3 = 0 \)) is easy: Here the relaxed problem reduces to a weighted median problem. Of course one can find a four-way partition by solving three two-way partitioning problems, but the results obtained this way are by far inferior.

4 Outline of the Program

After describing the two main ingredients, we can now outline the structure of the placement program. After the initial QP and the subsequent quadrissection, the chip area is divided into four rectangular regions, i.e. a 2 x 2-grid. This grid will now be refined to a 4 x 4-grid, then to an 8 x 8-grid and so on. At each stage, the circuits are assigned to the rectangular regions induced by the grid. The procedure stops when the grid is fine enough: The height of the rows equals the standard height of the circuits, and the width of the columns is something like 50 times the minimum width of a circuit. The final detailed placement is done by a special algorithm which is the subject of Section 6.

So in iteration \( k \), we start with a \( 2^k \times 2^k \)-grid and want to refine it in order to obtain a \( 2^{k+1} \times 2^{k+1} \)-grid. (In late stages, the number of regions is not necessarily a power of two, but this does not matter here.)

We start with a QP within the (old) \( 2^k \times 2^k \)-grid. Of course, we have to guarantee that each circuit remains within the region it is assigned to. Instead of introducing linear inequality constraints we achieve this by the following technique: The horizontal part of a connection between two pins \( p \) and \( q \) is split up into two unless the two circuits \( c(p) \) and \( c(q) \) lie in the same column of the grid. Suppose \( c(p) \) lies in the column from \( x_1 \) to \( x_2 \), and \( c(q) \) lies in the column from \( x_3 \) to \( x_4 \), with \( x_1 < x_2 \leq x_3 < x_4 \). Then the horizontal part of \( n_p(d(p, q)) \) is redefined as
\[
(x(c(p)) - x_2)^2 + (x(c(q)) - x_3)^2.
\]

Similar for the vertical part. Figure 2 illustrates this splitting–up technique.

![Figure 2: Splitting up nets at cut coordinates](image)

This splitting–up technique has several advantages. It guarantees that circuits move only within the regions they have been assigned to. It is not necessary to compute a single QP for each region: such a procedure always entails a loss in quality. Rather one QP is done for the whole chip simultaneously, respecting the membership of the circuits in the regions without any loss in efficiency. Of course, the splitting–up changes the objective function implicitly. The effect is that long connections are punished less hard; one can speak of something in between a linear and a quadratic objective function.

After the QP within the (old) \( 2^k \times 2^k \)-grid, cut coordinates are produced in order to refine the grid. Next the quadrissection algorithm is used within each (old) region. This yields a partition of the circuits to the regions of a \( 2^{k+1} \times 2^{k+1} \)-grid. Finally another QP with respect to this new grid is performed.

Before continuing with the next iteration, a certain repartitioning step is important (except in the first iteration). The repartitioning consists of applying the following procedure to each \( 2 \times 2 \)-subgrid of the current grid. (An \( n \times n \)-grid contains \( (n - 1)^2 \times 2 \) subgrids.) All the circuits belonging to the four respective regions are thrown together again. A QP of just these circuits is done, where the circuits may move freely within the union of the four regions. Then a quadrissection of this part follows, yielding a new assignment of the circuits to the four regions. Finally a QP with respect to this new assignment is done. If the original placement was better than the new one (with respect to some weighted netlength estimation), the old one is restored.

One additional feature should be mentioned. It may happen that in a certain region the circuits are all placed in one corner, very close to each other. In such a case, the quadrissection would have insufficient information to obtain a good partition. Therefore, after the QP for each region the center of gravity (cog) of all its circuits is computed. Such a cog is called feasible if there exists a disjoint placement of the circuits within the region whose cog is this point. For each region with an infeasible cog, the cog is moved towards the center of the region until it becomes feasible. Then a second QP is done with these new cog’s being forced. This may give a linear equality constraint for each region, which doesn’t increase the running time dramatically. QPs with cog–constraints have been introduced in [4]. However, in this paper the cog is always prescribed at the center of each region. Our method seems to be superior.
5 Placement of Macros

Although the placement program has been developed for standard cell designs, it can also deal with a certain amount of macros. By a macro we mean a circuit of non–standard height (we assume that most of the circuits have a standard height). Usually it is recommendable (though not absolutely necessary) to preplace very large memory arrays in advance.

The final placement procedure (see next section) can only deal with standard height circuits. So all macros have to be fixed before. Moreover, the quadrisection does not make much sense if there are circuits not yet fixed which are larger than the capacity of any subregion. So at the beginning of each iteration (before the quadrisection) all macros which are “too big” are fixed. Before fixing them, one of course has to guarantee that they are placed disjointly on legal positions. This is the task of a special procedure.

The core of this procedure is an algorithm for finding a disjoint, legal placement of a (small) set of macros within a certain region. Among all possible solutions, the one with minimum total movement is chosen. Before describing this algorithm a little more in detail, let us see how it is applied. The first attempt is to apply it to all regions of the grid. However, there may occur situations where a disjoint placement of the macros within their region is impossible. In such a case, the algorithm is applied to larger regions, e.g. those corresponding to $2 \times 2$–subgrids. Should there still remain some macros which could not be placed, the remaining ones are placed one after another, always choosing the nearest possible legal position. (Usually, this last step is necessary only for very few circuits.)

The main task – placing a set of macros disjointly within a region – is a problem equivalent to a complete placement problem. However, the number of objects to be placed is much smaller. Therefore the possibilities can be enumerated by a branch–and–bound algorithm.

The initial placement is the output of the preceding QP. This is said to have total movement zero. If it happens to be disjoint, it is considered optimal.

But suppose this initial placement contains two overlapping macros $a$ and $b$. There are four possible ways to introduce a linear inequality constraint which guarantees their disjointness: Either $a$ is placed to the left of $b$, or to the right of $b$ or below $b$ or above $b$. Here the process branches and all four possibilities are analyzed. In each case, we have an additional linear inequality constraint which we add to the following linear program (LP):

$$
\min \sum_{c \in M} \left( \pi(c) - \pi(c) + \pi(c) - \pi(c) \right)
$$

s.t.

$$
\pi(c) - \pi(c) \leq \pi(c)
$$

$$
\pi(c) - \pi(c) \leq \pi(c)
$$

$$
\pi(c) - \pi(c) \leq \pi(c)
$$

$$
\pi(c) - \pi(c) \leq \pi(c)
$$

$$
x_{\min} \leq \pi(c) \leq x_{\max}
$$

$$
y_{\min} \leq \pi(c) \leq y_{\max}
$$

where $M$ is the set of macros to be placed, $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ is the region under consideration, and $(\pi(c), \pi(c))$ denotes the original position (the output of the preceding QP). $\pi(c)$ is a variable for the new position of macro $c$. Furthermore there are four auxiliary variables $\pi(c), \pi(c), \pi(c), \pi(c)$ for each macro $c$.

The LP (2) describes the (trivial) problem of minimizing the total movement regardless of disjointness. By adding disjointness constraints it becomes non–trivial.

The solution of each of the four new LPs (after the branching) yields a new placement. These placements are analyzed one after another. If a placement still contains a pair of macros overlapping each other, we continue the process by again adding one of four possible constraints.

All the LPs arising in this process can be solved combinatorially, since they are duals of minimum cost flow problems. Namely, by introducing an auxiliary variable $v_{ij}$ with the value zero, we have that every constraint has the form $v_{ij} - v_{ij} \geq \delta_{ij}$, where $\delta_{ij}$ is a constant and $v_{ij}$ and $v_{ij}$ are variables. Each of these constraints corresponds to a directed edge in a graph and to a dual variable $f_{ij}$.

As the dual LP we then obtain

$$
\max \sum_{i,j} a_{ij} f_{ij}
$$

s.t.

$$
f_{ij} - \sum_{k} f_{jk} = \begin{cases} 
-1 & \text{if } v_{ij} \text{ is either } \pi(c) \text{ or } \pi(c) \\
1 & \text{if } v_{ij} \text{ is either } \pi(c) \text{ or } \pi(c) \\
0 & \text{otherwise}
\end{cases}
$$

for all $i, j$

This is obviously a maximum–cost flow problem. With the fastest known algorithm for minimum–cost flow problems with infinite capacities [7] we can thus solve both the primal and the dual LP in $O(n \log n (m + \log n))$ time. Here $n = |M|$ and $m$ is the number of disjointness constraints.

Although the worst–case running time of such a branch–and–bound algorithm is exponential, the method can be quite fast in practice, especially with a skillful branching strategy. In [6], a similar branch–and–bound method with a static branching rule was proposed. We use a dynamic branching strategy instead: At each step, the pair of circuits is chosen whose overlap increased the most after the previous constraint had been added. The order in which the four constraints are analyzed can also be important. Still it might be necessary to stop the search after a certain number of LP calculations and take the best solution determined so far.

6 Final Placement

The final placement procedure makes use of the row structure of a standard cell chip. All circuits considered here must have standard height (all others must be fixed in advance). The rows of the final grid – to whose regions the circuits are assigned – also have this standard height.

The final placement consists of two main phases. Let us call a maximal part of a row which is not intersected by any fixed object a zone. The first phase determines an assignment of the non–fixed circuits to the zones such that the total width of the circuits assigned to one zone does not exceed the width of this zone.

An initial assignment of the circuits to the zones is done based on the assignment to the regions of the final grid and the positions of the last QP. Now there are typically some zones where too many circuits lie, while others have some free capacity left. It is thus natural to solve a minimum cost flow problem, where the vertices correspond to the zones and edges join adjacent zones. The cost of an edge is an estimation of the cost of moving one unit (e.g. one inverter) from one zone to the other. Of course, the overloaded zones are the sources, while the zones with free capacity are possible sinks.

After having solved the minimum cost flow problem, the flow is realized by actually moving circuits of an appropriate total width from one zone to another for any edge carrying non–zero flow. The circuits to be moved are naturally selected by solving a knapsack problem.

If afterwards there are still zones with too many circuits, the above procedure will be iterated. Additional heuristics guarantee that the method terminates after a few iterations. Experiments
showed that the method works well even if the usage of the chip area is more than 99%.

The second phase of the final placement then finds an optimum disjoint placement (with respect to some linear netlength estimation) such that the circuits remain within their zones and also remain in their horizontal order according to the positions after the last QP. This is again a linear program, and just like the one mentioned in Section 5 it is the dual of a minimum-cost flow problem. So it can be solved quite efficiently. Of course, certain density constraints can also be taken into account.

7 Concluding Remarks

The placement of special circuits, like clock drivers or spare logic is done by specific tools after the main placement. The main placement program described above just ignores these circuits. Of course, some free space is left for later use wherever necessary.

The placement program interacts with a timing evaluation and optimization program (including power level optimization as well as buffer insertion and removal; see [1]). This changes the netlist and updates net weights. Initially the net weights are usually set uniformly to one (unless there is some information that certain nets should be short). After each call of the timing program the net weights are updated according to the worst-case slack of a path containing that net.

By now the placement program does not interact with a routing program (see [3]). To avoid wirability problems, the user has several options to control the density of the placement. For example, one can place certain routing–critical parts of the logic less densely.

The running time of the placement program (without timing optimization) is about six hours on an IBM RS/6000 595 for a chip with 200'000 circuits and 200'000 nets. (This is about the size of the MBA chip belonging to the chipset described in [5]. Figure 3 on the next page shows a placement of this chip.) A factor of two in the instance size roughly means a factor of three in the running time.

Many people helped by implementing the placement program and contributing ideas, but Christoph Albrecht, Asmus Hetzel and Dieta Schülter are to be mentioned in the first place.

REFERENCES
