Coloquinte: a mixed-size placer for the Coriolis toolchain

Gabriel Gouvine

September 2014-June 2015

Abstract

VLSI synthesis is a set of important optimization problems for industrial applications. As the per-transistor cost is rising with newer processes, improving the synthesis algorithms is one of the ways to keep increasing performance, efficiency and cost with older processes. Coriolis is an open synthesis toolchain developped at the Lip6 laboratory. It targets older processes with the hope of providing a low-cost industrial-grade toolchain.

I developped Coloquinte, an open placement framework integrating with Coriolis. It uses well known placement principles coupled with new algorithms: new global placement methods, legalization tools and detailed placement passes have been tested. During this research, I developped a few new algorithms and applied scheduling algorithms to legalization. Coloquinte's legalization and detailed placement passes are mostly new algorithms and heuristics.

To my knowledge, Coloquinte is the first analytical placement tool to try direct Steiner wirelength optimization.

Acknowledgements

This work would not have been possible without my advisor, Jean-Paul Chaput, who accompanied me before and during this project. He is the lead developper of Coriolis and Kite, Coriolis' detailed router. Thanks to him, Coloquinte was tested on real-world test cases with an industrial-grade routing tool.

I would like to thank Alix Munier as well, who spent was always available to discuss flow and scheduling algorithms. Yves Mathieu, my advisor at Telecom Paristech, gave me his point of view as a circuit designer and provided a lot of research ideas. Marie-Minerve Louërat, the department's head, made this internship possible. Last but not least, Roselyne Chotin-Avot has been my main source regarding logic synthesis, and handed me the FPGA benchmark circuit.

I am grateful to all the graduate students and professors working who have been working around me, and to the administrative employees of the UPMC who were remarkably helpful.

Contents

Chapter 1

Introduction

Due to the continuous progress of digital circuits during the past fifty years, electronic design automation has been necessary to design efficient circuits with millions or billions of transistors.

Placement is a particularly critical step of these design flows. Given a netlist, the goal is to place the cells on the final chip while optimizing for various performance objectives. Its quality has a tremendous influence on the final area and power consumption of the chip.

Coriolis is an open-source toolchain for electronic circuit design, successor to the older Alliance toolchain from the Paris6 LIP6 laboratory. It provides numerous tools to perform the physical design of analog and digital circuits.

I designed a placer for the Coriolis toolchain. It is based both on well-known concepts and on newly discovered algorithms. The Coloquinte placer compares favorably to state-of-the-art placers and hints to new optimization passes and algorithms for modern placement tools.

Chapter 2

Problem presentation

2.1 Design flow

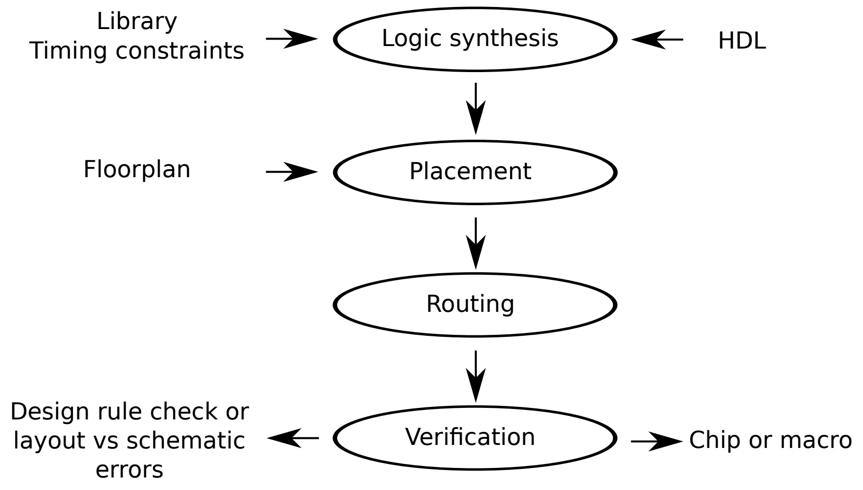


Figure 2.1: A typical digital design flow.

Modern digital circuits are synthetized from a high-level representation. The design flow to translate this specification to a real chip comprises several loosely-dependent steps.

First, the logic synthesis flow translates the RTL representation to real logic gates: it perform high-level and boolean optimizations and technology mapping to the real gate library.

Then, this netlist is placed and routed: the final position of the gates on the chip is fixed, and the wires between them are drawn. This is Coriolis' target.

2.2 The placement problem

2.2.1 Goal

In the traditional flow, the main goal of the placer is to give a highly routable result for the final routing step. It can incorporate other objectives, such as timing optimization, power optimization, thermal limits... All those objective typically relate to the wirelength.

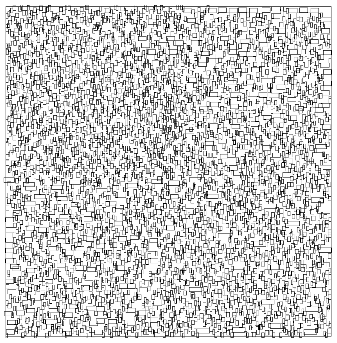
VLSI placement is a difficult problem. Modern chips may contain billions of transistors, and placement instances may contain millions of cells. Moreover, it includes several well-known NP-hard problems, such as the multiple-machine scheduling, the bin-packing problem and the Steiner tree problem.

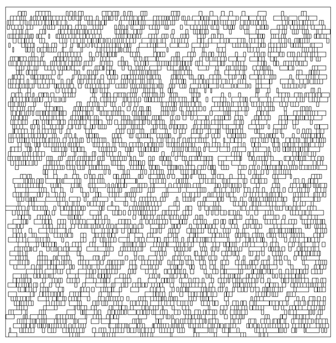
Today's design comprise two types of placeable instances: standard cells and macros. The standard cells are the physical implementations of simple logic gates, and generally have the same height. They are meant to be placed in rows with a uniform vertical pitch. Macros are preplaced blocks, whether preoptimized blocks or special-purpose IPs, and are generally arbitrary big rectangles.

The problem of placing a mix of macros and standard cells is called mixed-size placement. Placers targetting only macroblocks are called floorplanners. This report is about a generic mixed-size placer.

2.2.2 Typical approach

The first algorithms for the placement problem have been simple metaheuristics such as simulated annealing. With the complexity of current integrated circuits, these metaheuristics are not effective anymore: the search space is too large to be explored efficiently. Although tools such as Timberwolf [36] managed to push the limits further than naive simulated annealing, they are not suitable for circuits beyond a few thousands of cells.


(a) A global placement result


(b) A legal placement

Current design tools use three steps to overcome this limitation: an approximate global placement is constructed, followed by an overlap-removal step called legalization, and finally local optimizations during detailed placement.

Global placement is probably the most critical part of a placement tool. Two main scheme have been used: based on continuous optimization and based on graph partitioning. The next steps tend to have a lower impact on solution quality, mostly since local suboptimalities are easily corrected by local search algorithms.

Chapter 3

The Coriolis toolchain and Coloquinte

Coriolis is a toolchain that not only targets research, but is meant to be a fully-capable software for low-cost and academic designs, and for advanced EDA research. Its database, Hurricane, is the backbone of various analog and digital design tools written at the Lip6 laboratory.

3.1 History

3.1.1 The Alliance toolchain

The Lip6 laboratory has been involved in Electronic Design Automation (EDA) for years. Pushed by Alain Greiner and the Lip6 team, the Alliance toolchain is the first outcome of these efforts. It is a complete design flow, comprising a VHDL parser, a logic synthetizer, placement and routing tools, a layout editor and a design rules checker. It provides symbolic standard cell libraries (SXLib), including RAM and ROM libraries.

Development began in the 90s, with the first version released in 1993. Although it was succesful, work on Alliance was discontinued after 2000 due to the lack of financial support.

The Alliance toolchain is still in use today, most notably recently by Graham Petley to draw open standard cell libraries. It is still in use as an educational tool to present design flow principles to UPMC students.

3.1.2 The Hurricane database

However, beginning in 2000, a new toolchain emerged. Pushed by former Bull employees Christian Masson and Rémi Escassut, the Hurricane database (formerly Tsunami) is the core of the current Coriolis toolchain. It was freed by Bull under an LGPL license, and Coriolis itself is released by the Lip6 laboratory under the GPL.

The Hurricane database focuses on an efficient and complete representation of hierarchical designs. Over time, the initial database was extensively modified and improved. A derived version is used at the core of the proprietary NanoXPlore tools. In the laboratory, new tools were added on top of the database, in particular the Mauka placer and the Knik and Kite routers.

3.1.3 The Kite/Knik router

The current lead developper of Coriolis, Jean-Paul Chaput, is the author of Coriolis' router, Kite, the graphical interface and a large part of the database improvements.

Like most routers, Kite uses a two-pass approach to routing. The global router, Knik, uses a fairly classical maze-routing algorithm. On the other hand, Kite's detailed routing approach is unique: it is purely segment-oriented, without any pathfinding algorithm, which are left to the global router.

Kite has an industrial-strength: working on Coriolis means that I can experiment with a complete router, while most academic work is focused on the global router – until 2014, all routing-driven evaluations of placement tools in the literature were focused on global routing results. On the other hand, my interest in placement tools is an opportunity for the toolchain, which doesn't have a good placement tool yet.

3.2 Software architecture

Coriolis is written in a mix of C++ and Python. C++ is used for performance-critical algorithms and Python for interfaces and user control. The graphical interface, including a cell viewer, uses Qt.

This mixed use of Python and C++ makes it easy to compose several algorithms; it makes for powerful and flexible configuration files. Python has about the same role in Coriolis as Tcl has for a large part of the EDA industry.

3.3 The Coloquinte placer

My focus was not only on research: the goal was to replace the old Mauka placer in Coriolis, which was based on simple metaheuristics. Coloquinte must be flexible enough to be part of the Coriolis toolchain, and open source. Therefore, contrary to a lot of previous works, I implemented the whole tool, not only the highly researched global placement part. I took some time to research routing and logic synthesis algorithms as well, although they are not part of this report.

The Coloquinte library is written in C++ and was meant to be standalone. It needs to be interfaced with Coriolis and its database, Hurricane. In Coriolis, Coloquinte is used through the *Etesian* tool, which interfaces with the rest of the Coriolis environment.

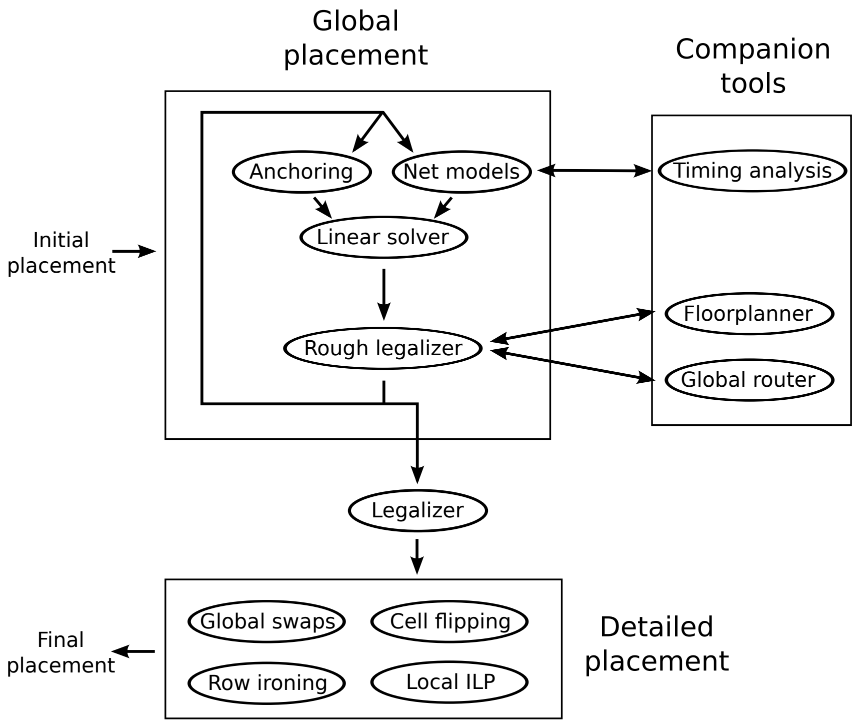


Figure 3.1: A typical flow in Coloquinte; as of now, the only tool interfacing with Coloquinte is the router

It is responsible for accessing the database and the configuration, and for chosing the optimization passes.

It allows to compose the various passes with minimal difficulty: tuning the optimization passes for specific purposes is not part of the Coloquinte library itself. This functionality simplifies algorithm testing and development and keeps the Coloquinte library relatively independent in case other tools need to use it. Moreover, it separates the implementation and the policy and allows the circuit designer or the developper to have easy control over the optimization passes they run.

Chapter 4

Proposed basic algorithms

Part of my work has been to create new algorithms for problems that I encountered when writing the placer.

The cascading descent is an exact algorithm for some simple scheduling problems, which is simpler and faster than previous algorithms for VLSI placement. It was already known by the scheduling community, but the problem was not solved efficiently in VLSI design.

Another algorithm solves a specific version of the transportation problem, which is itself a type of minimu-cost-flow problem. It is used by the global placement and legalization algorithms.

Last but not least, I designed a new algorithm to solve the rectilinear Steiner tree problem. Although this problem already has efficient solutions, my approach makes different tradeoffs in terms of memory use and parallelism.

4.1 Single machine scheduling

The standard cell placement problem bears much similarity with multiple-machine scheduling: the cells are "scheduled" in a set of rows, analog to the machines in the scheduling problem, and occupy a fixed width, analog to their completion time. Although the wirelength cost function is more complex than the ones usually used in scheduling, the problems bear sufficient similarities that scheduling research could extend to standard cell placement.

Since these problems are NP-complete, we must consider simplifications to develop efficient heuristics. One such simplification, which has been used by various detailed placers, is to consider a single row, the other rows being fixed. Although still NP-complete [15], this problem admits much simpler algorithms and easy subproblems. It has been considered by various detailed placers. However, the contributions of the scheduling community have so far been overlooked by the EDA community. I present some applications of their algorithms to standard cell placement.

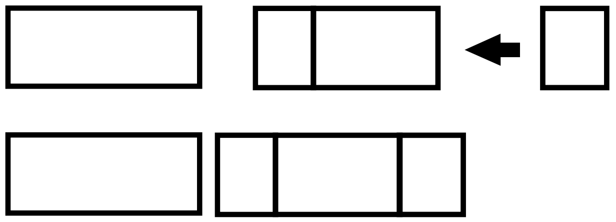


Figure 4.1: All algorithms for the ordered single row problem process the tasks/cells sequentially; the datastructure that represents how other tasks are pushed determines the algorithm's complexity

4.1.1 The ordered single row problem: use of the cascading descent algorithm

The single row problem can be further simplified. The ordered single row problem considers the order of the standard cells to remain fixed. It can be applied to whitespace reallocation in VLSI design, where circuits often are more than 10% whitespace, and has the significant advantage of being easy to solve.

It has been investigated for previous detailed placers [17] with algorithms that turned out to need quadratic runtime. Later, sophisticated algorithms yielded $m \log m \log \log m$ complexity [5], where n is the number of cells and m the cumulated number of slope discontinuities in the cost functions. Those methods are expensive and quadratic algorithms are simpler and fast enough in practice to be used in FastPlace [31].

I rediscovered a simpler algorithm with $m \log m$ complexity. Not only has it better complexity, but it is much faster in practice due to the use of a single priority queue rather than the complex linked structures used by [5]. Moreover, it is applicable to more generic piecewise quadratic cost functions. Although it has never been used for VLSI placement, there is a large litterature body regarding such scheduling problems. This algorithm is already known by the scheduling community as the cascading descent algorithm [32].

It is similar to the previous algorithms in that it adds one task (resp. cell) at the end of the schedule (resp. row), and maintains an optimal schedule at each step. The improvement is the use of a single indexation scheme to track all discontinuities in the cost-function's slope, which makes it possible to use a much faster queue structure.

This algorithm is used in almost every step of the placer, and is one of the big improvements brought by Coloquinte.

```
**Data:** $n$ ordered tasks with durations $d_i$ and piecewise linear cost functions $c_i(t)$
**Result:** Optimal execution times $t_i$
$queue \leftarrow emptyPriorityQueue()$
**foreach** task $T_i$ **do**
    **foreach** slope-discontinuity $t_d$ in $c_i$ **do**
        push($queue$, $(t_D - \sum_{k=1}^{i-1} d_i, \Delta\frac{dc_i}{dt}(t_D))$)
    **end**
    $slope \leftarrow \frac{dc_i}{dt}(+\infty)$
    $t_{ABS}[i] \leftarrow +\infty$
    **while** $slope > 0$ and not empty($queue$) **do**
        $(t_{ABS}[i], s) \leftarrow$ pop($queue$)
        $slope \leftarrow slope - s$
    **end**
    **if** $slope > 0$ **then**
        $t_{ABS}[i] \leftarrow -\infty$
    **else**
        push($queue$, $(t_{ABS}[i], -slope)$)
    **end**
**end**
**foreach** task $T_i$ **do**
    $t_i \leftarrow \min_{k=i+1}^{n} t_{ABS}[i] + \sum_{k=1}^{i-1} d_i$
**end**
```

Algorithm 1: The unconstrained cascading descent algorithm

4.1.2 Non-convex optimization in the ordered single row problem

The difficulty of VLSI placement comes from the non-convexity of the domain and cost function. The algorithm presented in the previous section is optimal for a convex cost-function but does not handle non-convex ones. The scheduling community considered numerous such problems: an algorithm for non-convex cost functions exists for the ordered single row problem [38]. Although it has worst-case quadratic complexity, it is perfectly applicable for local optimizations in VLSI, and even for full-row optimization.

In VLSI design, it allows to optimize for better metrics – the Steiner wirelength is non-convex – and to optimize cell orientation concurrently with whitespace allocation. By using the minimum of the cost function for each orientation of the cells, we can perform whitespace-aware cell flipping within a window of a row.

```
Data: $n$ ordered tasks with durations $d_i$, piecewise linear cost functions $c_i(t)$ and
      range constraints $\left[t_i^{min}, t_i^{max}\right]$
Result: Optimal execution times $t_i$
$queue \leftarrow emptyPriorityQueue()$
$t_{ABS}^{min} \leftarrow -\infty$
foreach task $T_i$ do
    $t_{ABS}^{min} \leftarrow \max\left(t_{ABS}^{min}, t_i^{min} - \sum_{k=1}^{i-1} d_i\right)$
    $t_{ABS}^{max} \leftarrow t_i^{max} - \sum_{k=1}^{i-1} d_i$
    foreach slope-discontinuity $t_D$ in $c_i$ do
        push($queue$, $(t_D - \sum_{k=1}^{i-1} d_i, \Delta\frac{dc_i}{dt}(t_D))$)
    end
    $slope \leftarrow \frac{dc_i}{dt}(+\infty)$
    $t_{ABS}[i] \leftarrow +\infty$
    while not empty($queue$) and ($slope > 0$ or $top(queue) > t_{ABS}^{max}$) do
        $(t_{ABS}[i], s) \leftarrow$ pop($queue$)
        $slope \leftarrow slope - s$
    end
    $t_{ABS}[i] \leftarrow \min(t_{ABS}^{max}, t_{ABS}[i])$
    $t_{ABS}[i] \leftarrow \max(t_{ABS}^{min}, t_{ABS}[i])$
    if $slope > 0$ then
        $t_{ABS}[i] \leftarrow t_{ABS}^{min}$
    else
        push($queue$, $(t_{ABS}[i], -slope)$)
    end
end
foreach task $T_i$ do
    $t_i \leftarrow \min_{k=i+1}^{n} t_{ABS}[i] + \sum_{k=1}^{i-1} d_i$
end
```
Algorithm 2: The cascading descent algorithm can accomodate range constraints

4.2 One-dimensional unbalanced transportation

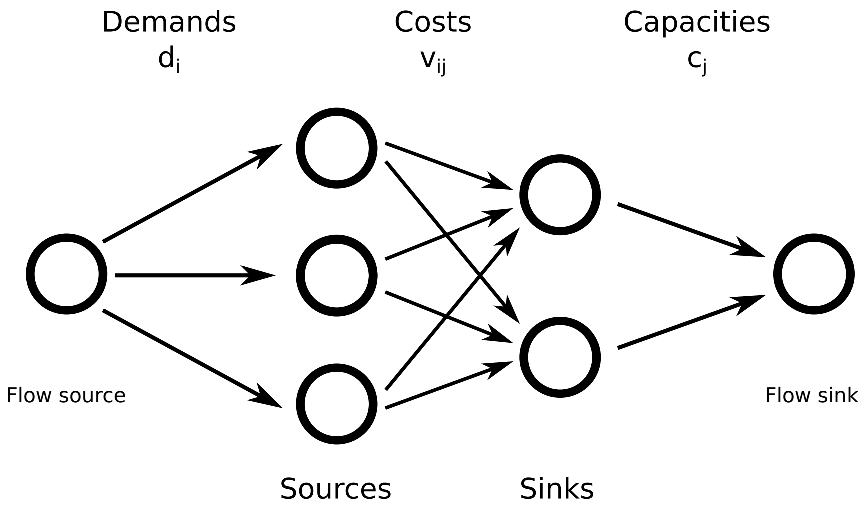


Figure 4.2: The min-cost max-flow formulation of a transportation problem: the edges on the left and right are capacitated with 0 cost, the edges in the middle are uncapacitated

The transportation problem is a classical specialization of the minimum-cost flow problem. It is useful in VLSI design, for example for partitioning during legalization passes. Since it has a cubic complexity, it is not possible to solve it optimally in practice given the huge number of cells in VLSI problems.

A special case arises where the sources and sinks can be optimally allocated in order, that when is the marginal costs $m_{ij} = \frac{c_{ij}}{d_i}$ obey $\forall 1 \leq i < n, \forall 1 \leq j < k, m_{ij+1} - m_{ij} \leq m_{i+1j+1} - m_{i+1j}$. In the balanced case where $\sum_{i=1}^{n} d_i = \sum_{j=1}^{k} c_j$, it is solved trivially by sending the flow of each sink to the first source with available capacity. However, I found no known algorithm for the unbalanced case.

In an even simpler special case, each source or sink is associated to a real position, and the marginal cost is simply the distance between them. Since it was useful for Coloquinte's legalization passes, I implemented a linearithmic algorithm to solve this one-dimensional problem. It is analogous to the cascading descent algorithm for ordered scheduling problems.

4.2.1 Analogy with the cascading descent algorithm

The algorithm adds new sources sequentially, from left to right. Like the cascading descent algorithm, it pushes the already-allocated sources to the left until an optimum is reached. We will show that like for the ordered scheduling problem this method yields an optimal solution.

The optimization of the cascading descent algorithm is to record the slope changes i.e. the variations of the derivative of the cost, and to store them in a single priority queue, with a unified index. In this case, the slope changes happen when a source becomes partially allocated to a new sink. Although the cascading descent algorithm cannot be used directly, I show that a unified index can be defined as well, and that only an amortized constant number of slope changes is non-zero in the 1D distance case.

4.3 Steiner tree algorithm

A Steiner tree is the shortest tree spanning a subset of nodes in a larger graph. A special case is the planar Steiner tree, which is the shortest 2D tree that spans a set of terminals while allowing other crossings. Both are NP-hard problem.

Planar Steiner tree algorithms are a core part of VLSI routing algorithms, and would be useful for optimizing placement algorithms. Fortunately, fast exact algorithms are available for small nets, and good heuristics are known for bigger ones [12].

For Coloquinte, I implemented another algorithm that directly yields a topology suitable for the global placement. It has two useful extensions for VLSI that no other algorithm handles yet: it handles the one-dimensional terminals that are generally found in standard cells, and can penalize one routing track direction. However, the computation speed is not on par with Flute.

For Steiner tree on n terminals, I define an *horizontal topology*, which is a tree on n nodes. Given the horizontal topology, the best associated Steiner tree is easily computed. It turns out that there are relatively few topologies that may produce optimal Steiner trees. In order to compute the optimal Steiner tree of a net, a lookup table of every potentially optimal topology is sufficient.

Compared to Flute, which uses a lookup table as well, Coloquinte apparently tradeoffs more computations for a smaller lookup table. Since the control flow is highly predictable and the simultaneous computation of multiple Steiner trees can be vectorized, the resulting algorithm is not slow compared to Flute.

4.3.1 Horizontal topology and optimal Steiner tree

The horizontal topology is the projection of a Steiner tree on a horizontal axis. It is a tree labelled by the Steiner-tree pins. Given a horizontal topology tree, we can construct the associated Steiner tree with minimum length using the following linear-time algorithm.

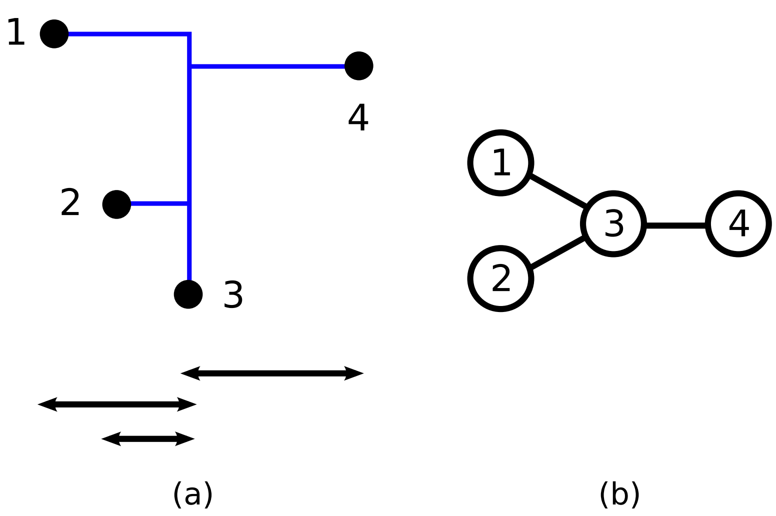


Figure 4.3: A Steiner tree (a), and its associated horizontal topology (b)

Picking a node with at most one unvisited neighbour, the shortest wire to this neighbour is constructed as an L-path, where the vertical part is at the neighbour's position.

4.3.2 Lookup table

All possible horizontal topologies are saved in a lookup table. There are n^{n-2} labeled trees on n nodes, but most of them can be pruned. Like in Flute, the pins are sorted by x coordinate.

Pruning algorithms

When the nodes are labeled following their horizontal ordering, some topologies do not yield optimal Steiner trees, or are redundant.

A first pruning pass considers the number of connexions on each side of a node: if one side has an excess of two connexions or more, the topology cannot be optimal. Moreover, there is a redundancy if the inbalance is only one: in this case, we can arbitrarily select one of the two topologies.

Finally, I used a brute-force algorithm to create the lookup tables, which compares all trees against one-another for every possible vertical order of the pins. This translates to a minimum set cover problem, that I solved greedily rather than optimally. The ordering obtained from the Prüfer sequence used to generate the tree turned out to give the smallest lookup table.

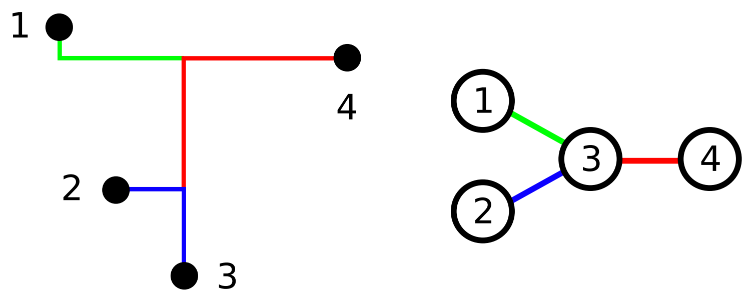


Figure 4.4: The same Steiner tree built from its horizontal topology following the order $2 \to 3$, $4 \to 3$, $3 \to 1$

```
**Data:** $n$ pins with positions $(x_i, y_i)$, an horizontal topology as a labelled tree
**Result:** The wirelength of the associated Steiner tree
foreach Node $i$ do
    $(\text{Mins}[i], \text{Maxs}[i]) \leftarrow y_i$
end
while Several nodes are left do
    Pick a node $i$ with one unvisited neighbour
    $\text{Mins}[i] \leftarrow \min(\text{Maxs}[i], y_i)$
    $\text{Maxs}[i] \leftarrow \max(\text{Mins}[i], y_i)$
end
return $\sum_{i=1}^{n}(\text{Maxs}[i] - \text{Mins}[i]) + \sum_{\text{edge}(k,l)} |x_k - x_l|$
```

Algorithm 3: A horizontal topology directly translates to a possible Steiner tree

Statistics and comparison with Flute

Number of pins	Number of topologies	Table size (B)
4	2	4
5	6	18
6	23	92
7	111	555
8	642	3,852
9	4334	30,308
10	33510	268,080
11	291943	2,627,487

Table 4.1: Statistics of the lookup table

Compared to Flute, the lookup table size is negligible. In Flute, the potentially optimal wirelength vectors (POWVs) alone are used for wirelength calculation, but a table of potentially optimal Steiner trees (POSTs) is needed to get the topology. Both are way bigger than Coloquinte's table for every pin count.

Number of pins	Coloquinte (MB)	Flute POWV (MB)	Flute total (MB)
7	5.5 10^{-5}	0.01	0.03
8	3.9 10^{-3}	0.17	0.48
9	0.03	2.56	8.49
10	0.27	#	#
11	2.62	#	#

Table 4.2: Comparison of the Coloquinte and Flute lookup tables

My hope was that the gains on code predictability and memory locality would be sufficient to improve over Flute, at least at low pin counts. It turns out that it is not the case, even more at high pin counts. At 7 pins and beyond, the need to process the whole lookup table for each net becomes prohibitively expensive. A straightforward optimization is to vectorize the wirelength calculations between several nets, but it is not sufficient to obtain Flute's performance. For high pin counts we could expect an 8x speedup on modern processors using AVX2. The performance was high enough for my use case, and I kept these algorithms so far, but in the long run I will probably use Flute.

Number of pins	Coloquinte runtime (s)	Flute runtime (s)
4	0.08	0.06
5	0.16	0.09
6	0.54	0.13
7	2.85	0.18
8	18.5	0.26
9	145.5	0.55

Table 4.3: Runtime for 1000000 nets on an Intel Haswell at 2.6GHz

4.3.3 Minimum spanning tree for large nets

For large nets, a brute-force lookup table approach consumes too much time and space; in practice, the lookup table is used up to 8 to 10 pins. For bigger nets, a minimum spanning tree algorithm is called and some local optimizations are applied to obtain a Steiner tree.

The rectilinear spanning tree problem can be solved in $n \log n$ time: it suffices to find the nearest neighbour in each octant for each point [13]. Then Prim's or Kruskal's algorithm is applied on this linear number of points. My implementation is based on the sweepline algorithm from [49].

I use its result as an horizontal topology and create the corresponding Steiner tree – eventually performing some local optimizations in the process.

4.3.4 Handling real pin shapes in Steiner and Minimum Spanning Tree routing

In practice, pins don't have a negligible size in standard cell designs. They are generally vertical wires in the first metal layer and span a large proportion of the cell's height. Some nets between nearby standard cells have about the same routed length.

Pins in macros may be much larger, while complex shapes arise in macros and in modern standard cells. Both cases cannot be handled by current Steiner tree algorithms, and pins are typically reduced to their center – which for standard cells is generally a good approximation. I don't know of any efficient heuristic for general pin shapes, whether in the plane or on a graph. This case requires a spanning tree or sequential shortest path algorithm, which is not optimal.

Steiner routing with 1D pins

A first step toward generalized Steiner routing is the case of vertical pins. The horizontal topology applies to those pins directly, although I didn't prove its optimality in this case.

With slight modifications, it could be generalized to rectangular pins as well and used to connect rectilinear shapes. However, the lookup table relies on the abscissa of the pin: taking the abscissa of the pin's center would probably result in another suboptimality.

Minimum Spanning Tree on arbitrary pin shapes

Note that the rectilinear spanning tree algorithm in [49] can be extended to handle rectilinear pin shapes with two additional line sweeps to get the closest neighbours in the four directions.

Chapter 5

Global placement

Global placement is an optimization pass that attempts to find good approximate positions for the cells. It is the most critical part both for runtime and solution quality.

5.1 Prior art

There are two main approaches to global placement tools. Partitioning algorithms attempt to cut the netlist with minimal cost and allocate these subsets to different placement regions. Analytical placement on the other hand uses continuous optimization methods to optimize the wirelength.

Other methods have been proposed, based for example on linear programming [11], but they are less common and I will not review them extensively.

5.1.1 Partitioning-based

Partitioning-based placers attempt to divide the netlist while keeping as few wires as possible between different regions. Capo, NTUPlace1 [43], FengShui [48] and PolarBear are successfull academic placers using this scheme.

The netlist is viewed as an hypergraph, where each cell is a node and each net a hyperedge. The partitioners must solve a hypergraph minimum-cut problem with capacity constraints. This problem is NP-complete even with two regions: they use heuristics to recursively subdivide the netlist, each subset being allocated to a different placement region.

The algorithms used all derive from the Fiduccia-Matheyse heuristic. They were continuously improved and find a good cut in quasi linear time.

Heuristic approach

As of today, the best heuristics for hypergraph partitioning are embodied by the MLPart and HMetis tools [8, 18].

Use of partitioning in global placement

This simple idea needs some tuning: simply partitioning the netlist loses some local information. For example, cells that are not in the region to be partitioned are not taken into account. This problem is solved through terminal propagation. Some advanced partitioning placers use a simple analytical placer to better accomodate wirelength [1].

Extension to routing-driven placement

One of the best partitioning placer, Capo, includes congestion-awareness and Steiner wirelength minimization with advanced algorithms [35].

5.1.2 Analytical placement

Analytical placement tools are based on continuous optimization. Their first optimization objective is generally the wirelength, but they may as well incorporate timing constraints or other complex constraints with very little modification.

They have several assets compared to partitioning-based placement. The use of continuous optimization and cost functions makes the result stable in the event of small netlist or initial placement modifications: for IC designers, they are thought to provide engineering change orders (ECO) with little disruption. From a software engineering point of view, they incorporate several distinct steps with a lot of specific parameters. This makes them extremely tunable and flexible for new cost functions, although the impact of a parameter change generally cannot be estimated directly.

A supposed downside is the need for fixed external pins. These anchors are said to be necessary to spread the cells during the first few iterations. However, this is indeed not the case for force-directed placers, including Coloquinte.

Examples of analytical placers include SimPL [20], mPl [10], ePlace [28].

Optimization algorithm

The core of an analytical placer is its optimization algorithm. This algorithm optimizes the cells' positions globally, subject to a convex cost function and generally unconstrained: the density constraints are taken into account through penalties.

The wirelength cost function is generally non-quadratic and non-differentiable. All analytical placers use a smooth approximation to this cost function. The main divide lies in the use - or not - of a quadratic cost function.

Quadratic placers A quadratic cost function has the advantage of being easy to optimize for. Solving $\arg\min xAx - 2bx$ amounts to solving $Ax = b$ with the symmetrical matrix A, which is extremely sparse in practice. This is equivalent to modelling the wires as springs pulling the cells together.

This cost function is not as restrictive as it seems. Since Gordian-L, such placers adjust the weights of the cost function to match the gradient of the true cost function [3].

This symmetric, sparse linear system can be solved efficiently with iterative methods such as the preconditioned conjugate gradient method. Therefore, quadratic placers are a very popular type of analytical placers.

Non-linear solvers An alternative is to optimize for the wirelength using nonlinear convex solvers. The most common method is the non-linear conjugate gradient method. However, it performs a line search, which requires several function evaluations at each iteration, which is why these solvers tended to be slower and more complex.

Some tools using non-linear optimization use multilevel optimization for performance reasons. That is, they coarsen the netlist for the first few placement phases. This is for example used by mPL [10].

Compared to the non-linear conjugate gradient method, the Nesterov method uses a constant step length instead of a line search and has good convergence properties. Recently, it gave extremely good results in ePlace in a flat non-linear placement tool [28].

Net models

The cost function needs to be accurate and enable efficient implementations. Although the exact net length, the length of its Steiner tree, is tempting, it is prohibitively expensive to compute and notoriously non-convex.

I don't know of any usage of the Steiner or spanning tree models in analytical placement so far: approximate models are generally used, namely the star, clique and half-perimeter models.

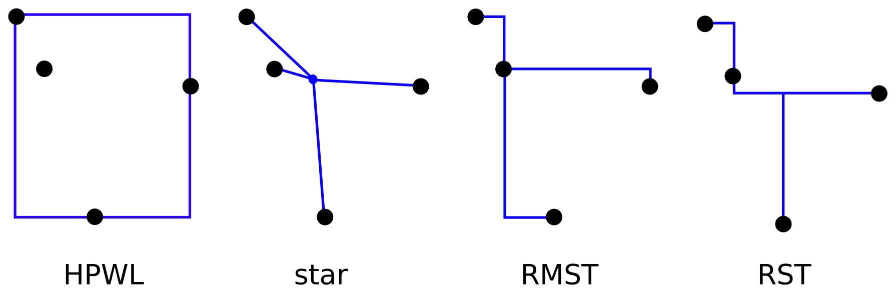


Figure 5.1: Various net models used in integrated circuit placement: the half-perimeter wirelength (HPWL), the star, the rectilinear minimum spanning tree (RMST) and the rectilinear Steiner tree (RST)

The half-perimeter model, or bound-to-bound model, is by far the most common: it is an optimistic evaluation of the net's length based on the size of its bounding box.

These cost functions contain absolute values and are therefore non-differentiable. In quadratic placement, the force is saturated beyond a certain threshold. For nonlinear placement, this cost function is smoothed using a log-sum-exp or weighted average approximation of the maximum.

Partitioning and force-directed methods

Analytical placement generally considers convex cost functions that do not include the density constraints: it tends to cluster the cells and produce a lot of overlaps.

There are two ways to get rid of this problem. The first one historically is to use the result of an analytical placement to guide a partitioning algorithm. More recently, force-directed placers appeared: in order to remove the overlaps, they add a penalty to the cost function.

Partitioning was used by the first placers Gordian [23] and Gordian-L [3], and is still in use in BonnTools [4]. However, most modern academic placers are force-directed.

Position-driven partitioning

The first global placement tools used a simple recursive bipartitioning based on a quadratic placement. This process is repeated, alternating quadratic placement and further partitioning, until sufficient precision is obtained. Generally, local optimizations are reapplied between adjacent regions.

In order to constrain the optimization process, sometimes nets spanning several regions are splitted and terminals are reported on the region's boundaries. As an alternative, center-of-gravity constraints are easy to integrate in the quadratic model, and another method is to contrain the center of gravity of each group of cells allocated to a common region.

BonnPlace introduced an alternative to bipartitioning, first with an optimal quadripartitioning algorithm [46], and later with a generalization to arbitrary multipartitioning [4]. They use an efficient algorithm for the transportation problem with few sources [6].

Force-directed placers

Most placers in the ISPD contests have been force-directed. They add a penalty to the cost function that drives the cells away from the dense regions. This penalty can be calculated directly from the density map, or can be seen as a force toward a closest overlap-free – legalized – placement.

The problem of calculating a penalty from the density map has been solved in various ways. APlace does it locally, using a bell-shaped potential function [16]. Other tools use a global potential: Kraftwerk [39] and ePlace [28] use an analogy with the electric potential to calculate the penalty force, while mPL [10] uses yet another scheme. Here, the problem

```
Data: $n$ movable cells, a cost function $WL$
Result: An optimized placement with positions $\mathbf{x}$
$\mathbf{x} \leftarrow \mathbf{0}$
do
    $penalty \leftarrow$ iteration- and placement-dependent penalty function
    $\mathbf{x} \leftarrow \arg\min(WL(\mathbf{p}) + \text{penalty}(\mathbf{p}))$
while Density constraints are not met;
return $\mathbf{x}_{LB}$
```

Algorithm 4: A generic force-directed placement algorithm

is the choice of a scheme that allows full-density regions, without creating artifacts such as halos. It may require complex force modulations [39].

When using a force toward a legal placement, the main requirement is to design a sufficiently fast approximate legalizer.

FastPlace [44] and RQL [45] chosed to use *cell shifting* methods: they do not perform a complete legalization and restrict themselves to small movements at each iteration. Moreover, FastPlace adds an *iterative local refinement* technique: it perform discrete moves between bins to improve both wirelength and density.

The legalization approach, on the other hand, originated in the SimPL family [20, 19, 21, 22], which coined the terms *lookahead legalization* and *rough legalization.* It is a form of primal-dual lagrangian optimization, where an optimistic result and a pessimistic projection on the feasible space are optimized together.

In the SimPL family, when legalizing an overfilled region, an enclosing bin with the appropriate density is found. This simplifies the work of the legalizer, which just needs to spread the cells uniformly in this bin. Since speed is paramount, it is done by a simple recursive partitioning algorithms, although SimPL takes the positions of fixed macros into account to chose its cutlines. In leaf regions, the coordinates of the cells are scaled independently. This approach has been used by other tools, most recently POLAR, which improved over SimPL's rough legalization [27].

Local optimization in analytical placers

A lot of analytical placers have included other passes to improve either the density or the wirelength. A typical example is the Iterative Local Refinement method in FastPlace [44]. Similar ideas are still in use in the most recent algorithms, like POLAR [27].

These algorithms divide the placement area into bins, and move or swap the cells to optimize the wirelength. Since the wirelength and the density can be measured exactly for such small changes, such algorithms are often used with an analytical placer.

5.1.3 Current state-of-the-art methods

As of 2015, both partitioning and force-directed placers are used. Most recent placers seem to use analytical placement, but both have given extremely good results on the public benchmarks from the ISPD contests.

Moreover, the separation is blurred: the tools tend to focus on one method but include the other as a hint or a post-processing. Partitioning placers sometimes use analytical methods, while analytical placers often make use of sophisticated discrete-optimization algorithms.

Although it is difficult to know the outcome on the industrial side, I expect analytical placers to be preferred. Partitioning tends to be purely top-down, and makes it relatively difficult to tune the cost function, include ECOs or make placement modifications through external tools. This remains a personal opinion based on perceived ease of implementation and maintenance: partitioning-based placers have been extended to handle complex cases such as routing-driven [35] and timing-driven [47] placement as well.

The needs are different in the FPGA world and in 3D circuits: when allocating a logical circuit to several FPGAs or stacked chips, good partitioning algorithms are necessary. Partitioning and analytical placers seem to be complimentary, the analytical placer being used when the placement region is continuous, while the partitioning-based placer takes discrete decisions.

5.2 An analytical placer with lookahead legalization

My choices for the Coloquinte placer reflect this conclusion. I designed a force-directed analytical placer with lookahead legalization. It seems the most promising approach and can be easily adapted to timing-driven placement, to arbitrary density constraints or to new industry challenges.

```
Data: $n$ movable cells, a cost function $WL$
Result: An optimized placement with positions $\mathbf{x}$
$\mathbf{x}_{LB} \leftarrow \mathbf{0}$
$\lambda \leftarrow \epsilon$
do
    $\mathbf{x}_{UB} \leftarrow$ legalization($\mathbf{x}_{LB}$)
    $\mathbf{x}_{LB} \leftarrow \arg\min(WL(\mathbf{x}) + \lambda \operatorname{dist}(\mathbf{x}, \mathbf{x}_{UB}))$
    update $\lambda$
while $WL(\mathbf{x}_{UB}) < \alpha\, WL(\mathbf{x}_{UB})$;
return $\mathbf{x}_{LB}$
```

Algorithm 5: Structure of a global placement algorithm based on lookahead legalization

I used a quadratic placer, which I considered faster before I heard of ePlace and the Nesterov method: this method should probably be preferred in the long run, and I plan to add it to Coloquinte. Coloquinte builds on SimPL's ideas of using a roughly legalized placement as an anchor to pull the placement - the lookahead legalization step. Two placements are maintained: an optimistic one which doesn't respect the density constraints, and a pessimistic one which respects the constraints.

I kept a purely analytical design, and didn't experiment with refinement methods. My focus has been on improving the lookahead legalization step without slowing it, and trying new net models to better reflect the real wirelength. Compared to SimPL and its successors, Coloquinte greatly improves the quality of this lookahead legalization, which impacts the quality of the exact legalization step as well.

Since it is a placement library which ought to evolve with the needs of Coriolis, Coloquinte provides a collection of algorithms, net models and parameters.

5.3 Net models and optimisation

Coloquinte provides several quadratic net models. Simple net models to implement are bound-to-bound HPWL, star and clique. Other models such as minimum spanning tree and Steiner tree heuristics are implemented but more time-consuming. Those models can be mixed easily (for example, HPWL for small nets and Steiner or MST for high pin counts). The two most efficient models in my experience have been bound-to-bound and Steiner. Other models are obviously possible but not implemented yet: fully power- and delay-aware Steiner routing could be interesting, particularly for detailed placement. By default, Coloquinte uses the bound-to-bound net model, which has proven to be the most efficient in practice. The saturation threshold for the quadratic model is one standard cell height.

In the long run, Coloquinte will probably implement generic nonlinear optimization using Nesterov method, mostly for faster optimization. I didn't know of Nesterov's method when I began writing Coloquinte, which is why it was initially designed as a quadratic placer. However, it is probably the way to go for better solution quality.

5.4 Influence of the penalty function

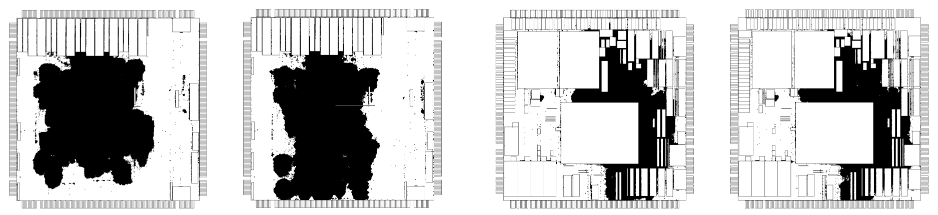
In force-directed placers, in particular in placers that use lookahead legalization or cell shifting/ILR, an important design choice is penalty function applied to the cells. Equivalently, I will talk about the *pulling force* applied to it. Typically, either a constant force or an elastic force is applied, with some dependency in the cell's area. In order to ensure convergence, this force becomes stronger at each iteration.

This leaves a large span of design choices: how the penalty on an individual cell is calculated, and how the penalty factor λ changes at each iteration. These choices are purely empirical, and for me it was mostly a trial-and-error process. A very simple scheme uses constant or elastic pulling forces to the legalized position, proportional to the module's area, with a constant increment at each iteration. But better – and more complex – methods are possible, both in the choice of the force on a given cell – which I call *force modulation* – and in the choice of the increment of the penalty factor – *force scheduling.*

I tested some of them; in my experience, there is no winning scheme, and each choice involves tradeoffs. Benchmarks with macros in the placement area tend to be extremely sensitive to minor changes in the parameters.

5.4.1 Force modulation

RQL, for example, targets better pulling forces for individual cells: it uses a complex calculation to obtain the fixed point of the force, and nullifys the strongest forces [45].



(a) Bigblue1 (WL + 0.8%) (b) Adaptec2 (WL - 2.1%)

Figure 5.2: The effect of individual forces modulation on two benchmarks; effect of using an elastic force (right) versus a constant force (left)

I didn't experiment with such a complex scheme in Coloquinte, but I had to chose how the force would vary with the distance relative to the roughly legalized placement. The default is currently to use a linear force – a spring – rather than a constant one. Depending on the benchmark, it is or is not an improvement.

5.4.2 Force scheduling

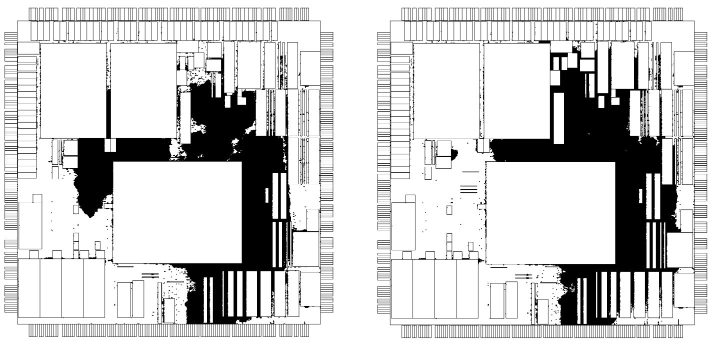
On the other hand, ComPLx tries to improve the schedule of the iteration-dependent penalty factor *lambda* by making the increment $\Delta\lambda$ change at each iteration. It computes the pulling force increment from various measures on the current placement and their evolution – in ComPlX, from the distance between the two placements [21]; in Maple's local refinement, from the density [22]. ComPLX's update scheme is purely empirical and does not focus on a particular goal. Maple's scheme, on the other hand, tries to conciliate density and wirelength changes but is restricted to local refinement. I used a fully adaptative scheme for analytical placement, with mixed results.

On my test placement instances, I remarked that most of the progress was made during a few iterations. During these iterations, the lower-bound on the wirelength increases faster, and there is proportionally few feedback from the lookahead legalizer. Therefore, I tried to keep the rate of improvement almost constant at each iteration, with the hope of both improving the placement results and improving the runtime – with a larger average force increment.

Force scheduling in Coriolis Etesian uses a form of prediction to keep the variation at each iteration stable for some metric Φ. I mostly used wirelength-based metrics rate, although various measures of the disruption between upper and lower bound should be tested as well. In the current version of Etesian, I use the ratio between upper and lower bound, $\Phi = \frac{WL_{LB}}{WL_{UB}}$.

The increment at step k uses a prediction on the variation of Φ to compute $\Delta\lambda = \Delta\Phi_{desired} / \left(\frac{d\Phi}{d\lambda}\right)_{predicted}$. Advanced predictions methods are not required, and are even counter-productive. The zero-order prediction such as $\left(\frac{d\Phi}{d\lambda}\right)_{predicted} = \frac{\Delta\Phi_{k-1}}{\Delta\lambda_{k-1}}$ is sufficient, while the first-order prediction $2\frac{\Delta\Phi_{k-1}}{\Delta\lambda_{k-1}} - \frac{\Delta\Phi_{k-2}}{\Delta\lambda_{k-2}}$tends to be instable. I tried various linear combinations of the previous iterations' $\frac{\Delta\Phi}{\Delta\lambda}$, as well as other predictive update schemes, but kept the zero-order prediction as the default.

In Etesian, $\Delta\lambda$ is constrained to a range of positive values in order to ensure theoretical convergence – it is useful during the first iterations as well, where the lower-bound wirelength improves and $\Delta\Phi$ is generally negative. $\Delta\Phi_{desired}$, $\Delta\lambda_{min}$ and $\Delta\lambda_{max}$ are derived from the target quality/runtime tradeoffs i.e. the user's configuration.



(a) Without force scheduling (b) With force scheduling

Figure 5.3: Adaptative force scheduling may make placement results worse: on Adaptec2, the result is 2.8% better *without* force scheduling

The results, however, are not as expected. Although it generally helps, force scheduling adds other design choices: it can change placement results in both ways and needs to be benchmarked on many more circuits with various target metrics. Compared to ComPLx's scheduling method, the choice is explicit: force scheduling is restricted to the choice of the metric Φ to improve at a constant pace rather than of an arbitrary update scheme.

5.5 Lookahead legalization

Lookahead legalization – or rough legalization – in analytical placers aims at respecting the density constraints, contrary to a true legalization which removes all overlaps. The biggest difference with previous placers is the use of new algorithms for lookahead legalization. In Coloquinte, lookahead legalization is based on bi- or multi-partitioning: the placement region is subdivided recursively and each cell is allocated to its preferred region based on positional criteria. In such an algorithm, very similar to the ones used by SimPL, early partitioning decisions may have a big effect on the final outcome.

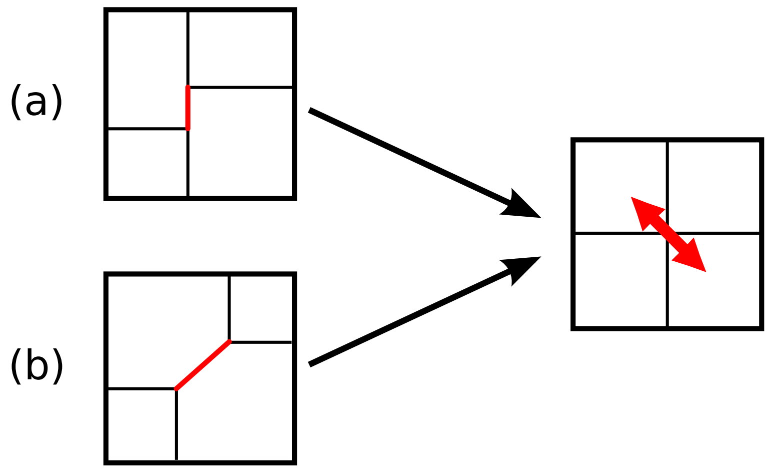


Figure 5.4: An illustration of the disruptions involved with recursive bipartitioning and multipartitioning: a large frontier is mapped to a single point, resulting in higher wirelength disruption. Further refinement is necessary to obtain better results

5.5.1 Local optimizations in rough legalizers

In Coloquinte, lookahead legalization is not done in a purely empirical manner but targets an optimization objective instead. Considering a cost function during lookahead legalization makes it flexible: it can minimize cell displacement or quadratic displacement, based on Manhattan or Euclidean distance...

The rationale is that local search and specialized algorithms can improve this legalization's result at a very low cost. A naive method is to perform reallocation of the cells between two nearby regions. This can be done with a simple sorting or more sophisticated quick-median-like algorithms. When considering regions adjacent at their corners for improvement, this method already yields good results.

5.5.2 Formulation as a transportation problem

A lookahead legalization algorithm needs to allocate cells to regions with minimal displacement costs subject to capacity constraint. It includes the bin-packing problem, which is NP-complete, but its relaxation is good enough in practice, when a lot of small cells are involved: it is the transportation problem. The transportation problem is a special type of network flow problem where the goal is to allocate the flow of n sources to k sinks subject to capacity constraints and source-to-sink costs.

In fact, minimum-cost-flow formulations are used for legalization [7], partitioning [4] and even for the first lookahead legalization passes of a force-directed placer [2]. However, the fastest known algorithms for this problem are based on a minimum-cost flow algorithms which have complexity $nk(n+k)\log^2(n+k)$ [30, 6]. Since lookahead legalization needs high values of both n and k, this algorithmic complexity makes an exact solution impractical, which explains why transportation problems have never been considered for lookahead legalization: all lookahead legalization algorithms are based on empirical heuristics such as cutline shifting.

My approach, on the other hand, considers the transportation problem and solves it approximately. Considering the transportation cost should already yield consequent improvement: heuristics that do not consider the transportation problem have a high risk of introducing disruption and cannot take arbitrary transportation costs into account. Moreover, the algorithm can minimize the cost subject to various distance functions, which makes it possible to truly evaluate the quality of the results.

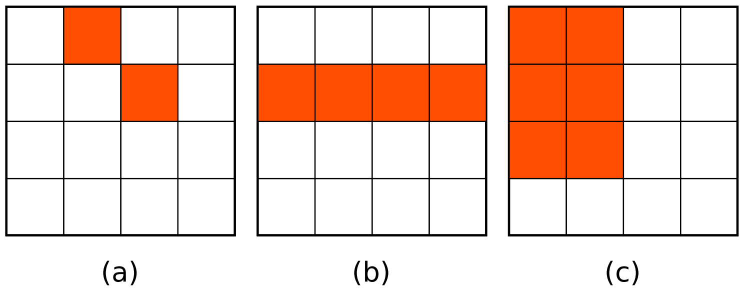


Figure 5.5: Three ways to reoptimize neighbouring bins: bipartitioning (a), complete row or column (b), and multipartitioning (c)

The placement region is partitioned until it is sufficiently fine-grained; after each partitioning step, nearby regions are reoptimized together in order to get a better solution. It is similar to partitioning quadratic placers such as BonnPlace. The simplest way to do it is to reallocate the cells between only two regions. Another pass based on the one-dimensional transportation algorithm proved to be useful to optimize complete rows or columns. I experimented with slower, more precise methods derived from BonnPlace's algorithm as well.

Two-region optimization

When optimizing cell allocation between only two regions, efficient algorithms are possible. The simplest solution is to sort the cells by their marginal cost first, and split them according to the regions' capacities. This yields an $n \log n$ algorithm, which is not optimal.

This is similar to the problem of calculating the median of a sequence. Using the quickmedian algorithm or a worst-case optimal version, this problem takes linear time. However, I didn't experience any runtime benefit compared to the sort-based version in C++: the current version of Coloquinte uses the latter.

1D transportation

I try to take advantage of the low dimensionality of this problem; although 2D transportations problems are not solved efficiently, balanced 1D transportation is easy. To my knowledge, unbalanced 1D transportation had not been solved and I devised a linearithmic algorithms to solve it optimally: this step avoids the early biases that may occur during bipartitioning.

This algorithm is used during lookahead legalization to perform *global* bipartitioning: the cells are reallocated optimally in a row - or column - of regions.

It does not extend to non-Manhattan distances, however, and cannot be applied diagonally: in this case, the two-region algorithm is used instead.

Multipartitioning

BonnPlace uses transportation problems for VLSI partitioning: they developped an efficient algorithm for small k [6]. I implemented and tested it in Coloquinte, but it is still to slow for systematic use, in particular for lookahead legalization. Although it is not as fast as the simple two-region algorithm and impractical for lookahead legalization but is ultimately the most accurate solution.

Compared to the original algorithm, I use a sequential-shortest-path method to solve the (small) network flow problem: in the applications I considered, the average number of shortest-path runs per source was extremely small (less than 20, and generally less than 2).

5.5.3 Handling leaf regions

Once the regions are small enough - below four standard cell heights in Coriolis - the cells in each region are spread. This is done independently on each coordinate, much like previous heuristics. However, whereas the legalized region in SimPL always has full density, this is not the case in Coloquinte where the whole placement region is considered.

In SimPL, it is possible to just sort the cells and sequentially assign them to a position. In Coloquinte, it would result in unnecessarily spreading some cells inside a region despite the density constraints being met. To handle non-dense regions, Coloquinte solves an ordered scheduling problem with the cascading descent algorithm: in a region of height h, the cells of areas a_i are assigned proportional widths $\frac{a_i}{h}$ and assigned as close as possible to their target position.

Chapter 6

Legalization

Legalization is the process of transforming a global placement result into a legal, overlap-free placement to be passed to the detailed placer. For standard cells, the problem is to allocate the cells to predefined rows. Various legalization algorithms have been proposed. The first ones have been simple greedy ones. More recent algorithms often model legalization as a minimum-cost flow problem from overfilled regions to whitespace.

In Coloquinte, the exact legalization phase is run after a lookahead legalization, on an already almost-legal placement. Coloquinte's legalization algorithm is quite simple and obtains good results - but like all legalization algorithms it may fail on complex instances, when a large number of small macros are to be placed.

6.1 Prior art

Greedy algorithms for legalization are very simple to implement and surprisingly efficient [43]. Most placers use the naive algorithm, that puts the cells from left to right at the closest free position. Surprisingly, it is subject to a software patent [14]. Although they are generally good enough, one of the problems of such greedy algorithms is that they are oblivious of the whitespace they waste once the cells are placed. It is a problem with clustered global placements where the first cell placed constrains all the others.

Newer legalizers do not place cells sequentially. One approach to improve the greedy algorithm is to push the cells toward the left to make use the available whitespace. This approach has once again been subject to a software patent [29], but easily hits quadratic runtime at high density. Abacus [40] circumvents this problem by using a quadratic cost function, which makes for a lower worst-case complexity (linear) when pushing a cell.

Other legalizers use bin-based approaches to place groups of cells simultaneously [25].

6.2 A greedy legalizer based on cascading descent

Coloquinte places the cells greedily like Tetris and Abacus, and like Abacus is able to push the cells that are already placed. However, it does it by iteratively solving the ordered single row problem. Just like other greedy algorithms, it performs a preliminary sorting - for example on the x coordinate. When appending a cell to a row, all other cells in the row are moved toward the optimum position according to a Manhattan or quadratic penalty function, with amortized logarithmic complexity.

The practical complexity is linearithmic, but worse cases or failures are theoretically possible. Since the legalization problem includes the NP-complete bin-packing problem, no worst-case-efficient algorithm is known. In practice, the runtime is below a second for hundreds of thousands of cells if a rough legalization was run previously.

This algorithm is perfect for standard cell placement, where it should use less whitespace, but is not suitable for mixed-size legalization as is. Mixed-size legalization may use multiple rows for a single cell: the constraints form an acyclic graph. It makes the act of pushing other cells inherently complex, since it can possibly break already formed clusters. Therefore, to simplify the algorithm, small macros are legalized once and for all, and can't be pushed later. Although it penalizes libraries with multiple cell heights, the result for those libraries will remain better than other legalization algorithms, which do not handle them at all and would need to fall back to a Tetris-like algorithm anyway.

6.3 Two-pass legalization

A rough legalization algorithm is already used at each iteration of the global placer. In Coloquinte, exact legalization is always performed after an identical rough legalization pass. At the beginning, this step was meant as a hint for exact legalization. I envisioned various ways it could guide the final legalization: by modifying the order in which the cells are treated, or mixed with the original objective in various ways.

I tried several schemes to exploit this information, but the best results have been obtained when the original objective is entirely discarded and the output of the rough legalizer is used directly. Mixing it with the original objective or using it as a hint always led to worse results in my experience.

My interpretation is that the global optimization from the rough legalizer has a tremendous effect. Moreover, its output is inherently optimistic: the displacement is generally less than is necessary because of the way it handles the leaf regions. Therefore, the exact legalizer receives a good global hint that is still biased toward the original objective.

Chapter 7

Detailed placement

Detailed placement is the final improvement of a circuit that is supposed to be good at the global scale. It moves and swaps the cells locally. The algorithms here have been mostly pure heuristics, although mixed-integer programming has been proposed as well.

The detailed placement in Coloquinte is based on several passes. These passes typically use local search associated with an exact algorithm for some subproblem. The exact algorithm amounts to searching a much larger neighbourhood without the exponential complexity of the naive method: typically, only the topological order of the cells is subject to brute-force local search, while positions and orientations are optimized exactly under this constraint.

Some basic passes achieve a large improvement with low complexity. Just like the global placement step, the quality/runtime tradeoff can be adjusted easily by modifying the passes called and their parameter.

Almost all algorithms I describe here have been implemented both for half-perimeter and Steiner wirelength optimization. In practice, Steiner optimization turns out to be slow and the algorithms called by Coriolis are almost all half-perimeter-driven.

7.1 Global swaps

Our main detailed placement passes moves standard cells between different rows. Other common moves, like intra-row cell reordering and cell shifting are left to row ironing passes, which can take advantage of more advanced algorithms.

This pass is similar to the vertical swap pass from [31]: it only looks for local moves, typically less than 3 rows. In Coloquinte, this pass may evaluate the true Steiner wirelength rather than the nets' perimeters and look for better orientations of the cells.

7.2 Fixed-topology optimizations

In Coloquinte, some optimization passes change the positions and orientations without modifying the order of the cells. They try to find the best possible alignment of the pins between different rows without resorting to local search. There are two ways to perform this optimization: globally by formulating the problem as the dual of a min-cost flow problem, or independently for each row.

7.2.1 Dual minimum-cost-flow

The min-cost flow formulation was originally proposed for floorplanning [42], and never applied to large scale designs. I implemented it using Lemon's network simplex library, which is generally considered to be the fastest implementation [24]. This pass turns out to be impractical for big designs, where the algorithm may take hours to complete.

Other minimum cost flow algorithms may take advantage of the particular structure of the problem, but I didn't try an interior point or a successive shortest path method. Cost and capacity scaling were extremely slow, and only the network simplex algorithm completed the optimization on big designs.

7.2.2 Ordered single row problem

When restricted to a single row, the fixed-topology placement problem obviously becomes an ordered row problem. It can be solved in linearithmic time: it is an inexpensive and efficient pass to finalize global placement, which replaces single-segment clustering in BonnTools and FastPlace. In Coloquinte, it is available for half-perimeter wirelength optimization and to optimize for the current Steiner tree topology.

It is possible to integrate cell orientations in the objective function, which becomes non-convex. The non-convex ordered single row problem being solvable in quadratic time, it can get the optimal positions and orientations for the current ordering of the row. In practice, it is generally slower and turns out to be less efficient than several simpler passes, but can be useful when extremely high quality is sought.

All possibilities have not been tried yet: this feature makes it possible to optimize for other cost functions, such as non-convex approximations of the Steiner wirelength, or for non-convex domains when a macro takes part of the row.

7.3 Row ironing

Row ironing, in the Capo terminology, is the pass of locally optimal intra-row optimizations. Since there may be significant free space in modern integrated circuits, Coloquinte does not assume a dense placement and takes free space into account with no added complexity.

Row ironing tries to reorder a sequence of cells in the row to improve the wirelength. I saw no benefit in trying to widen the search domain with a branch-and-bound approach like Capo's and chosed a brute-force approach more similar to Fastplace's, where every permutation is tested.

There are several row ironing passes in Coloquinte, but they all share this same structure: every permutation is tested for a few cells and the best one is kept. Only the underlying algorithms to get the positions and orientations of these cells differ, and the size of the optimization window - generally 3 to 5 cells.

An ordered single-row problem is solved for each possible ordering, which makes it suitable to non-dense placements. Although it is not done in the current software, it can integrate density constraints by restricting the range of individual cells. Moreover, the non-convex version allows us to optimize for orientation with a $O(n^2 n!)$ complexity instead of $O(2^n n \log n n!)$.

Both methods may use the usual half-perimeter wirelength, but they can optimize for a given Steiner topology as well, yielding inexpensive Steiner wirelength minimization.

7.4 Cell flipping

Cell flipping is a useful step during placement. It is the process of finding better orientations for the cells through mirroring. With a HPWL objective, vertical and horizontal mirroring are independent, and a single algorithm is called twice.

Good cell flipping has been shown to improve the placement results on HPWL-driven benchmarks by more than 1% [37]. Even if the row ironing and fixed-topology optimization passes can perform greedy and whitespace-aware cell flipping, I integrate the greedy cell flipping pass as a standalone module to be called during global placement and at the designer's convenience. I do not exclude to implement some more complex algorithms based on linear programming like [37] in subsequent iterations.

The current cell flipping algorithms flips the cells while there is a local improvement to be gained. It uses a stack to track possibly suboptimal cells, and pushes new cells only when they become the extreme terminal of a net. For the half-perimeter wirelength, this algorithm achieves local optimality. However, the gain compared to a single-pass greedy algorithm like [37] is negligible.

```
Data: A placement
Result: A new placements with cells optionally mirrored vertically, locally optimal
        for the half-perimeter wirelength
stack ← emptyStack()
foreach Cell c do
    push(stack, c)
end
while not empty(stack) do
    c ← pop(stack)
    if the wirelength is lower when flipping c then
        flip c
        foreach Net connected to c do
            if c is replaced as furthest left pin then push new extreme cell;
            if c is replaced as furthest right pin then push new extreme cell;
        end
    end
end
```

Algorithm 6: The greedy cell flipping algorithm

7.5 Going further: integer linear programming and 2D window optimization

A promising field for detailed placement optimizations is mixed integer programming, in particular integer linear programming (ILP). Recently, several successful models have been proposed to optimize the placement of large windows of standard cells [34, 9, 26] and for floorplanning instances [41], with different methods and tradeoffs.

I implemented both standard cells and floorplanning ILP models, and tested various models and additional constraints to speed-up the solution process. None of them is in use in Coriolis yet.

7.5.1 ILP for standard cell placement

ILP could be particularly useful for detailed placement. First, it guarantees optimality for the optimized window, or at least provides a lower bound. Second, it can consider moves that are not part of the usual local search methods, which only consider simple swaps and reorderings within a single row. For standard cell designs, where the cells don't have a uniform size and swapping possibilities are rare, this limits detailed placement efficiency [33].

All ILP methods targeting detailed placement in the literature are site-based i.e. they tile the area with uniform rectangles and use boolean variables to represent whether each cell occupies each rectangle. I won't detail these models here.

In order to be usable on large-scale digital placement instance, which contain tens of thousands of cells, the detailed placer should yield the optimal solution for a given window in less than a second. Using the GLPK and Gurobi solvers on handcrafted problem instances, I didn't manage to obtain fast enough solutions beyond 8 cell windows, contrary to previous publications.

There may be several reasons for this failure. Most likely, the lack of a branch-and-price approach makes the solution process much slower. The particular structure of my handcrafted problems may be at fault too, or the lack of a good initial solution. If I am not at fault due to another mistake, fixing any of these problems requires a direct interface between Coloquinte and the GLPK solver, which I didn't implement yet. Testing ILP-based detailed placement capabilites is one of the next steps for Coloquinte. However, this still requires better knowledge of branch-and-price ILP methods.

Another interesting option would be to implement branch-and-bound algorithms based on the dual min-cost-flow problem. Since the boolean variables would not be included in this linear relaxation, it would be less tight. However, solving it would be much easier: the boolean variables account for most of the variables in the models, and the minimum-cost flow problem allows for much faster solutions than generic linear programming approaches.

7.5.2 Floorplanning

ILP models seem promising in floorplanning as well: compared to other algorithms, it is possible to include complex constraints with minimal efforts. For example, symmetry constraints, soft (deformable) macro blocks, ad-hoc block spacing, wirelength and area all translate easily into an ILP model. The blocks to be placed and the constraints are much more complex than in detailed placement, and the models used are different – models for floorplanning were used much earlier: no research was done toward site-based models in floorplanning either. The original model from [41] uses boolean variables to represent whether two blocks are one above the other or one on the right of the other – i.e. 4 possible combinations for each pair of blocks.

Floorplanning problems typically have few variables. Since other approaches are generally bad (targeting only limited objectives such as area), the floorplanner can tolerate non-optimal solutions. However, as the number of modules grows, the problem quickly becomes untractable and a more sophisticated approach is unavoidable.

I used models derived from [41] for analog placement with soft modules. Although this method alone is not scalable, it has been a good introduction to ILP models.

Chapter 8

Benchmark results

Coloquinte has been benchmarked on the ISPD05 placement problems and on circuits built in the Lip6 laboratory.

8.1 ISPD05 benchmarks

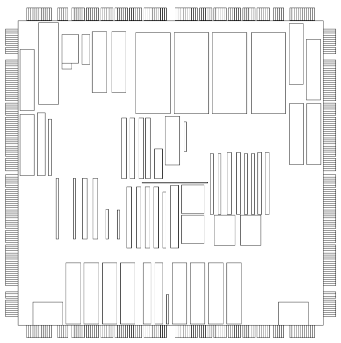
The ISPD05 benchmark set was released for the contest of the ISPD conference is 2005, and has been used as the main evaluation for placers ever since.

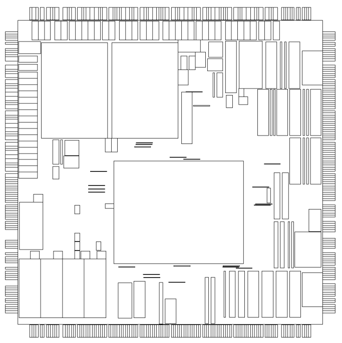
These results present the default version of Etesian, the best results published for ePlace, and the best results for other quadratic placers according to [28]. It is probably not an adequate comparison, since I don't take the runtime into account, and they seem to be slow compared to published results. Still, it shows that Coloquinte's approach is valid for a high-quality placement tool.

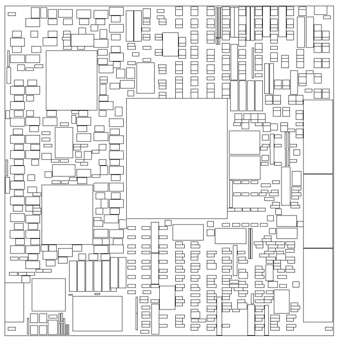
Problem name	Best published result	Best quadratic placer	Coloquinte result
Adaptec1	**74.63**	76.36 (Maple)	76.49
Adaptec2	**84.84**	86.16 (POLAR)	87.90
Adaptec3	**194.57**	201.30 (POLAR)	202.12
Adaptec4	**179.02**	179.91 (Maple)	182.27
Bigblue1	**90.99**	93.74 (Maple)	94.76
Bigblue2	141.83	143.95 (POLAR)	**141.16**
Bigblue3	**308.77**	317.17 (Bonn)	#
Bigblue4	**753.20**	775.71 (Maple)	#

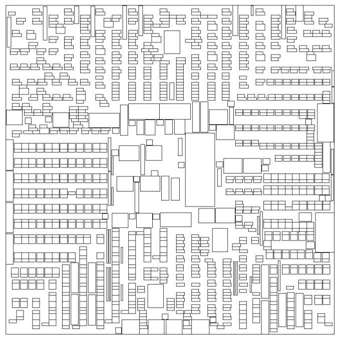
Table 8.1: HPWL results on the ISPD05 benchmark, compared to ePlace and other quadratic placers [28]

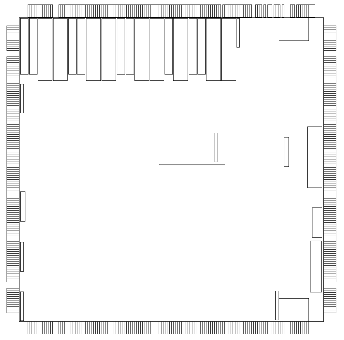
The results are not available for the Bigblue3 and Bigblue4 benchmarks: Bigblue3 requires support for movable macros, which I haven't implemented yet, while Bigblue4 is


(a) Adaptec1


(b) Adaptec2


(c) Adaptec3


(d) Adaptec4


(e) Bigblue1

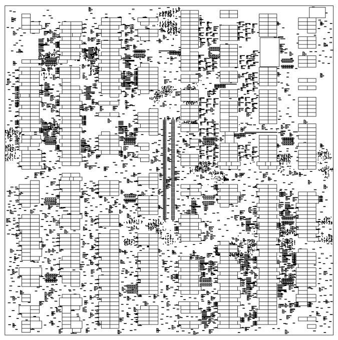
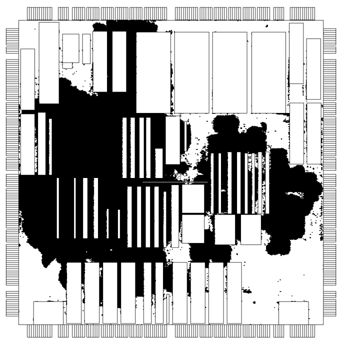
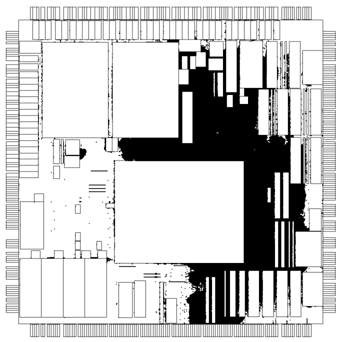

(f) Bigblue2

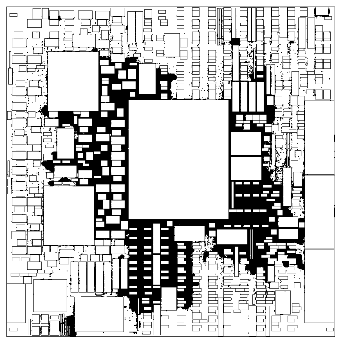
Figure 8.1: The fixed blocks and cells in some ISPD05 circuits

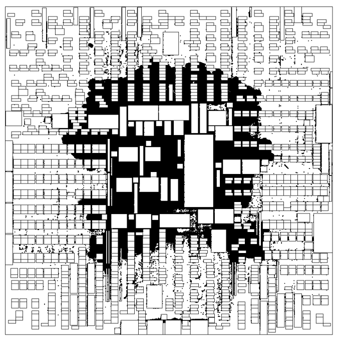
too big and causes memory overflows in the Hurricane database.

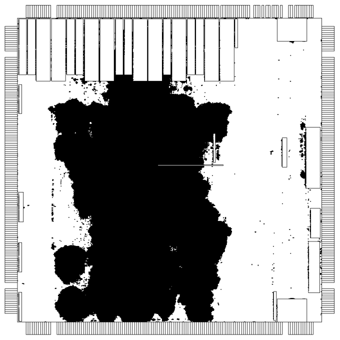
These results are shown for the current, default algorithm. As seen earlier, each parameter influences final placement quality by several percents, often in different ways depending on the circuit. Since slight modifications may have a tremendous effect on the final result, more parameter tuning is necessary in Coloquinte. I think that automatized parameter search, or even parameter learning depending on the circuit's features, is almost unavoidable to largely improve on current placement results if no new class of placement algorithm is introduced.


(a) Adaptec1


(b) Adaptec2


(c) Adaptec3


(d) Adaptec4


(e) Bigblue1

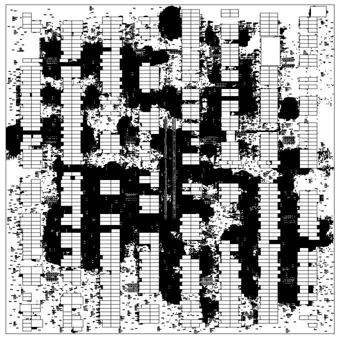

(f) Bigblue2

Figure 8.2: Coloquinte's placement for some ISPD05 circuits

8.2 FPGA placement

Placing a regular structure is a good way to see at first glance whether a placer yields a good result [33]. I only used real circuit as benchmarks, however, but it is still possible to see how the global placer recovers the circuit's structure.

FPGAs are highly regular structures, while still being fairly typical circuits. I synthetized one of the FPGAs designed in the laboratory using Coriolis, and Jean-Paul Chaput built the same netlist using an industrial tool. The results are quite encouraging, both for Coriolis and regarding the state of modern industrial tools.

	Coriolis	Industrial tool
HPWL	**25495328**	28885624
Steiner WL	40559968	#
Routed WL	47713376	**40045672**

Table 8.2: The industrial tool optimizes for the true wirelength objective

Coriolis yields a better HPWL – it is still the default optimization objective – but the industrial tool yields much better routed wirelength. This makes me believe that there is some kind of Steiner-driven optimization in modern tools. Moreover, Coloquinte is once again much slower. Today's industrial tools seem extremely competitive: Coloquinte still needs a lot of performance optimization and tuning to achieve this level of placement quality.

Although it is a huge improvement over the older Mauka placer in Coriolis, it seems to me that the placer still is a bottleneck, due to Coloquinte/Etesian being the slowest tool. From a user's point of view, however, file format compatibility, interfaces and an even better routing are more important: the placement algorithms are "good enough" and they can largely forget about them.

Chapter 9

Conclusion

Currently, Coriolis constitutes the only open state-of-the-art toolchain for placement and routing. Coloquinte is an efficient placer that integrates well with the other tools in the Coriolis toolchain. Designing and writing Coloquinte has been the major contribution of this part-time project. Some additional work, not described in this paper, targeted new routing and logical synthesis methods.

In Coloquinte, I introduced improvements for lookahead legalization, legalization and detailed placement. Coloquinte as a whole shows once again that purely force-directed placers are viable, but does not introduce better results either. Nonetheless, it is a full-featured state-of-the-art placer and is fit for Coriolis, and brings a lot of small improvements to the placement flow.

I will likely continue working on Coriolis on my spare-time. There is still a lot of work to do on placement algorithms, but Coriolis is bigger than Coloquinte, and it is entirely possible that I work on other parts of the toolchain or other research subjects as well.

Bibliography

[1] Saurabh N. Adya, Igor L. Markov, and Paul G. Villarrubia. Improving min-cut placement for vlsi using analytical techniques. In *Proc. IBM ACAS Conference, IBM ARL*, pages 55–62, 2003.

[2] A.R. Agnihotri and P.H. Madden. Fast analytic placement using minimum cost flow. In *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 128–134, Jan 2007.

[3] C.J. Alpert, T.F. Chan, A.B. Kahng, I.L. Markov, and P. Mulet. Faster minimization of linear wirelength for global placement. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(1):3–13, Jan 1998.

[4] U. Brenner, M. Struzyna, and J. Vygen. Bonnplace: Placement of leading-edge chips by advanced combinatorial algorithms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(9):1607–1620, Sept 2008.

[5] U. Brenner and J. Vygen. Faster optimal single-row placement with fixed ordering. In *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pages 117–121, 2000.

[6] Ulrich Brenner. A faster polynomial algorithm for the unbalanced hitchcock transportation problem. *Oper. Res. Lett.*, 36(4):408–413, July 2008.

[7] Ulrich Brenner, Anna Pauli, and Jens Vygen. Almost optimum placement legalization by minimum cost flow and dynamic programming. In *Proceedings of the 2004 International Symposium on Physical Design*, ISPD '04, pages 2–9, New York, NY, USA, 2004. ACM.

[8] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. Design and implementation of move-based heuristics for vlsi hypergraph partitioning. *J. Exp. Algorithmics*, 5, December 2000.

[9] Stephen Cauley, Venkataramanan Balakrishnan, Y. Charlie Hu, and Cheng-Kok Koh. A parallel branch-and-cut approach for detailed placement. *ACM Trans. Des. Autom. Electron. Syst.*, 16(2):18:1–18:19, April 2011.

[10] Tony Chan, Jason Cong, and Kenton Sze. Multilevel generalized force-directed method for circuit placement. In *Proceedings of the 2005 International Symposium on Physical Design*, ISPD '05, pages 185–192, New York, NY, USA, 2005. ACM.

[11] P. Chin and A. Vannelli. Interior point methods for placement. In *Circuits and Systems, 1994. ISCAS '94., 1994 IEEE International Symposium on*, volume 1, pages 169–172 vol.1, May 1994.

[12] C. Chu and Yiu-Chung Wong. Flute: Fast lookup table based rectilinear steiner minimal tree algorithm for vlsi design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(1):70–83, Jan 2008.

[13] Leonidas J. Guibas and Jorge Stolfi. On computing all north-east nearest neighbors in the l1 metric. *Inf. Process. Lett.*, 17(4):219–223, 1983.

[14] D. Hill. Method and system for high speed detailed placement of cells within an integrated circuit design, April 9 2002. US Patent 6,370,673.

[15] YUAN Jinjiang. The NP-hardness of the single machine common due date weighted tardiness problem. *Journal of Systems Science and Complexity*, 5(4):328, 1992.

[16] A.B. Kahng, S. Reda, and Qinke Wang. Architecture and details of a high quality, large-scale analytical placer. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pages 891–898, Nov 2005.

[17] Andrew B. Kahng, Paul Tucker, and Alexander Zelikovsky. Optimization of linear placements for wirelength minimization with free sites. In *ASP-DAC*, pages 241–244. IEEE, 1999.

[18] G. Karypis, R. Aggarwal, V. Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):69–79, March 1999.

[19] Myung-Chul Kim, Jin Hu, Dong-Jin Lee, and Igor L Markov. A simplr method for routability-driven placement. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 67–73. IEEE, 2011.

[20] Myung-Chul Kim, Dong-Jin Lee, and Igor L Markov. Simpl: An effective placement algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(1):50–60, 2012.

[21] Myung-Chul Kim and Igor L Markov. Complx: A competitive primal-dual lagrange optimization for global placement. In *Proceedings of the 49th Annual Design Automation Conference*, pages 747–752. ACM, 2012.

[22] Myung-Chul Kim, Natarajan Viswanathan, Charles J Alpert, Igor L Markov, and Shyam Ramji. Maple: multilevel adaptive placement for mixed-size designs. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, pages 193–200. ACM, 2012.

[23] J.M. Kleinhans, G. Sigl, F.M. Johannes, and K.J. Antreich. Gordian: Vlsi placement by quadratic programming and slicing optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(3):356–365, Mar 1991.

[24] Péter Kovács. Minimum-cost flow algorithms: an experimental evaluation. *Optimization Methods and Software*, 30(1):94–127, 2015.

[25] Yu-Min Lee, Tsung-You Wu, and Po-Yi Chiang. A hierarchical bin-based legalizer for standard-cell designs with minimal disturbance. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 568–573, Piscataway, NJ, USA, 2010. IEEE Press.

[26] Shuai Li and Cheng-kok Koh. Mip-based detailed placer for mixed-size circuits. In *Proceedings of the 2014 on International Symposium on Physical Design*, ISPD '14, pages 11–18, New York, NY, USA, 2014. ACM.

[27] Tao Lin, C. Chu, J.R. Shinnerl, I. Bustany, and I. Nedelchev. Polar: Placement based on novel rough legalization and refinement. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 357–362, Nov 2013.

[28] Jingwei Lu, Hao Zhuang, Pengwen Chen, Hongliang Chang, Chin-Chih Chang, Yiu-Chung Wong, Lu Sha, D. Huang, Yufeng Luo, Chin-Chi Teng, and Chung-Kuan Cheng. eplace-ms: Electrostatics-based placement for mixed-size circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 34(5):685–698, May 2015.

[29] R. Miller, W.C. Naylor, and Y.C. Wong. Detailed placer for optimizing high density cell placement in a linear runtime, November 29 2011. US Patent 8,069,429.

[30] James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *OPERATIONS RESEARCH*, pages 377–387, 1988.

[31] Min Pan, N. Viswanathan, and C. Chu. An efficient and effective detailed placement algorithm. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pages 48–55, Nov 2005.

[32] Yunpeng Pan and L. Shi. Dual constrained single machine sequencing to minimize total weighted completion time. *Automation Science and Engineering, IEEE Transactions on*, 2(4):344–357, Oct 2005.

[33] David A. Papa and et al. Constructive benchmarking for placement, 2004.

[34] P. Ramachandaran, A.R. Agnihotri, S. Ono, P. Damodaran, K. Srihari, and P.H. Madden. Optimal placement by branch-and-price. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 1, pages 337–342 Vol. 1, Jan 2005.

[35] J.A. Roy and I.L. Markov. Seeing the forest and the trees: Steiner wirelength optimization in placement. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(4):632–644, April 2007.

[36] C. Sechen and A. Sangiovanni-Vincentelli. The timberwolf placement and routing package. *Solid-State Circuits, IEEE Journal of*, 20(2):510–522, April 1985.

[37] Chiu-Wing Sham, E.F.Y. Young, and C. Chu. Optimal cell flipping in placement and floorplanning. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 1109–1114, 2006.

[38] Francis Sourd. Scheduling a sequence of tasks with general completion costs. Technical report, 2002.

[39] P. Spindler, U. Schlichtmann, and F.M. Johannes. Kraftwerk2: A fast force-directed quadratic placement approach using an accurate net model. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(8):1398–1411, Aug 2008.

[40] Peter Spindler, Ulf Schlichtmann, and Frank M. Johannes. Abacus: Fast legalization of standard cell circuits with minimal movement. In *Proceedings of the 2008 International Symposium on Physical Design*, ISPD '08, pages 47–53, New York, NY, USA, 2008. ACM.

[41] S. Sutanthavibul, E. Shragowitz, and J.B. Rosen. An analytical approach to floorplan design and optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(6):761–769, Jun 1991.

[42] Xiaoping Tang, Ruiqi Tian, and M.D.F. Wong. Minimizing wire length in floorplanning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(9):1744–1753, Sept 2006.

[43] Zhe-Wei Jiang Tung-Chieh Chen, Tien-Chang Hsu and Yao-Wen Chang. Ntuplace: A ratio partitioning based placement algorithm for large-scale mixed-size designs, 2005.

[44] N. Viswanathan and C.C.N. Chu. Fastplace: efficient analytical placement using cell shifting, iterative local refinement,and a hybrid net model. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(5):722–733, May 2005.

[45] N. Viswanathan, Gi-Joon Nam, C.J. Alpert, P. Villarrubia, Haoxing Ren, and C. Chu. Rql: Global placement via relaxed quadratic spreading and linearization. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 453–458, June 2007.

[46] Jens Vygen. Geometric quadrisection in linear time, with application to vlsi placement. *Discret. Optim.*, 2(4):362–390, December 2005.

[47] Xiaojian Yang, B.-K. Choi, and M. Sarrafzadeh. Timing-driven placement using design hierarchy guided constraint generation. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pages 177–180, Nov 2002.

[48] Mehmet Can Yildiz and Patrick H. Madden. Global objectives for standard cell placement. In *Proceedings of the 11th Great Lakes Symposium on VLSI*, GLSVLSI '01, pages 68–72, New York, NY, USA, 2001. ACM.

[49] Hai Zhou, Narendra Shenoy, and William Nicholls. Efficient minimum spanning tree construction without delaunay triangulation. In *UNI 4.0 SECURITY ADDENDUM, ATM FORUM BTD-SIG-SEC*, pages 192–197, 2001.