Integrated Placement and Routing for VLSI Layout Synthesis and Optimization

by
Ping-San Tzeng

Abstract

This dissertation investigates ways to integrate various VLSI layout algorithms via carefully designed integrated data structures. Such an integrated approach can achieve better overall results by iterating non-sequentially among the various algorithms in a demand-driven manner. The shared data structure which is modified incrementally by all the different algorithms serves as an efficient communication medium between them. This approach has resulted in several new prototype tools, including a new placement program that combines wire-length optimization with a new 2-D compaction algorithm, a new area-routing approach that employs hierarchical rip-up and reroute techniques in an integrated global and detailed routing environment, and also a system that integrates the area router with a placement adjustment algorithm. This integrated system can iterate automatically between area routing and placement adjustment phases to generate optimized results for macro-cell problems with over-the-cell routing.

Acknowledgements

I would like to thank my advisor, Prof. Carlo H. Séquin, for his constant support, guidance and inspiration. He not only gave me many technical advices, he also carefully reviewed this manuscript several times and helped me to improve the writing of this dissertation. Also, I would like to thank Prof. Richard Newton and Prof. David Donoho for reviewing this dissertation and giving me many useful comments. I am also indebted to Glenn Adams, Dr. Ren-Song Tsay, Dr. Han-Young Koh, and Naoto Ichihara, for many enlightening discussions and suggestions.

I am gratefully to Semiconductor Research Corporation for their support of this research project. I would also like to thank Cadence Design Systems, Inc. for giving me the opportunity to use their compactor to generate some of the final results.

Finally, I would like to thank my wife Yu-Ling, my father Po and my mother Shao-Shao for their support in my long graduate study. I dedicate this dissertation to them.

Table of Contents

Table of Contents iii
List of Figures v
List of Tables vii
Chapter 1 Introduction 1
1.1 The Placement and Routing Problem 1
1.2 Dissertation Overview 4
Chapter 2 Placement Using Efficient 2-D Compaction 5
2.1 Placement Paradigm for Resolving Overlaps 5
2.2 Data Structure 8
2.2.1 Definitions 8
2.2.2 Building the RULD-Graph 10
2.2.2.1 Building the Triangulation Graph 10
2.2.2.2 Edge-direction Assignment 14
2.2.3 Linear Constraints from RULD-graph 15
2.2.4 Interpretations and Discussions 18
2.3 Compaction Algorithms 19
2.3.1 1-D Compaction Algorithm 19
2.3.2 2-D Compaction Algorithm 20
2.3.2.1 An Alternative Control Strategy 23
2.3.3 Evaluation of the Compaction Algorithm 24
2.4 Wire-Length Optimization Algorithms 26
2.4.1 Simple Quadratic Formulation 27
2.4.2 Quadratic Formulation using Exact Pin Positions 28
2.4.2.1 Cell-Shifting Algorithm 30
2.4.2.2 Cell-Orienting Algorithm 30
2.4.3 Half-perimeter Formulation 33
2.5 Overall Placement Algorithm 33
2.5.1 Initial Placement Phase 33
2.5.2 First Refinement Phase - Quadratic Optimization 34
2.5.3 Second Refinement Phase - Half-perimeter Optimization 35
2.5.4 Evaluation of the Optimization Phases 36
2.6 Results 37
2.7 Summary 40
Chapter 3 Area Routing with Hierarchical Rip-up and Reroute 41
3.1 The Area Routing Problem 41
3.2 Routing Hierarchy and Data Structure 44
3.2.1 The Wiring Model 44
3.2.2 Partitioning Scheme 45

3.2.3 Data Structure 48
3.3 Routing Algorithms 50
3.3.1 Global Construction Algorithm 51
3.3.2 Global Rerouting Algorithm 52
3.3.3 Detailed Rerouting Algorithm 54
3.3.4 Control Strategy and Congestion Data Structure 57
3.4 Results 61
3.5 Summary 65
Chapter 4 Integrated Placement and Routing 66
4.1 Routing and Placement Adjustment 66
4.2 Basic Idea 68
4.3 Area Routing 69
4.4 Placement Adjustment 70
4.4.1 From Congestion to Space Requirement 70
4.4.2 Moving Cells 73
4.4.2.1 Moving Wires 74
4.4.2.2 Stretching and Shrinking Wires 76
4.5 Overall Algorithm 77
4.6 Results 80
4.6.1 Comparison with Fixed Placement and Compaction 80
4.6.2 Macro-cell problems with Over-the-Cell Routing 84
4.6.3 Channel-based Macro-cell Problems 87
4.7 Summary 88
Chapter 5 Discussions and Conclusions 89
5.1 Integrated Data Structure 89
5.2 Iterative Optimization Using Cooperative Algorithms 90
5.3 Conclusion 91
References 93

List of Figures

Chapter 1 Introduction

1.1 Examples of row-based design styles 2
1.2 Problems of using row structure in macro-cell designs 3

Chapter 2 Placement Using Efficient 2-D Compaction

2.1 Problems of using traditional 1-D compaction in placement 7
2.2 The Basic idea of using a compactor/spacer in placement 7
2.3 Triangulation graphs and edge-swapping operations 8
2.4 A legal RULD-graph and an illegal node 9
2.5 The cut-paths in RULD-graph 10
2.6 A desired triangulation graph versus the Delaunay triangulation 10
2.7 A possible search sequence in finding the enclosing triangle on inserting a new node 13
2.8 Two ordering schemes in traversing a 2-D binary tree 13
2.9 The criterion for deciding the direction of edges in a RULD-graph 14
2.10 Legalizing nodes in a RULD-graph 15
2.11 The explicit constraints in a RULD-graph 16
2.12 The three situations in Theorem 1 17
2.13 Replacing the cut-path in Theorem 1 18
2.14 Comparing a RULD-graph with a constraint graph 18
2.15 Comparing a RULD-graph with a rectangular-dual 19
2.16 Transforming projection constraints to spacing constraints 20
2.17 Changing a horizontal edge to a vertical edge in horizontal compaction 20
2.18 Changing a vertical edge to horizontal edge during horizontal compaction ...21
2.19 Comparison of results from various compaction methods 25
2.20 The time-complexity analysis of 2-D compaction algorithms 26
2.21 The potential moves of a pin in the four orientation-change operations 31
2.22 The changes of the wiring costs during two placement processes 36
2.23 The placement result of the macro-cell benchmark AMI33 38

Chapter 3 Area Routing with Hierarchical Rip-up and Reroute

3.1 The complexity of searching for a feasible rerouting sequence 42
3.2 Comparison of gridless routing and grid-based routing 44
3.3 Four-way partitioning and two-way partitioning 45
3.4 The recursive partitioning scheme 46
3.5 The two different methods in selecting the cuts in partitioning 47
3.6 Dynamic grouping for the detailed routing 48
3.7 The data structure for wires 49
3.8 The use of stretched links 50

3.9 The use of the three routing algorithms 51
3.10 Partitioning a horizontal $1 \times N$ strip into a $2 \times N$ strip 51
3.11 The simple pattern router of the global construction algorithm 52
3.12 The two basic rerouting operations in $2 \times N$ strips 53
3.13 The two rerouting primitives for detailed routing 55
3.14 The changes of the levels in routing a macro-cell example 58
3.15 Examples of the congestion elements 59
3.16 The use of congestion elements in backtracking 61
3.17 An area routing example 64
3.18 The CPU-time v.s. the wire length for area routing problems 65

Chapter 4 Integrated Placement and Routing

4.1 Some routing results may be difficult to compact 67
4.2 Two different types of edges in the $2 \times N$ routing frame 70
4.3 The area covered by an edge of a RULD-graph 71
4.4 Congestion that cannot be resolved by increasing the spacing by the maximum overload count 72
4.5 The two approaches to reconnecting broken nets after moving cells 73
4.6 The surrounding regions for the cells 75
4.7 The use of the edge-covered region to fill the empty space 76
4.8 The layouts in different stages of an actual placement and routing process ...79
4.9 Comparing the integrated placement and routing approach and the Place-Route-Compact approach on AMI33 82
4.10 Comparing the integrated placement and routing approach and the Place-Route-Compact approach on AMI49 83
4.11 An example of the randomly generated cells 84
4.12 Results on AMI33otc with over-the-cell routing 85
4.13 Results on AMI49otc with over-the-cell routing 86
4.14 Resulting layout of AMI33 87

Chapter 5 Discussions and Conclusions

List of Tables

Chapter 1 Introduction

Chapter 2 Placement Using Efficient 2-D Compaction

2.1 The number of the triangles tested when new nodes are inserted into a triangulation graph 13
2.2 Comparison of two different compaction control strategies 23
2.3 Comparison of the compaction results 24
2.4 Evaluation of the cell-orienting algorithm on initial placements of different qualities 32
2.5 Placement results on MCNC macro-cell benchmarks 38
2.6 Placement results on MCNC standard-cell benchmarks 39

Chapter 3 Area Routing with Hierarchical Rip-up and Reroute

3.1 Characteristics of the randomly generated area routing problems 63
3.2 Results on the randomly generated area-routing problems 63

Chapter 4 Integrated Placement and Routing

4.1 The effect of the placement adjustment algorithm on the number of stretched links and conflicts 77
4.2 Comparing the Place-Route-Compact approach and the integrated placement and routing approach 81
4.3 Results of on macro-cell problems with over-the-cell routing 84
4.4 Results of on MCNC macro-cell benchmarks 87
4.5 Comparison of the results on MCNC macro-cell benchmarks 88

Chapter 5 Discussions and Conclusions

Chapter 1
Introduction

1.1 The Placement and Routing Problem

Designing VLSI chips is a complex task comprising many steps, such as architecture/behavior design, logic design, physical design, and various verification steps. The task of *physical design* is to convert a *circuit*, usually consisting of a set of *cells* and a list of *nets*, into a layout that can be realized on a VLSI chip. In addition to *cell design*, the major task of physical design is to *place* all the cells and then *route* all the nets to connect the pins with proper wires and contacts. Normally, the goal of this *placement and routing* process is to generate realizable layouts in minimum area with minimum delay, which is related to the wire length.

In principle, the placement and routing algorithms have to deal with two-dimensional objects in a two-dimensional space. Because true two-dimensional algorithms are normally quite complex, cells are usually organized into special structures such that the placement and routing problems can be reduced to pseudo-one-dimensional problems. The most common approach is to organize cells into parallel rows, as in *gate-array* and *standard-cell* designs (Fig. 1.1). With the row structure, the placement algorithms can compute legal positions for all the cells easily after moving cells around, optimizing the objective functions. In addition, the row structure partitions the routing region into independent routing channels, which can be routed with high efficiency [RF82] [RSVS85]. The combination of the row structure and the channel routing techniques have been very successful in carrying out the placement and routing task effectively.

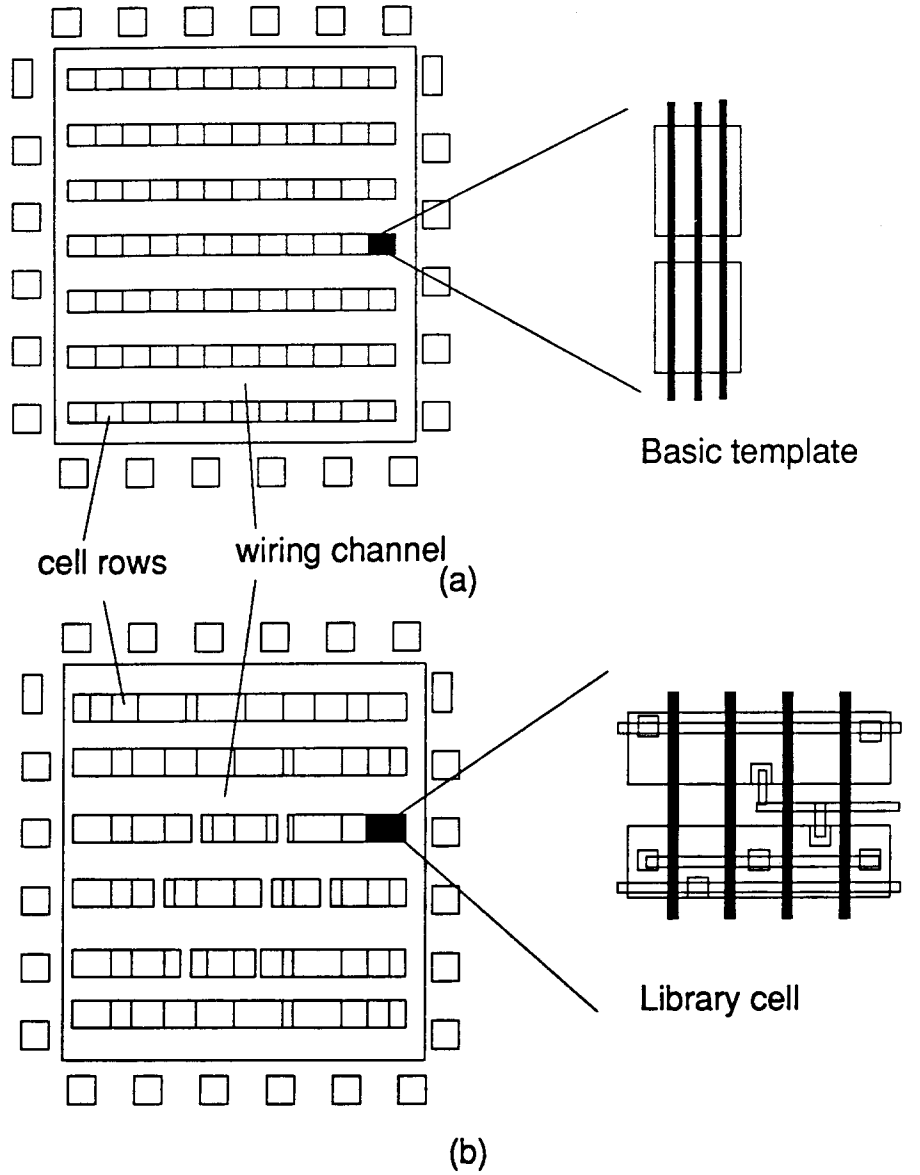


Fig. 1.1 (a) A possible structure of gate-array chips. In such a structure, a library cell may consist of one or several basic templates. (b) A possible cell placement of standard cell designs where the distance between rows can be adjusted based on actual wiring density.

The combination of a row structure and of channel routing provides a satisfactory solution for many VLSI designs but it does not solve all the problems. In many cases, true *2-D placement and routing algorithms* are required. First, many designs consist of complex logic functions such as ALUs, or of dense memory arrays, which have to be implemented with large *macro-cells* of varying sizes and aspect ratios. Usually, a lot of space is wasted when these macro-cells are forced into rows (Fig. 1.2).

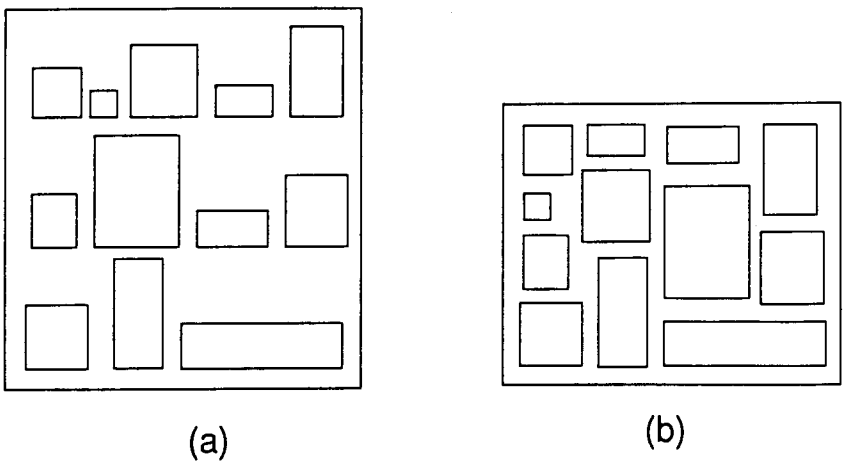


Fig. 1.2 (a) Using row structure for macro-cells can cause a lot of wasted space. (b) Organize the cells in a true 2-D structure can generate a much smaller chip.

A true 2-D placement also requires true 2-D routing algorithms. The channel routing techniques are effective only when the routing region is divided into channels by cells that do not let wires pass through freely. As the manufacturing technology improves, more metal layers become available for routing, and wires at the higher layers can usually go freely over the cells. With these additional routing resources, the cells can be packed together closely and the router has to search for solutions in a true 2-D space instead of in the channels between cells . This kind of problem is usually referred to as *over-the-cell routing* or as *the general area routing problem.*

The *macro-cell placement* and *area routing* problems are difficult because of their 2-D nature. Even though some algorithms have been proposed for the macro-cell placement problem [SB87] [SS90] as well as the area routing problem [Shi87] [TS88], most algorithms have their drawbacks and cannot handle high-complexity VLSI designs efficiently and effectively. In addition to the issues arising when each problem is considered separately, an even more challenging problem is how to integrate the 2-D placement and routing algorithms so they can work together to generate optimized layouts. This interaction is necessary because the placement usually needs to be adjusted so proper space can be allocated between cells to accommodate all the wiring. In the row structure and channel routing scheme, the routing channels can be adjusted easily by moving whole rows vertically. However, in a true 2-D placement and routing problem, cells need to be moved in both horizontal and vertical directions. When there are no independent channels, these moves may render most of the routing results invalid and may require expensive rerouting processes. In most cases, a considerable amount of human intervention is required when the placement needs to be adjusted to complete the routing.

A possible approach to avoiding the expensive rerouting process and/or reducing the amount of human intervention is to use more precise congestion estimate to space cells before conducting the final routing. For gate-array or standard-cell problems, many hierarchical approaches have been proposed to integrate the placement and routing together [BHP83] [Sze86] [SK88] [KPS89] [Chr89]. These hierarchical approaches partition the problem recursively, conducting some form of global routing before proceeding to the next-level placement. With these global routing results, more precise wire-length and congestion estimates can be used to improve the placement and/or to space the cells. In addition to these top-down hierarchical approaches, a similar principle has also been employed in a bottom-up approach using a two-level hierarchy [SLS87]. However, all these hierarchical approaches work primarily on problems with large number (1000's to 100,000's) of small cells of about the same size. These partitioning approaches have not been very successful on macro-cell problems with less than a hundred cells of varying sizes.

In [IBSV89], a system integrating placement and area-routing is proposed to handle sea-of-gate layouts as problems with flexible, porous macro-cells. Using a simplified global router to estimate the routing requirement after the placement step, the system can adjust the placement to ease the routing task. Even though the system can iterate between the simplified global routing and the placement adjustment phases, the actual routing phase still works on a fixed placement. If the router fails due to some local congestion, there is still no automatic solution available.

1.2 Dissertation Overview

Most of the "integrated" placement and routing approaches mentioned above mix placement and routing phases of different levels in a sequential process. The focus of this dissertation concerns the form of integration that iterates between the actual placement and routing tasks in an optimization loop. The primary application targets are the macro-cell problems with over-the-cell routing. The work includes new data structures and algorithms for both the macro-cell placement problem and for the general area routing problem. Based on these new data structures and algorithms, an integrated placement and routing scheme is developed to optimize the layout by iterating between the placement and routing processes.

This dissertation consists of five chapters. Chapter 2 presents the placement algorithm which relies on a *triangulation data structure* and on a *2-D compactor* to resolve overlaps among cells while optimizing the objective function. Chapter 3 introduces the area router, which applies *rip-up and reroute* techniques *hierarchically* to complete the routing task efficiently. Chapter 4 then shows how the hierarchical router is integrated with a placement adjustment algorithm to complete the routing within a minimum amount of area. Chapter 5 presents some of the important lessons learned from this work.

Chapter 2
Placement Using Efficient 2-D Compaction

This chapter introduces a new placement algorithm that uses efficient 2-D compaction to resolve overlaps while optimizing the wire length. Section 2.1 first reviews previous approaches in resolving overlaps and then shows the basic idea of this new approach. Section 2.2 presents the definition and the construction algorithm of the new data structure, and Section 2.3 shows how the compaction algorithm works with this new data structure. Section 2.4 then introduces the wire length optimization algorithms that works with the compactor. Section 2.5 describes the overall placement algorithm, and Section 2.6 presents some placement results as well as a comparison with other placement tools.

2.1 Placement Paradigm for Resolving Overlaps

The objective of placement in VLSI layout synthesis is to place a given set of cells in a two-dimensional space such that a given net-list can be realized with minimum delay and area. Because it is difficult to minimize the delay and area directly, most placement programs try to optimize the wire length. By formulating the objective function as the sum of squared wire lengths, an optimal solution can be found very quickly with quadratic optimization techniques[TKH88]. However, the result is usually *illegal* because it has been obtained without considering overlapping blocks. To obtain a legal layout, additional constraints have to be introduced:

$$|x_i - x_j| \geq (sx_i + sx_j)/2 \quad \text{or} \quad |y_i - y_j| \geq (sy_i + sy_j)/2\,, \qquad \text{for all i, j}$$

where

(x_i, y_i) = position of the center of cell i

(sx_i, sy_i) = size of cell i

For N cells, there will be N^2 constraints that are inherently *non-convex* functions, no matter how they are formulated. These non-convex constraints make it essentially impossible to find a global optimum in polynomial time, and various heuristics have been introduced to enforce these constraints. These heuristics for resolving overlaps usually are the critical element of any placement algorithm and strongly affect the overall performance.

Various approaches have been proposed to resolve overlaps while optimizing the objective function. Among them, there are two basic paradigms: (1) *resolving overlaps gradually* and (2) *generating legal solutions repeatedly*.

2.1 Placement Paradigm for Resolving Overlaps

In the first paradigm, the placement algorithms work on *illegal placements* most of the time while *resolving overlaps gradually*. Only in the final stage of the placement, a legal solution can be generated. Based on this paradigm, a common approach is to formulate the no-overlap constraints as *penalty terms* in a new cost function:

$$C = \text{(wire length cost)} + \lambda \text{ (overlap penalty function)},$$

where

λ = an adjustable weighting factor

For such a cost function, non-linear optimization techniques [SB87] and simulated-annealing techniques [SS90] have been proposed to find a good solution. However, both approaches are very time-consuming yet still cannot always find the global optima. Furthermore, the value of λ strongly affects the quality of the solution and there is no single optimal value for all problems. Usually, λ is increased slowly based on the results from optimization theory [SB87] or based on a statistical analysis of benchmark examples [SS90].

Another placement approach that resolves overlaps gradually is to use *partitioning* [Bre77] [Lau79] [SK88] [TKH88] [KSJ88]. Typically, blocks are assigned to non-overlapping regions in a hierarchical, top-down manner until the partitions are comparable to the size of individual blocks. At this level, if the blocks cannot fit into the final partitions, overlaps may still occur and different heuristics have to be used to remove these overlaps.

This first general paradigm has a major drawback: the intermediate placement results are not legal, so their evaluations are based on estimations that are not really precise. In the second paradigm, placement algorithms *generate legal solutions repeatedly* for more precise evaluations. In row-based design styles, such as *gate-array* and *standard-cell*, legal solutions can be generated easily by assigning blocks to overlap-free rows or columns with linear ordering. However, this pseudo-one-dimensional approach is normally not applicable when blocks are of different sizes and shapes as in macro-block layout or in component placement on printed circuit boards.

For macro-block placement problems, some researchers have introduced creative new cost functions to redistribute blocks evenly from the initial placement. In [Joh87], new objective functions are created based on a planar triangulation derived from the initial placement. In [RJ89], cell area distributions are improved by optimizing the cell density projected on a rotating axis. However, while these approaches reduce the amount of overlap, they fail to produce legal and compact layouts quickly.

Compaction/Spacing is yet another method for resolving overlaps among cells of different sizes and shapes. Compaction has been widely used in enforcing layout rules when converting symbolic layout to mask geometry [HP79]. In placement, the use of compaction has been limited as a clean-up process to generate a legal solution and/or to optimize the total area *after* wiring optimization. Using compaction *during* the placement optimization process, so far, has been impractical because existing compaction algorithms either run too

slowly or cannot generate satisfactory results. In general, graph-based 1-D compaction is fast enough but may not generate dense results for blocks of strongly varying sizes. Two-dimensional compactors [SLW83] [KW84] [Xio89] [SSVS86] can generate smaller results but have been too slow to be used repeatedly in the placement process. Furthermore, existing compaction algorithms may produce poor layouts with high wiring costs because they destroy important neighboring relations among blocks that were derived in previous wiring optimization steps. As shown in Fig. 2.1, compaction of very loose or highly clustered initial placements may result in layouts of unsatisfactory aspect ratios in which the original neighboring relations have been lost.

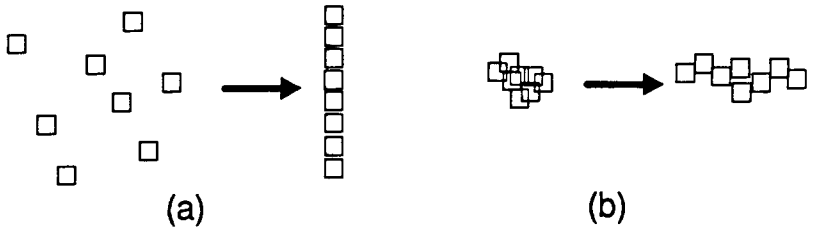


Fig. 2.1 Horizontal compaction of loose (a) or dense (b) initial placements.

This chapter presents a new 2-D compaction algorithm for use in the placement process. Unlike typical layout compactors, the compaction algorithm is designed primarily for spacing and packing rectangular blocks without connecting wires. With a new data structure based on a *planar triangulation graph*, the compactor runs efficiently and is capable of maintaining the important neighboring relations among blocks. With such a 2-D compactor, the second paradigm that generates legal solutions repeatedly becomes feasible for the macro-cell placement problems. This can be achieved by alternating between *wire-length optimization phases* that may create overlaps, and *legalization phases* that use the 2-D compactor to generate dense legal placements quickly (Fig. 2.2). With such a paradigm, the placement tool can evaluate legal placements precisely and can choose the best solution.

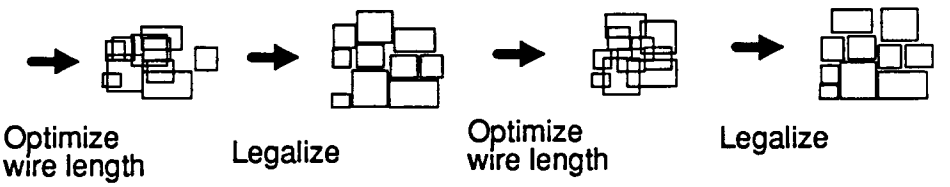


Fig. 2.2 Basic idea of using a compactor/spacer in the placement process.

2.2 Data Structure

2.2.1 Definitions

The basic data structure for the 2-D compactor is based on *planar triangulation.* As defined in [PS85], a graph is **planar** if it can be embedded (i.e. drawn) in a plane without crossings. The embedding of a planar graph determines a partition of the plane. This partition is a **triangulation** if all the bounded regions are triangles. Such an embedding of a planar graph will be referred to as a **triangulation graph.** For a given placement of blocks, a triangulation graph can be built with the centers of all blocks plus four additional boundary nodes, N_R, N_U, N_L, and N_D (Fig. 2.3(a)). For such a triangulation graph, every edge, except the four edges on the boundary of the graph, is enclosed by a quadrilateral. An **edge-swapping** operation on an edge (*A*,*C*) enclosed by quadrilateral *ABCD* is to replace edge (*A*,*C*) with (*B*,*D*) (Fig. 2.3(b)).

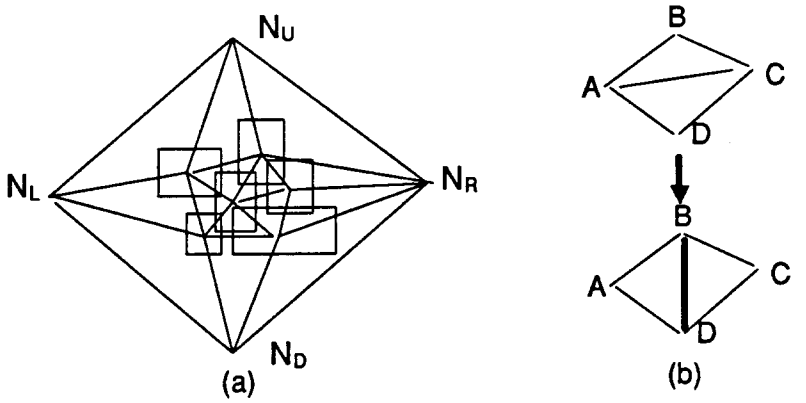


Fig. 2.3 (a) A triangulation graph from a given placement and (b) an edge-swapping operation on edge (*A*,*C*).

In [VKLS90], planar triangulation is used to represent topological constraints between elements for compacting layout in a two-dimensional manner. However, the proposed compaction algorithm is very slow because actual 2-D distances instead of manhattan distances are used in enforcing the constraints. To make use of the fast 1-D graph-based compaction algorithms, the triangulation graph is enhanced by marking each edge with an **orientation**, i.e. *horizontal* or *vertical.* Furthermore, each end of every edge is labeled with the **direction** viewed from the connected node. For a vertical edge, its upper end is marked as **Down** and its lower end as **Up**. For a horizontal edge, its right end is marked as **Left** and its left end as **Right**.

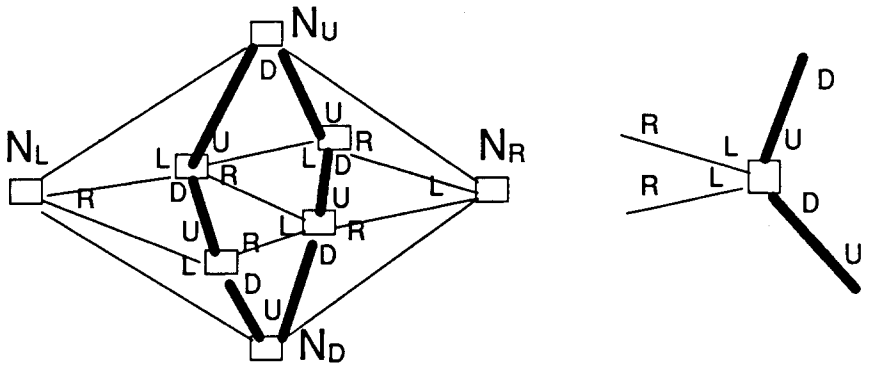


Fig. 2.4 (a) A RULD-graph with four legal nodes and the four boundary nodes. (b) An illegal node without any *Right* edge. *In most figures, bold lines represent vertical edges and thin lines represent horizontal edges.*

Because the triangulation graph is embedded in a plane, the edges connected to each node can be ordered counter-clockwise around the node. If the directions of the connecting ends of these edges form an ordered sequence containing all four types of directions: (*Right...*, *Up...*, *Left...*, *Down...*), the node will be called a **legal** node. If all the nodes in a triangulation graph except the four additional, special boundary nodes N_R, N_U, N_L, and N_D are legal, the graph will be called a **Legal RULD-graph**, or simply a **RULD-graph** (Fig. 2.4).

For a given RULD-graph, a *horizontal* (vertical) **crack-path** is a sequence of nodes, $(n_1, n_2, ..., n_k)$, such that there exist *horizontal* (vertical) edges $e_j = (n_j, n_{j+1})$, with $1 \le j \le k-1$ and the directions of e_j at n_j are all the same. A **cut-path** is a crack-path from one boundary node to the opposite boundary node, e.g. N_R to N_L. A *horizontal* (vertical) crack-path $B = (b_1, b_2, ..., b_n)$ is a **branch** of the *horizontal* (vertical) crack-path $A = (a_1, a_2, ..., a_m)$ if the following conditions hold:

(1) $b_1 = a_s$, $b_n = a_e$, for some s and e such that $1 \le s < e \le m$;

(2) Every edge (b_i, a_j), $1 < i < n$ and $s < j < e$, is *vertical* (horizontal).

Fig. 2.5 shows examples of crack-paths and branches.

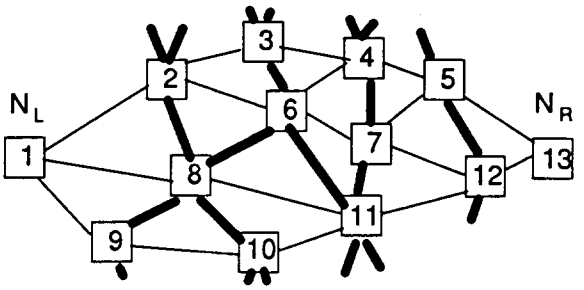


Fig. 2.5 A portion of a RULD-graph. Node 1 and 13 are boundary nodes. For example, (2,6,7) is a horizontal crack-path and (1,8,11,12,13) is a horizontal cut-path. Crack-path (6,4,5) is a branch of the crack-path (1,2,6,7,5,13) but the path (2,3,4,5) is not.

2.2.2 Building the RULD-Graph

Given an initial placement of blocks, a corresponding RULD-graph can be built in two steps. First, a *triangulation graph* is constructed using the center points of all blocks; then proper orientations are assigned to every edge of the graph to create a legal RULD-graph.

2.2.2.1 Building the Triangulation Graph

For a given placement of blocks, many possible triangulation graphs can be built by treating every block as a point at its center. The best-known is the **Delaunay triangulation**, which can be built in $O(n \log(n))$ time for n points [PS85]. However, to allow the compactor to maintain the neighboring relations between blocks, the triangulation graph should have proper edges to connect adjacent nodes. When blocks can be of different sizes and may overlap, a Delaunay triangulation based on the centers of the blocks does not record these relations very well (Fig. 2.6).

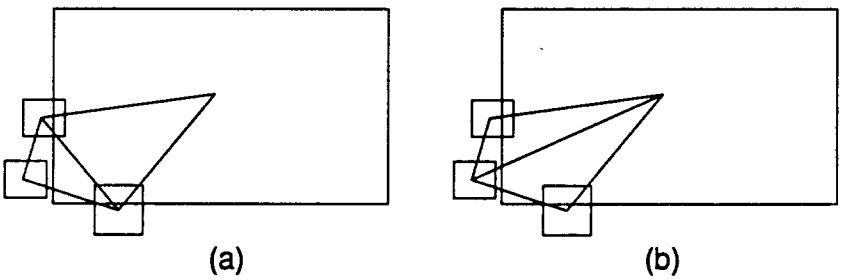


Fig. 2.6 (a) is a Delaunay triangulation while (b) is a more desirable one.

A more suitable triangulation for our purpose is a *minimum-weight* triangulation [PS85] with a weight function based on an "effective" distance that takes the sizes of blocks into account. By assuming the value of 1.0 if two blocks just touch in the orientation of interest, the weight function, *Weight* (i,j) for edge (i,j), is defined as:

$$Weight\ (i,j) = X^2_{adj}\ (i,j) + Y^2_{adj}\ (i,j)$$

$$X_{adj}\ (i,j) = X_{dist}(i,j)/X_{size\_sum}(i,j) \quad \text{if } X_{dist}(i,j) \le X_{size\_sum}(i,j)$$

$$= X_{dist}(i,j) - X_{size\_sum}(i,j) + 1.0 \quad \text{if } X_{dist}(i,j) > X_{size\_sum}(i,j)$$

where:

$X_{dist}(i,j)$ = the x-distance between the center of block i and j;

$X_{size\_sum}(i,j)$ = half the sum of the x-sizes of blocks i and j.

Unfortunately, there is no known polynomial-time algorithm for building a minimum-weight triangulation. Therefore, a *greedy edge-swapping routine* is used to improve the total weight of a given triangulation. This routine works on one node at a time. For a node m surrounded by edges (m, n_i), $1 \le i \le k$, the greedy algorithm tries to swap all surrounding edges (n_i, n_{i+1}) $1 \le i < k$, and edge (n_k, n_1). The edge is swapped if the surrounding quadrilateral is convex and if the new edge has lower weight than the original one. This routine is repeated on the same node until there are no more edge-swapping operations that improve the total weight.

Such a triangulation graph is built incrementally. For a given block placement, the four boundary nodes N_R, N_U, N_L, and N_D are first placed with five edges $(N_R N_U)$, $(N_U N_L)$, $(N_L N_D)$, $(N_D N_R)$, and $(N_R N_L)$, to make a triangulation graph that encloses all the block centers. Then the nodes are inserted one at a time. For every node N to be inserted, the enclosing triangle ABC is found and three new edges (A,N), (B,N), and (C,N) are created. Then the greedy edge-swapping routine is tried on node N. After all the nodes have been inserted, each one is tried once again in random order to improve the total weight of the triangulation. A reasonable local optimum can be achieved by this method. The overall algorithm can be described as the following C-like pseudo-code:

```
build_triangulation_graph()
{
   G = build_initial_graph_with_four_boundary_node();
   for(;;) {
      n = sel_node () ;
      if ( n == NULL ) break;
      insert_node_to_graph(G, n);
      improve_node_by_greedy_edge_swapping(n);
   }
   for_all_node ( n ) {
      mark_node(n);
   }
   do {
      change = 0 ;
      for_all_node ( n ) {
         if ( node_is_marked ) {
            improve_node_by_greedy_edge_swapping(n);
            if ( some_edge_is_swapped ) {
               mark_neighbors(n);
               change = change + 1;
            } else {
               unmark_node(n);
            }
         }
      }
   } while ( change > 0 ) ;
}
```

In this process, all the basic routines can be accomplished in constant time except the operation to insert a new node into the graph because it needs to find an enclosing triangle first. If the nodes are inserted randomly, the average time complexity to find the triangle around the n-th node can be as high as $O(n)$. To speed up this search process, the nodes are sorted based on their positions such that the next node to be inserted is close to the previous one. Under such a situation, the search can start from the node just inserted and choose one of its attached triangles. If the new point is not inside this triangle, an adjacent triangle closer to the point is selected until the enclosing triangle is found (Fig. 2.7).

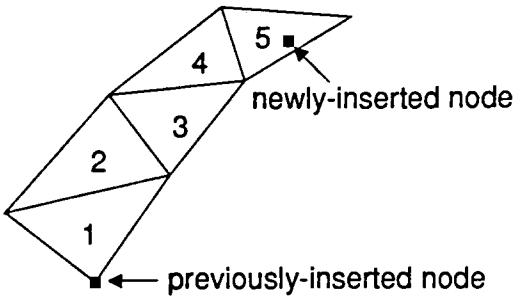


Fig. 2.7 A possible search sequence in finding the enclosing triangle for the new node.

To insert nodes in a proper order, the block centers are sorted into a 2-D binary tree with alternating X and Y cuts such that each leaf node contains at most eight points which can be inserted in random order. Such a tree can be traversed in a depth-first order. If the traversal of the nodes is carefully arranged into Hilbert order [Kah89], it can be assured that the inserted node is close to the previous one (Fig. 2.8(a)). But even with a simple fixed ordering (Fig. 2.8(b)), on average, only three to four triangles need to be tested on every insertion. Table 2.1 shows that this number holds even for large problems with thousands of points.

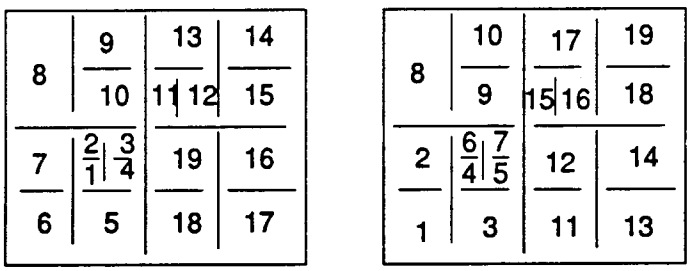


Fig. 2.8 Traversing a 2-D binary tree in (a) Hilbert order and (b) simple fixed ordering.

# nodes	# Triangles tested	
	maximum	average
100	10	3.57
400	34	3.61
900	46	3.76
1600	76	3.81
2500	75	3.95

Table 2.1 The maximum and average number of triangles tested when new nodes are inserted into a triangulation graph. In each example, the N nodes are arranged into a 2-D array with $\sqrt{N}$ rows and $\sqrt{N}$ columns.

2.2.2.2 Edge-direction Assignment

After building the triangulation graph, initial orientations are assigned to all the edges in the graph. Edges with slopes in the range of (−1.0,1.0) are initially classified as *horizontal*, with the directions of their left ends marked as *Right* and their right ends marked as *Left*. All other edges are *vertical* with their lower ends marked as *Up* and upper ends as *Down*. This initial assignment cannot guarantee that all nodes are legal; some nodes may miss an edge in a certain direction. However, such an assignment has two important properties:

(1) The edges are ordered in a counterclockwise traversal around a node as (R..., U..., L..., D...). There may be missing directions but no out-of-order cases.

(2) A node can't miss edges in two consecutive directions (e.g. *Right* and *Up*) because any internal angle of a triangle must be less than 180 degrees.

This initial assignment is simple but it doesn't capture the neighboring relations between blocks very well when the aspect ratios of blocks differs strongly from 1.0 (Fig. 2.9). Therefore, an improvement phase is conducted to get a better orientation assignment based on the following formula for a *desired* direction for all edges:

Edge (i,j) is **horizontal** if $Edge\_slope(i,j) < Critical\_slope(i,j)$
vertical otherwise.

where

$Edge\_slope(i,j)$ $= | Y_{dist}(i,j) / X_{dist}(i,j) |$
$Critical\_slope(i,j)$ $= | Y_{size\_sum}(i,j) / X_{size\_sum}(i,j) |$
$X_{dist}(i,j)$ = the x-distance between the centers of block i and j
$X_{size\_sum}(i,j)$ = half the sum of the x-sizes of block i and j.

In this formula, the *Critical_slope* is defined as the slope of the edge when the corners of two blocks touch as in Fig. 2.9(c).

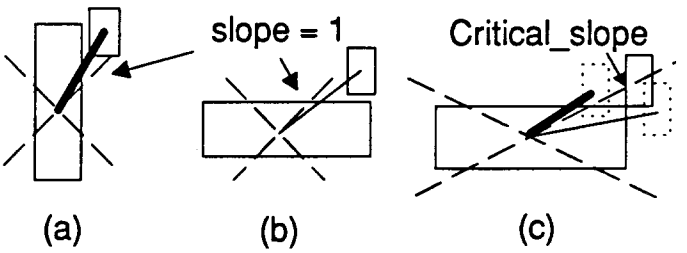


Fig. 2.9 (a) An edge that should be horizontal but with slope greater than 1. (b) An edge that should be vertical but with slope smaller than 1. (c) The interpretation of the Critical_slope in deciding the direction of edges.

In the improvement phase, the desired orientation is computed for every edge. The *desired* orientation will be assigned to the edge if the two important properties mentioned

above won't be destroyed on the direction change. This is necessary since the legalization process works only when the two properties hold.

After the initial classification and the improvement phase based on block size, a legalization process is conducted with edge-swapping operations. Because of the two properties mentioned above, illegal nodes that miss edges in a certain direction will have edges in the two adjacent directions. For a triangle *ABC* where node *A* does not have any, say, *Down* edge, edge (*A*,*B*) and (*A*,*C*) must be both horizontal. Furthermore, edge (*B*,*C*) must be horizontal too or the two properties cannot hold on either node *B* or *C*. With the edge (*B*,*C*) being horizontal, the edges (*B*,*D*) and (*C*,*D*) of the adjacent triangle have to be in one of the following four cases (Fig. 2.10):

(a) both are vertical;

(b) one is horizontal and the other is vertical;

(c) both are horizontal and their directions at D are different;

(d) both are horizontal and their directions at D are the same.

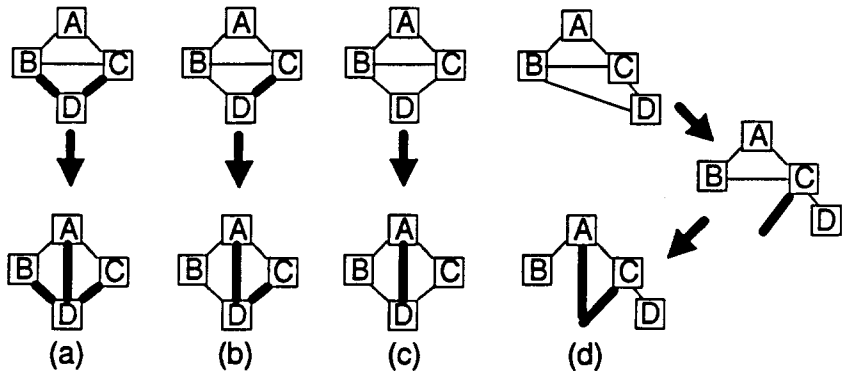


Fig. 2.10 Legalizing node A by introducing a *Down* edge.

For the first three cases, node A can be legalized easily by replacing edge (B,C) with a vertical edge (A,D). In case (d), either B or C (C in Fig. 2.10(d)) is illegal and needs to be legalized first before edge (B,C) can be swapped to legalize A. This algorithm can always legalize all the nodes in the graph. Even though case (d) requires a recursive call of the legalization routine, this process always terminates because eventually, a boundary node will be encountered where case (d) cannot occur. Normally, this recursive process will terminate after just two or three steps.

2.2.3 Linear Constraints from RULD-graph

Like most layout compactors, linear constraints[HP79] are used to space the blocks properly. With enough linear constraints, a legal placement without overlaps can be obtained with the longest path algorithm[HP79]. Two kinds of constraints are derived from a RULD-graph: **explicit constraints** and **implicit constraints**. Each edge in the RULD-graph represents an *explicit constraint*. A *vertical* (horizontal) edge represents a *vertical*

(horizontal) spacing constraint between the two connected nodes. These constraints work just like those used in traditional layout compactors. However, as shown in Fig. 2.11, these constraints alone cannot prevent all overlaps. Rather than adding new edges and destroying the planarity of the graph, such constraints are calculated dynamically whenever a corresponding block constellation is encountered. This leads to the notion of *implicit constraints.*

Explicit constraints fail to prevent overlaps when a vertical edge connects two blocks whose projection on the x-axis don't overlap, such as block B and C in Fig. 2.11. In the traditional 1-D compaction, vertical constraints are generated only when the block's projections on the x-axis overlap. It turns out that all illegal overlaps can be prevented if blocks with vertical constraints between them have overlapping projections on the x-axis. This notion is captured in the RULD-graph representation with a *horizontal implicit constraint*:

$$X_B^{left} \le X_C^{right} \text{ and } X_B^{right} \ge X_C^{left},$$

where

$X_B^{left/right}$: the x position of left/right edge of block B.

Similarly, blocks connected with a *horizontal edge* are assigned *vertical implicit constraints* for their y-coordinates. In some situations, a block may not be big enough to satisfy all such constraints simultaneously (Fig. 2.11). In this case, block B is provided with a *virtual extension* so that $X_B^{right} \ge X_C^{left}$ and $X_B^{left} \le X_F^{right}$.

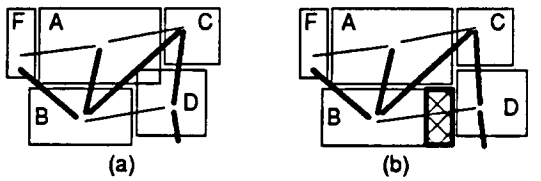


Fig. 2.11 (a) The explicit constraints in the RULD-graph do not prevent all overlaps (A and E). (b) Block B is extended with the shadowed region to satisfy all implicit constraints.

By enforcing explicit and implicit constraints, a legal placement without overlaps can always be obtained based on the following theorems:

Theorem 1: Blocks won't overlap if the *explicit* and the *implicit* constraints are enforced in both, horizontal and vertical, directions.

Proof: For any two blocks, exactly one of the three possible cases below can be true:

(1) both are on the same vertical cut-path;

(2) one is on a vertical cut-path and the other is on a branch of the cut-path; and

(3) there is at least one vertical cut-path between them.

In case (1), the two blocks will be separated by the explicit constraints derived from the edges on the cut-path.

In case (2), the cut-path is assumed to be $(n_1, n_2, ..., n_k)$ and the branch is $(m_1, m_2, ..., m_l)$ and $m_1=n_b$, $m_l=n_e$. For any block n_i and m_j, only one of the three possible cases can happen (Fig. 2.12):

(2a) they are connected by a horizontal edge (n_i, m_j);

(2b) there is a horizontal edge (n_c, m_d) such that $c > i$ and $d < j$; or

(2c) there is a horizontal edge (n_c, m_d) such that $c < i$ and $d > j$.

For case (2a), the two blocks won't overlap because of the constraints from edge (n_i, m_j). For case (2b), block n_c and m_d will have overlaps in their projections on the y-axis (the dotted lines in Fig. 2.12). Therefore, n_i and m_j will be separated by this overlapping part because n_i is higher than n_c while m_j is lower than m_d. The same argument applies for case (2c).

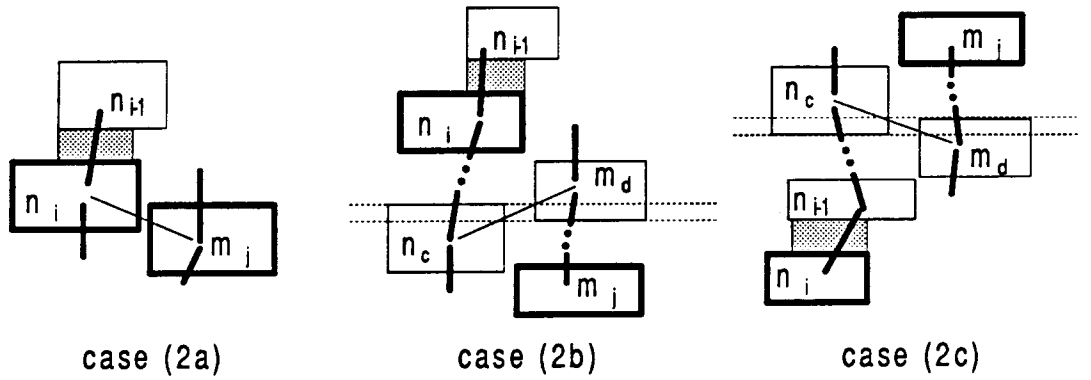


Fig. 2.12 The three situations in case (2). The shadowed regions are the filler rectangles to form the *isolation zone* for the cut-path in case (3).

For case (3), it can be shown that a sequence of *rectilinear isolation zones* can be created for all the cut-paths between the two nodes that separate the two blocks. For each cut-path, its corresponding rectilinear isolation zone covers the blocks on the cut-path and *filler* rectangles drawn between subsequent blocks n_i and n_{i-1} with the width equal to the overlaps of their horizontal projection (Fig. 2.12). These rectangles can be of zero width or height. Based on the same argument used in the proof for case (2), no block on the branch of the cut-path can overlap with these rectangles either.

For the two blocks in case (3), a sequence of cut-paths can be formed from left to right by replacing the old cut-path, $(n_1, n_2, ..., n_k)$, with nodes from one of its branches on the right side $(m_1, m_2, ..., m_l)$ to obtain a new cut-path, $(n_1, ..., n_b, m_2, ..., m_{l-1}, n_e, ... n_k)$ (Fig. 2.13). For such a sequence of cut-paths, a sequence of isolation zones can be formed from left to right. Therefore, the two blocks will be separated by these isolation zones.

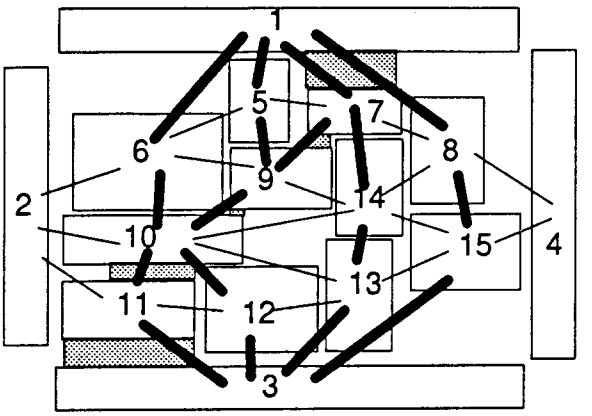


Fig. 2.13 For the cut-path (1,7,9,10,11,3), the rectilinear isolation zone is formed by adding the shadowed filler rectangles between the blocks. By replacing (10, 11, 3) with the branch (10, 12, 3), a new cut-path (1,7,9,10,12,3) can be formed.

2.2.4 Interpretations and Discussions

The RULD-graph replaces the horizontal and vertical constraint graphs used in the original 1-D compaction algorithm [HP79]. Our compaction algorithm also uses the vertical and horizontal edges to represent horizontal and vertical constraints respectively. However, the RULD-graph can easily capture diagonal adjacency relations that cannot be represented in the traditional constraint graph. Without this information, the relative positions between neighboring nodes can be destroyed easily during compaction.

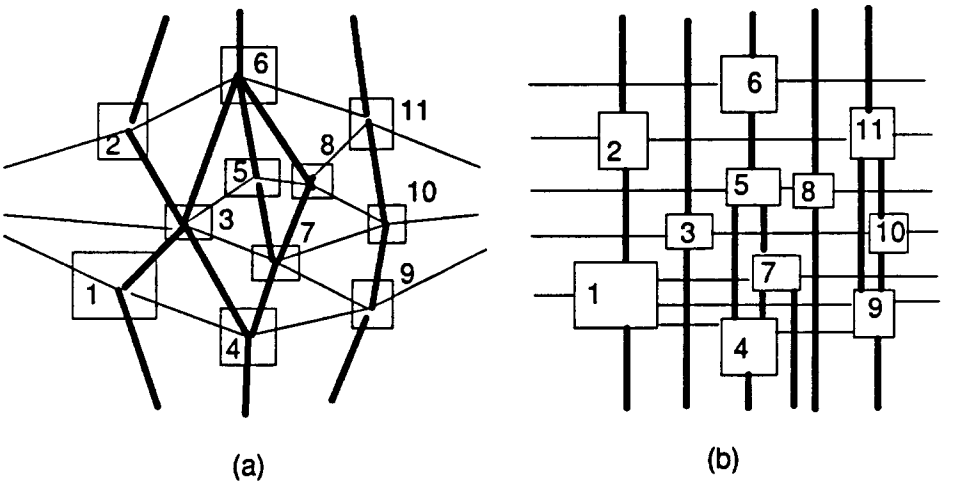


Fig. 2.14 Comparison between (a) RULD-graph and (b) constraint graph. The edges, (1,3) (3,5) and (8,11) in the RULD-graph cannot be represented in the constraint graph.

Another data structure strongly related to the RULD-graph is the *rectangular-dual* for a triangulation graph [KK88] [TFKM91] that forms a partition of the rectangular layout space. For a given partition, a corresponding legal RULD-graph can always be built with all the explicit and implicit constraints satisfied. However, enforcing all the implicit and explicit constraints in a RULD-graph does not always create a rectangular-dual partition. As shown in Fig. 2.15, no rectangular partition can be found for the compaction results such that each block is covered by exactly one partition.

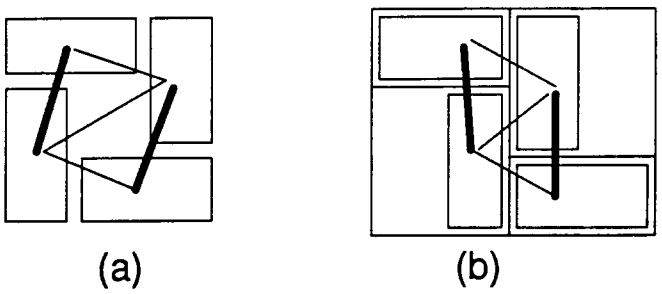


Fig. 2.15 (a) Compaction result for the given RULD-graph. (b) A possible rectangular-dual for the triangulation graph.

2.3 Compaction Algorithms

From a given RULD-graph, a legal placement without overlaps can be computed quickly using a 1-D compaction algorithm to enforce all the explicit and implicit constraints. The result is usually not dense enough, and a 2-D compaction algorithm that moves cells in both x- and y-directions simultaneously is needed to further reduce the total area. This section describes these compaction algorithms and presents the results of some experiments for evaluating them.

2.3.1 1-D Compaction Algorithm

The 1-D compaction algorithm generates results without any overlaps by enforcing all the explicit and implicit constraints. To build the complete constraint graph, implicit constraints are transformed into simple linear spacing constraints. While compacting to the *right*, a block B is represented with two point nodes *B.left* and *B.right*, whose x-coordinates represent X_B^{left} and X_B^{right}, respectively. Node *B.right* will have spacing constraints from the *left nodes* of all blocks on its right that have a horizontal edge to block B (Fig. 2.16). Node *B.left* will have spacing constraints from *B.right* (distance ≥ width of the block) and from the *right nodes* of those blocks that have a vertical edge to block B (distance ≥ 0). Similarly, node *B.left* will have spacing constraints to all *right nodes* of the blocks with horizontal edges on the left side of block B.

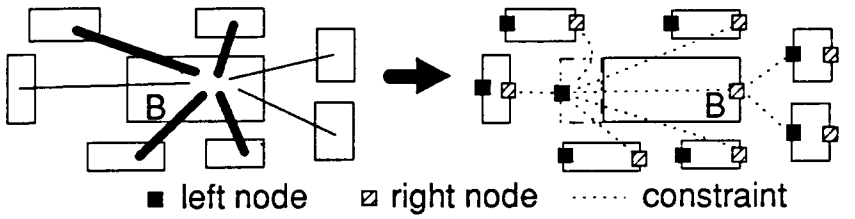


Fig. 2.16 Transforming projection constraints to spacing constraints.

In our actual implementation, this transformation is done in an implicit way. The implicit constraints are derived on the fly. This process, like traditional 1-D compaction, can be computed in $O(n)$ time for n blocks.

2.3.2 2-D Compaction Algorithm

As demonstrated by 2-D compactors [SSVS86] [WMND88], layout density can be improved by moving selected blocks perpendicularly to the direction of compaction. However, such 2-D moves can be quite expensive if the constraint graph and the critical paths have to be recomputed after every move. In our compaction algorithm, these lateral moves are achieved by simply manipulating the edges in the RULD-graph. To shorten the horizontal critical path during the horizontal compaction, the compactor will try to change a horizontal edge into a vertical one. Fig. 2.17 shows four of the eight possible cases of changing a horizontal edge into a vertical edge when blocks are compacted to the *right*. The other four cases can be obtained by flipping these four cases vertically. In principle, there are 16 different possible quadrilaterals around an edge (four edges, two possible orientations), but seven of them are illegal in a RULD-graph. Only seven of the nine legal cases can be changed (two possible moves for one case). In the two remaining cases, the horizontal edge is caught between two other horizontal edges at one node and thus cannot be changed until one of the adjacent edges has been modified.

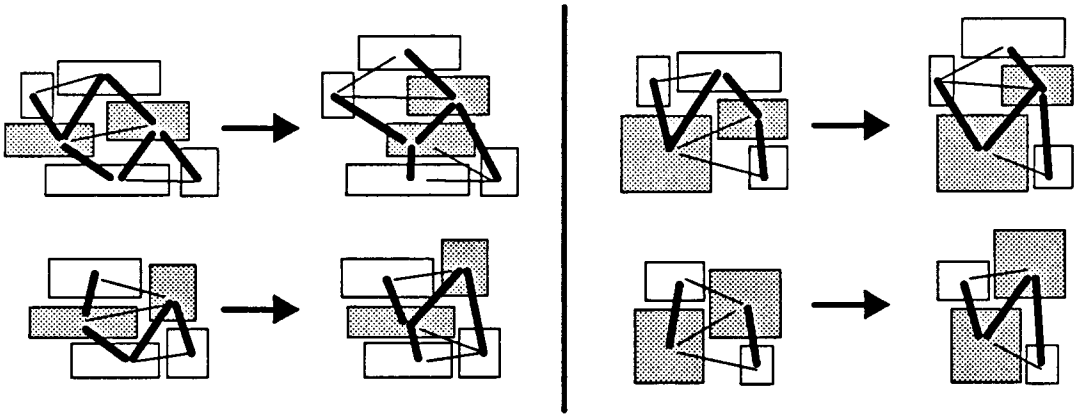


Fig. 2.17 The four basic ways of changing a horizontal edge to a vertical edge while compacting to the *right*.

Sometimes, the critical path consists of implicit constraints as well as explicit constraints. During a horizontal compaction, the implicit constraints are derived from vertical edges so the compactor also tries to replace vertical edges with horizontal edges either to shorten the critical path or to convert them into explicit constraints. There are also eight possible cases of changing a vertical edge into a horizontal edge. Four of them are shown in (Fig. 2.18) and the other four also can be obtained by flipping these four cases vertically. The simple edge-reorientation may then be followed by some edge-swapping operations.

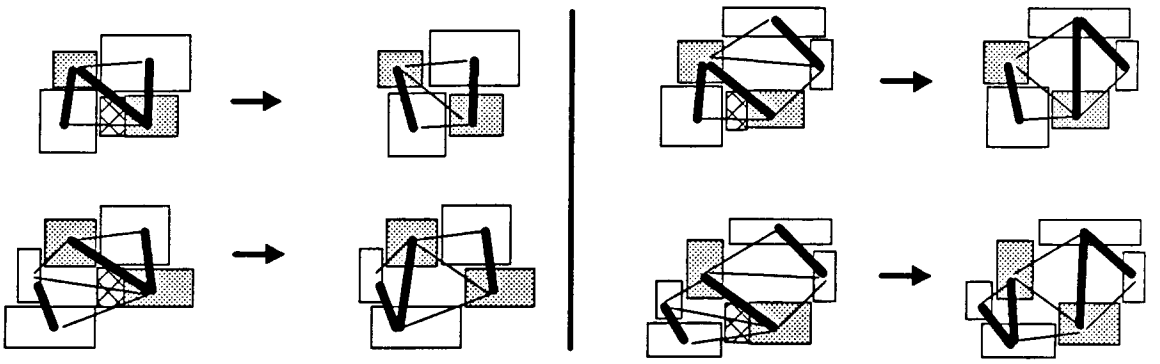


Fig. 2.18 The four possible ways of changing a vertical edge that causes a horizontal implicit constraint on a horizontal critical path while compacting to the *right*.

Because the constraints are generated based on the RULD-graph instead of the absolute position of blocks, there is no need to recompute the constraints on every move. Therefore, these 2-D moves can be achieved much more efficiently than those of other 2-D compactors.

In addition to these moves, the critical path also can be shortened by rotating a block on the critical path by 90 degrees. Such a rotation won't modify any edges of the RULD-graph but it may change both horizontal and vertical critical paths. Because in some types of problem, the blocks cannot be rotated, this operation can be disallowed explicitly.

With this set of 2-D moves in the "toolbox", the remaining problem is to select a proper edge to switch direction. The move selection is based on the *supercompaction* proposed in [WMND88]. For a given placement problem, the compactor tries to generate a dense layout with the specified aspect ratio. The compactor works on one direction at a time trying to generate a smaller layout while keeping the aspect ratio close to the specified value. To compute the size of the resulting layout, the compactor maintains with every node in RULD-graph the values of the longest paths to the four boundary nodes. When the compactor tries to reduce the horizontal size, the compactor checks all the edges and nodes on the horizontal critical path and then selects the move that will cause the minimum increase of the vertical critical path. The increase of the vertical critical path can be computed from the longest paths of the involved nodes to the two vertical boundary nodes. After every

move, the longest paths for all nodes to the four boundary nodes are updated with an event-driven longest path algorithm. The overall control strategy can be summarized by the following C-like pseudo code:

```
2d_compact (desired_ratio_of_X_size_to_Y_size)
{
   aspect_ratio[horizontal] = desired_ratio_of_X_size_to_Y_size;
   aspect_ratio[vertical]  = 1 / aspect_ratio[horizontal];
   best_result = big_number ;
   ori = horizontal;
   ori2 = vertical ;
   new_ratio = critical_path_length(ori) / critical_path_length(ori2);
   if ( new_ratio < aspect_ratio[ori] ) {
       exchange (&ori, &ori2) ;
   }
   for(;;) {
       do {
           mv = select_2d_move(ori);
           make_2d_move(mv);
           new_ratio2 = critical_path_length(ori2) / critical_path_length(ori);
       } while ( new_ratio2 < aspect_ratio[ori2] * (1+ERR_RATIO) )

       new_result = eval_placement();
       if ( new_result >= best_result ) break ;
       best_result = new_result ;
       exchange (&ori, &ori2) ;
   }
   return (best_result);
}
```

In this algorithm, the compaction in one direction is terminated when further moves will cause too much increase of the critical path in the other direction. This is achieved by controlling the resulting aspect ratio to be close to the specified value. If the variable ERR_RATIO is set to be 0.0, the generated layout will have almost the same aspect ratio as the specified value. However, to find a dense solution, the compactor usually has to shorten the critical path in the compaction direction by temporarily increasing the critical path in the other direction. Therefore, the variable ERR_RATIO is normally set to be 0.15 to allow a 15% deviation of the resulting aspect ratio from the specified value so that the compactor won't get stuck prematurely.

When the compaction in one direction is terminated, the procedure eval_placement evaluates the legal placement based on its area and on its estimated wire length, using weights supplied by the user. If no further improvement can be achieved, the compactor will quit and return the best solution.

2.3.2.1 An Alternative Control Strategy

In [TS91b], a control strategy based on *zone-refining* [SSVS86] was proposed. This proposed zone-refining approach can run faster but it usually produces results with longer wire lengths.

This zone-refining approach runs faster because it minimizes the overhead for updating the critical paths. In the direction of compaction, the critical path is computed simply by adding the two critical paths from the two sides separated by the free zone. For the critical paths in the perpendicular direction, the projection constraints are replaced with *quadrilateral constraints* [TS91a] [TS91b], which are easier to compute.

There are several disadvantages to the use of the zone-refining control strategy. First, using the zone-refining control strategy normally causes more 2-D moves than using the supercompaction control strategy to achieve layouts with the same density. This is because the supercompaction approach selects the most promising 2-D move from the possible moves on the critical path, while the zone-refining approach can only move cells adjacent to the free zone. Although all the 2-D moves are local and preserve most of the neighboring relations among blocks, too many 2-D moves, especially when they are all in the same region, can destroy a lot of neighboring relations among blocks. These unnecessary 2-D moves usually increase the wire-lengths of the results.

The other disadvantage of the zone-refining control strategy is the requirement of an explicit size limit in the direction perpendicular to the compaction direction. If the limit is set too high during the compaction process, the compactor makes too many unnecessary 2-D moves and destroys a lot of neighboring relations. If the limit is too low, the compactor may not be able to reduce the critical path to generate a dense layout.

To compare these two control strategies, some illegal placements with overlaps were generated by sampling the intermediate results of actual placement processes on two MCNC macro-cell benchmarks, AMI33 and AMI49. These illegal layouts were then subject to the two different compaction algorithms. Table 2.2 summarizes and compares the results.

		Zone Refining			SuperCompaction		
		wire-length		CPU-time	wire length		CPU-time
Example	#Block	Average	Min.	Average	Average	Min.	Average
AMI33	33	5.76	5.25	1.46	5.20	4.69	1.22
AMI49	49	10.38	8.37	2.32	10.08	8.15	2.66

Table 2.2 Comparison of the results from two different compaction control strategies. There are ten different placements tested for AMI33 and eleven for AMI49. The CPU-times are measured in seconds on a SUN SPARC-1.

The supercompaction strategy can produce better results in terms of wire length and thus potentially result in chips with better performance. On the other hand, it is somewhat slower on larger layout problems. With typical macro-cell problems containing on the order of a hundred blocks, the speed penalty is not considered a serious problem. Therefore, the supercompaction control strategy has been adopted to achieve results with better wire length.

2.3.3 Evaluation of the Compaction Algorithm

The compaction algorithm has been implemented in C in the UNIX environment. Because this compactor is designed for spacing and packing rectangular blocks without connecting wires, the well-known layout compaction benchmarks [Boy87a] [Boy87b] can not be used to test the compactor. To evaluate the compactor, a series of random examples were generated. In these examples, rectangles with random sizes are arranged into a two-dimensional array (Fig. 2.19(a)). Using these examples, we compare our RULD-graph compactor with a very simplistic implementation of traditional 1-D compactor and another 2-D compactor, Zorro[SSVS86]. Table 2.3 summarizes these results. In this table, our compaction results are obtained with four compaction passes, consisting of *Left, Down, Right,* and *Up* compactions, with the target aspect ratio set to be 1. The 1-D compaction results are generated using the pre-compactor of Zorro with four compaction passes, consisting of *horizontal, vertical, horizontal,* and *vertical* compactions. The Zorro's 2-D results are obtained with a horizontal 1-D pre-compaction, and then with four vertical 2-D compaction passes, consisting of *Up, Down, Up,* and *Down* compactions. Because the aspect ratio cannot be specified while compacting with Zorro, each example is given a target width that is equal to our compacted result. Fig. 2.19 shows the initial layout and the three compacted layouts on the example with 400 cells.

	1-D Compaction		2-D: Zorro			2-D: RULD-graph			
#cell	Area	void space	Area	void space	CPU-time	Area	(/ Zorro)	void space	CPU-time
100	72x77	40.9%	65x61	17.4%	5.7	65x64	(1.049)	21.5%	3.0
400	135x149	39.4%	122x113	11.6%	30.3	122x120	(1.062)	16.8%	15.0
900	202x236	44.5%	179x168	12.0%	94.1	179x177	(1.054)	16.5%	40.5
1600	258x313	39.0%	245x226	11.0%	235.1	245x242	(1.071)	16.9%	84.6
2500	307x405	38.3%	304x282	10.5%	496.3	304x300	(1.064)	15.9%	158.5

Table 2.3 Comparison of the compaction results. The "(/Zorro)" column shows the ratio between the areas generated by the RULD-graph compaction and by Zorro. The CPU-times are measured in seconds on a SUN SPARC-1.

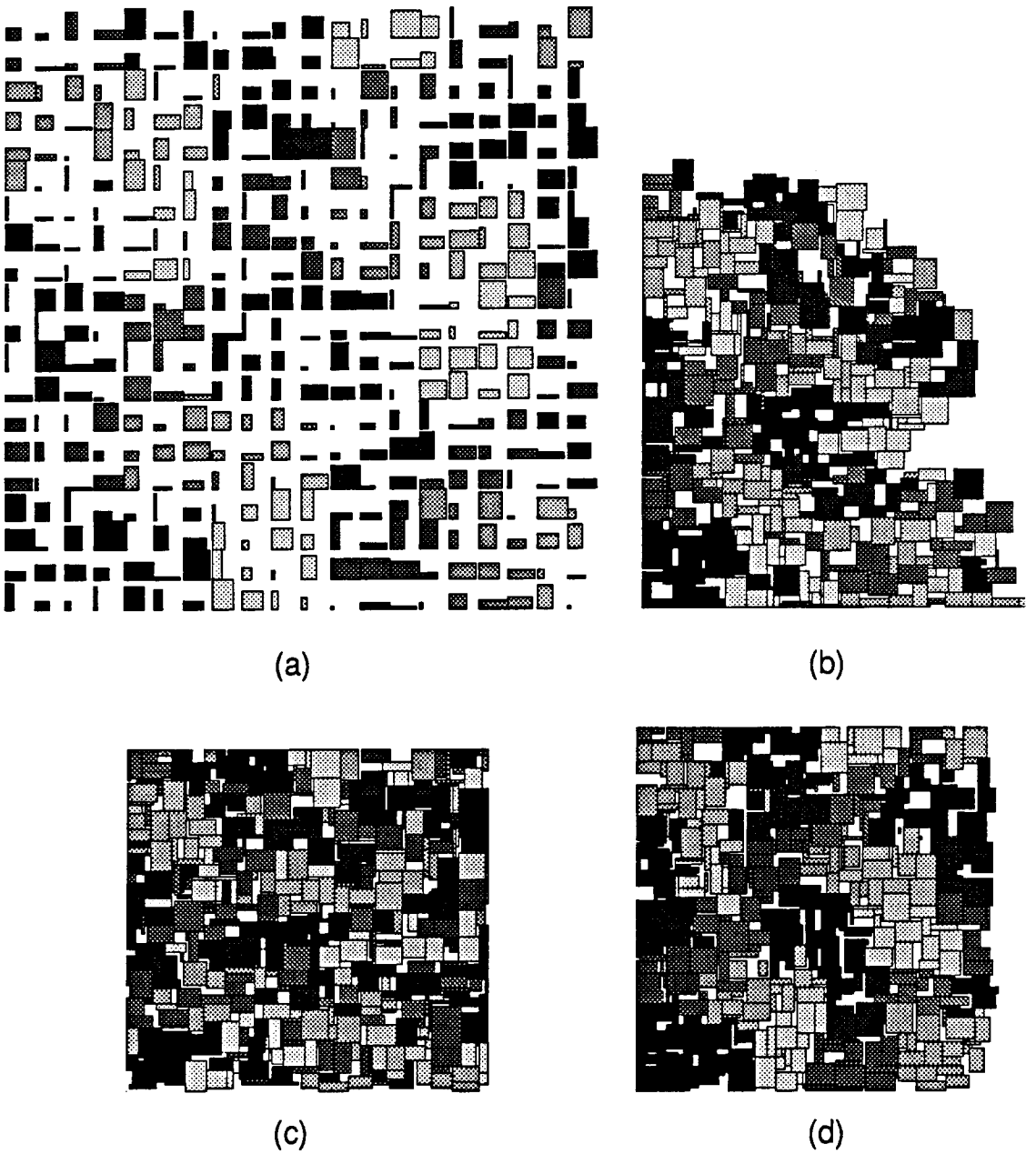


Fig. 2.19 (a) The initial layout. (b) The result of 1-D compaction. (c) The result of 2-D compaction by ZORRO. (d) The result of our 2-D compactor.

The results show that for simple cell-packing problem without wires, the RULD-graph compactor can achieve very compact layouts which are much smaller than the very simplistic 1-D compaction results produced by the Zorro pre-compactor. The results also show that the RULD-graph compactor is much faster than Zorro in 2-D compaction while generating layouts with areas fairly close to Zorro's results. Zorro can produce results that are about 6% smaller because it does not try to preserve the neighboring relations among cells. This

is demonstrated by Fig. 2.19. In these figures, cells are filled with different patterns based on their initial locations. By comparing the pattern distributions of the compacted results with the initial layout, it is clear that our compactor maintains most of the neighboring relations of cells while the other two compaction methods destroy a lot of them.

To further analyze the complexity of our compactor, Fig. 2.20 shows the relation between the run-times and the number of the cells for Zorro and the RULD-graph compactor. It shows that complexity of our compactor is close to $O(n\log n)$ for these cases, where n is the number of the cells.

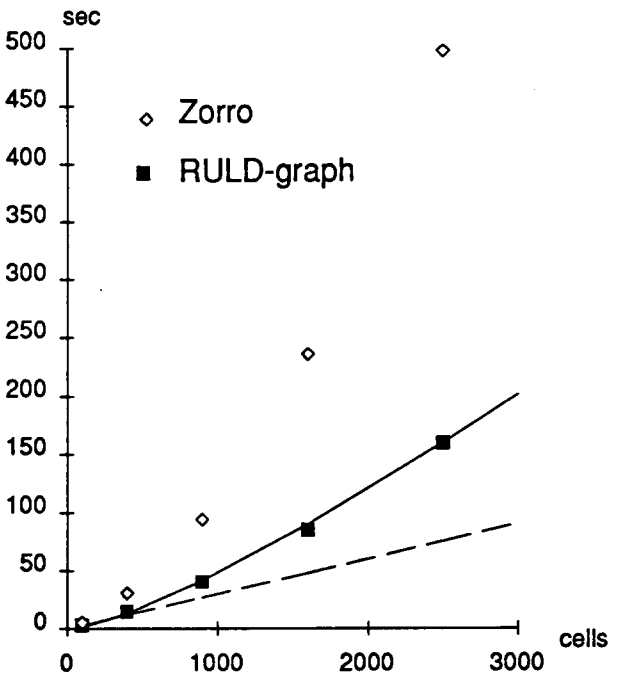


Fig. 2.20 The CPU-time vs. the number of cells for Zorro and the RULD-graph compactor. The solid line comprises points with $O(n \log n)$ complexity and the dashed line is for $O(n)$ complexity.

2.4 Wire-Length Optimization Algorithms

Most placement algorithms optimize an objective function based on estimated total wire length. The most popular estimate used in comparing placement results is the *half-perimeter metric*[Oht85], which calculates the sum of the half-perimeter of the bounding boxes of all nets. This half-perimeter measure is essentially the lower-bound of realizable wire length and is usually a very good estimate of final routed results. However, this is not a continuous function that can be optimized easily and quickly. To make use of standard optimization

techniques, many placement tools formulate the cost function as the sum of the squared wire length, which is a quadratic function.

In comparing two different placements of the same problem of significantly different quality, the one with better quadratic cost normally has better half-perimeter cost, too. However, this correlation disappears when the quadratic costs of the two placements under comparison are roughly equal. To achieve the best result in the minimum amount of time, the quadratic cost function is used initially to obtain a good solution. Then simple greedy algorithms are used to further optimize the half-perimeter cost function.

In the following sections, the formulation of the quadratic functions and the corresponding optimization algorithms will be first presented. Then the simple greedy algorithm that optimizes the half-perimeter cost function will be introduced.

2.4.1 Simple Quadratic Formulation

By assuming that all the pins are in the center of the cells, a simple quadratic cost function F can then be formulated as:

$$F = \frac{1}{2}\sum_{ij} c_{ij}\left((x_i - x_j)^2 + (y_i - y_j)^2 \right)$$

where

(x_i , y_i) = the position of the center of cell i

c_{ij} = the number of connections between cell i and j;

or in the matrix form:

$$F = \boldsymbol{x}^T \boldsymbol{B}\, \boldsymbol{x} + \boldsymbol{y}^T \boldsymbol{B}\, \boldsymbol{y}$$

$$= \sum_{ij} b_{ij} x_i x_j + \sum_{ij} b_{ij} y_i y_j$$

where

$$b_{ij} = -c_{ij} \qquad \text{if } i \neq j$$

$$= \sum_{k} c_{ik} \qquad \text{if } i = j.$$

In computing c_{ij}, a two-pin net between the cell i and j is counted as 1.0. For a net with more than two pins, it is assumed that every pair of cells connected by the net has the same amount of connection derived from the net, which is scaled to be much smaller than 1.0 so these multiple-pin nets won't dominate the wire length cost. This is computed by assuming the quadratic cost F for the net is always 1.0 disregarding the number of pins when the pins are put on a straight line with distance 1.0 between any two adjacent pins. Based on this

assumption, there will be $q-i$ pairs of pins with distance i, for i=1,...,q−1, for a net with q pins. The quadratic cost derived from these connections is:

$$\sum_{i=1}^{q} c\,(q-i)\,i^2,$$

where

c = amount of connection for each pin pair.

By setting this function to be 1, c will be equal to $12/(q^2(q-1)(q+1))$.

With the above quadratic cost function F, the optimal cost is zero when $x_1=x_2=....=x_N$ and $y_1=y_2=....=y_N$, i.e. all the blocks are at the same position. However, when there are fixed pins, the x-part of the equation can be rewritten as:

$$F = \frac{1}{2}\sum_{ij} c_{ij}(x_i-x_j)^2 + \frac{1}{2}\sum_{ik} c_{ip_k}(x_i-x_{p_k})^2$$

where

(x_{p_k}, y_{p_k})= the position of pin k

c_{ip_k} = number of connections between cell i and pin k.

To optimize such a quadratic cost function, the Gauss-Seidel formulation [GV83] is used, which is essentially a gradient-descent method to move each cell toward its optimal position gradually. The x-positions of the blocks can be computed with the following equations.

$$x_i^{(t+1)} = x_i^{(t)} + \frac{1}{b_i}\left(\sum_j c_{ij}(x_j^{(t)}-x_i^{(t)}) + \sum_k c_{ip_k}(x_{p_k}-x_i^{(t)})\right)$$

where

$x_i^{(t)}$ = x-position of block i in iteration t,

b_i = $\sum_j c_{ij} + \sum_k c_{ip_k}$.

The time complexity for computing one move for N cells is $O(cN)$, where c is the average number of cells with connection to each cell. In the worst case, $c=N$ and the time complexity is $O(N^2)$. However, for large problems with many blocks, c is usually much smaller than N and the time complexity is almost linear.

2.4.2 Quadratic Formulation using Exact Pin Positions

The simple formulation introduced in the previous section assumes all pins are at the center of the cells. When the cells are small, such as in the standard-cell or gate-array design styles, the distances from the pins to the center of the cells are usually much shorter than the

average net length, so this simplified model is precise enough. However, for macro-cell problems, this simplified model is no longer a good approximation because the net length can be fairly close to the distance from the pins to the cell center. To estimate the wire length with the exact pin positions, the x-part of the quadratic cost function can be formulated as:

$$F_x = \frac{1}{2}\sum_m \sum_{ij} w_m (x_{p_i} - x_{p_j})^2$$

$$= \frac{1}{2}\sum_m \sum_{ij} w_m \left((x_{c_i} + dx_{p_i}) - (x_{c_j} + dx_{p_j})\right)^2$$

where

$\sum_m$: all nets;

$\sum_{ij}$: all pin pairs in net m;

w_m = weight on each pair of pins for net m;

x_{c_i} = x-position of the center of cell c_i which contains pin p_i;

x_{p_i} = absolute x-position of pin p_i;

dx_{p_i} = $x_{p_i} - x_{c_i}$, i.e. x-position of pin p_i relative to the center of cell c_i.

In most macro-cell placement problems, cells can be rotated or mirrored to one of the eight possible orientations. To allow rotational degrees of freedom, a continuous angle variable θ can be introduced to represent the relative position of a pin to the cell center:

$$\begin{pmatrix} dx_{p_i} \\ dy_{p_i} \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} \Delta x_{p_i} \\ \Delta y_{p_i} \end{pmatrix}$$

where

$(\Delta x_{p_i}, \Delta y_{p_i})$ = relative position of the pin p_i to cell center without rotation.

In [SB87], such continuous angle variables are introduced to allow all blocks to rotate with arbitrary angles. However, with these continuous angle variables, the cost function is no longer quadratic, and general non-linear optimization routines have to be used. Furthermore, to bring the blocks to one of the eight legal final orientations, additional constraints are also required. To simplify the optimization task, heuristics are used to optimize the cost function in two steps. First, a *cell-orienting algorithm* tries to find near-optimal orientations of all the cells for the current placement. Then a *cell-shifting algorithm* moves cells with fixed orientations to their optimal positions. Because the cell-orienting algorithm is based on the concept of *force* introduced in the cell-shifting algorithm. The cell-shifting algorithm will be discussed first.

2.4.2.1 Cell-Shifting Algorithm

In the cell-shifting phase, the orientations of the cells are assumed to be fixed, i.e. the relative position from each pin to the center of the cell, dx_{p_i}, is assumed to be constant. Based on the Guass-Seidel method or a gradient-descent method, the optimal solution can be obtained quickly by a sequence of moves:

$$x_h^{(t+1)} = x_h^{(t)} + \frac{1}{d_h} \sum_{m_h} \sum_{j} w_{m_h} \left((x_{c_j}^{(t)} + dx_{p_j}) - (x_h^{(t)} + dx_{p_{m,h}}) \right)$$

where

$\sum_{m_h}$: all nets with pins on cell h

$\sum_{j}$: all pins in net m_h except pin $p_{m,h}$

$p_{m,h}$ = the pin on cell h that belongs to net m_h

$x_h^{(t)}$ = the x-position of the center of cell h in iteration t

$d_h = \sum_{m_h} \sum_{j} w_{m_h}$

Based on these equations, the nets are processed one at a time for calculating the effect of each net on all the cells with pins belonging to the net. After accumulating the effect from all the nets, all the cells are moved simultaneously. For a problem with N cells and M nets, the complexity of the algorithm is $O(N + d\,M)$, where d is the average of the square of the number of pins for all nets. For macro-cell problems, M is usually greater than N, and this optimization process is slower than the simple quadratic formulation. Fortunately, d is normally a small constant, and the overall complexity is still close to linear.

2.4.2.2 Cell-Orienting Algorithm

In the cell-orienting phase, the cells are rotated or mirrored with their centers fixed. Instead of using the complex and time-consuming non-linear optimization routines as shown in [SB87], simple heuristics based on the concept of *force* are used.

Traditional force-directed relaxation methods [FCW67] [HWA73] use attraction forces derived from the nets to pull connected cells together to minimize the wire length. To fit into the concept of force, the equation for computing cell moves can be rewritten as:

$$x_h^{(t+1)} - x_h^{(t)} = \frac{1}{d_h} \sum_{m_h} \sum_{j} w_{m_h} \left((x_{c_j}^{(t)} + dx_{p_j}) - (x_h^{(t)} + dx_{p_{m,h}}) \right)$$

$$= \frac{1}{d_h} \sum_{m_h} F_{x,pin}(p_{m,h})$$

where

$$F_{x,pin}(p_{m,h}) = \sum_j w_{m_h}((x_{c_j}^{(t)} + dx_{p_j}) - (x_h^{(t)} + dx_{p_{m,h}}))$$

$= x$-component of the force on pin $p_{m,h}$

$\sum_j$: all pins in net m_h except pin $p_{m,h}$

In this new equation, the desired move $x_h^{(t+1)} - x_h^{(t)}$ is the average of the forces on all pins. These forces are just like the forces used in the traditional force-directed method[HWA73]. If each pin can be moved independently, the desired move of each pin would be the force on the pin. Based on this idea, we can evaluate the effect of four possible orientation change operations: *rotate 90°*, *rotate –90°*, *mirror-x*, and *mirror-y*, by comparing the force on each pin to the *potential move* of the pin on each operation (Fig. 2.21).

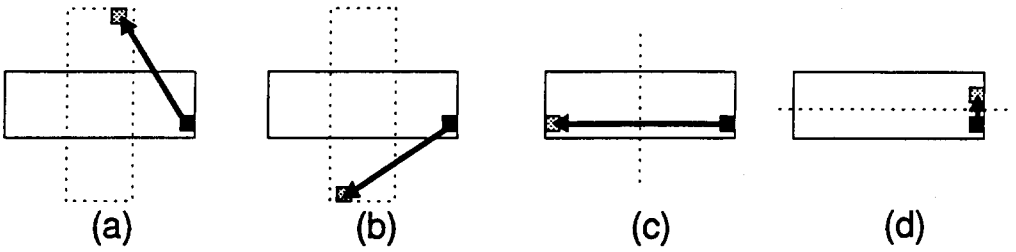


Fig. 2.21 The potential moves of a pin on the four orientation-change operations: (a) rotate 90° (b) rotate -90° (c) mirror-x (d) mirror-y.

For a block with only one pin, the orientation change that would minimize the force on the pin would be advantageous. However, blocks normally have more than one pin and the effects on all the pins of a cell need to be accumulated. The overall effect of operation *op* on a cell *h*, $E_{cell}(h, op)$, is evaluated based on the following equations:

$$E_{cell}(h, op) = \frac{1}{d_h}\sum_{i_h} E_{pin}(i_h, op)$$

$$E_{pin}(i_h, op) = \frac{\text{projection}(F_{pin}(i_h), Mv(i_h, op))}{|Mv(i_h, op)|}$$

$$= \frac{F_{pin}(i_h) \bullet Mv(i_h, op)}{|Mv(i_h, op)|^2}$$

where

$Mv(i_h, op)$ = potential move of pin i_h on operation *op*

$F_{pin}(i_h)$ = force on pin i_h

$\sum_{i_h}$: all pins on cell h;

$\bullet$: inner product of the two vectors.

If the two vectors, $F_{pin}(i_h)$ and $Mv(i_h, op)$, are the same for all the pins of the cell, the function E_{cell} will be 1. Therefore, if $E_{cell}(h, op)$ is greater than 0.5, the operation op is normally advantageous. While optimizing the wire length, these operations first are evaluated for all the cells, and then the most advantageous operations, i.e. the one with the largest value of $E_{cell}(h, op)$, will be taken to change the orientations of the cells. This process is repeated several times on all cells. For most test cases, there will be no further advantageous orientation change operations after this process is repeated for two or three times. Therefore, a simple control strategy is used which repeats this process exactly three times.

This approach turns out to be very effective in minimizing the wire length. Table 2.4 shows the results from some experiments for evaluating this approach. The experiment was conducted on the intermediate legal placements sampled from actual placement processes of the two MCNC benchmarks, AMI33 and AMI49. For each of the four different placements of quite different quality, the cell-orienting algorithm works first to improve the wire-length while fixing the centers of the cells. Then the generated results are processed by a greedy algorithm, trying to further improve the results. The greedy algorithm works on one cell at a time, measuring the wire length for all possible orientations of the cell and selecting the best one. This greedy algorithm cannot guarantee a globally optimal solution, but it can evaluate many possible configurations that may improve the placement. This greedy algorithm runs 10 to 20 times slower than the cell-orienting algorithm and it usually improves the wire length by less than 3%.

Example (scale)	Quadratic Wire Length			
	Initial	After cell-orienting	After greedy-change (improve)	
AMI33	12.42	8.78	8.66	(-1.3%)
	10.52	8.51	8.49	(-0.2%)
	8.79	6.54	6.36	(-2.8%)
	8.17	6.79	6.63	(-2.4%)
AMI49	1980	1637	1599	(-2.3%)
	1000	902	888	(-1.6%)
	911	830	814	(-1.9%)
	753	682	676	(-0.9%)

Table 2.4 Evaluation of cell-orienting algorithm on initial placements of different quality. The unit of the quadratic wire lengths is $10^6 \mu m^2$.

2.4.3 Half-perimeter Formulation

The quadratic optimization routines optimize the quadratic cost function. To further optimize the half-perimeter cost function which is most often used in comparing placement results, a greedy algorithm is used to work directly on this specific cost function. This greedy algorithm is similar to the refinement algorithm used in various placement programs. The algorithm works on one cell at a time, trying to improve the cost function by two kinds of moves - *pairwise interchange* and *orientation change*. The algorithm first tries to swap each cell with one of the cells in its neighborhood. For any given cell, the candidates for swapping can be found easily by a recursive depth-first neighborhood search in the RULD-graph, usually limited to a depth of four to six. When two cells are swapped, the RULD-graph remains unchanged; only the two nodes are exchanged. After all the cells have been tried once, the eight possible orientations of each cell are then tested and the best one will be selected.

2.5 Overall Placement Algorithm

For a given net list, the actual placement process consists of three phases, an initial placement phase and two refinement phases. To simplify the problem, the inputs are assumed to have pre-placed I/O pads on the boundary of the chip. For problems without pre-placed I/O pads, an additional pad-placement phase should be conducted first. A good approach has been proposed in [Tsa89], which first obtains a rough placement of both pads and cells using an eigenvector approach and then maps all the pads to the chip boundary.

2.5.1 Initial Placement Phase

For a given net list with pads on the chip boundary, the simple quadratic formulation is first used to find an initial placement. As shown in Section 2.4.1, by assuming that all pins are at the center of the blocks, an optimal solution with respect to the quadratic wire length can be obtained very quickly. From this initial solution, which is normally illegal, a RULD-graph is built and a legal placement is generated quickly with the 2-D compaction algorithm.

From this placement, the quadratic formulation based on the exact pin positions is then used to obtain a near-optimal solution with the more precise wire length estimate. As shown in Section 2.4.2, the cell-orienting algorithm is used first and then the cell-shifting algorithm. After the cell-shifting algorithm moves the cells, the placement is normally illegal. A RULD-graph is then built and the 2-D compactor is used to restore a legal placement quickly.

Even though the compactor tries to preserve the neighboring relations between cells, many of the adjacent relations between cells may be purely coincidental and not very

meaningful because the initial solution is obtained without any concern for the overlaps. Therefore, two more refinement phases are conducted.

2.5.2 First Refinement Phase - Quadratic Optimization

This refinement phase further optimizes the quadratic cost function by taking into account some of the overlap information. In this phase, the quadratic formulation based on the exact pin positions (Section 2.4.2) is used to optimize the wire length. To take into account the no-overlap constraints, the cell-shifting algorithm is modified slightly. With the modified portion shown in bold-face, the overall algorithm works as follows.

```
optimize_quad_wire_length()
{
    cell_orienting();
    old_cost = eval_quad_cost();
    Init_hp_cost = eval_half_perimeter_cost();          /*modification#1*/
    for (i=1;;i++) {
        compute_gauss_seidel_move();
        If (i == 1) select_move();                      /*modification #2*/
        move_cells();
        new_cost = eval_quad_cost();
        if (new_cost >= old_cost * 0.95) break;
        new_hp_cost = eval_half_perimeter_cost();
        If (new_hp_cost < Init_hp_cost * 0.75) break;   /*modification #1*/
        old_cost = new_cost;
    }
}
```

The first modification is to prevent the cells from moving too close together by limiting the cost improvement in each optimization phase. For some problems, the optimal placement without considering overlaps may have too much overlap and the cost of any legal placement can be much higher than the optimal but illegal placement. To prevent the cells from collapsing into a placement with too much overlap, each optimization phase is usually limited to improving the half-perimeter wire length by only about 25%.

The second modification is to try to generate a placement whose topology is close to the final legal placement. This is achieved by only carrying out "significant moves" that will change the ordering of a cell and one of its neighbors in the RULD-graph. Based on the formulation used in Section 2.4.2.1, if block A and block B are connected by a horizontal edge and $x_A^{(t)} < x_B^{(t)}$, then block A will be moved to new position $(x_A^{(t+1)}, y_A^{(t+1)})$ when $x_A^{(t+1)} > x_B^{(t+1)}$. The moves that won't change the ordering of the cells are rejected because the moved cells very likely will be pushed back close to their original positions after the compaction phase. By keeping these cells close to their final positions, the desired moves of the other blocks can be computed more precisely. This move-selection step is only used

on the first move of each block because the RULD-graph will be destroyed by moves that change the relative positions between blocks.

This quadratic optimization process normally causes new overlaps, and the compactor is used to quickly re-generate a legal placement. From this new legal solution, another quadratic optimization process followed by compaction can be initiated. This optimization-compaction loop works as follows:

```
place_refine_by_quad_opt()
{
    hike_num = 0 ;
    old_cost = eval_quad_cost();
    for(;;) {
        optimize_quad_wire_length();
        2d_compact();
        new_cost = eval_quad_cost();
        if (new_cost == old_cost ) {
            hike_num ++ ;
            if (hike_num > 3) {
                restore_best_placement();
                break;
            }
        }
    }
}
```

In this algorithm, the procedure optimize_quad_wire_length always decreases the wire length but the procedure 2d_compact usually increases the wire length. Sometimes, the wire length of the new legal placement may be worse than the one of the previous legal placement because the overlaps caused by the wire length optimization are difficult to resolve. In this case, the algorithm continues from the new configuration with higher cost to explore more possible solutions. Based on the experiment conducted, no further significant improvements can be observed after such a cost hike happens two or three times. Therefore, the program stops on the 4th cost hike and returns the best solution ever obtained.

2.5.3 Second Refinement Phase - Half-perimeter Optimization

The second refinement phase optimizes the half-perimeter metrics with the greedy pairwise interchange and orientation change algorithm introduced in Section 2.4.3. After each pass of interchange and orientation change, the 2-D compactor, working on the same RULD-graph, legalizes the layout again. Usually, after two or three passes, very few pair-interchanges or orientation changes take place, and no significant cost improvement can be achieved. Therefore, this refinement algorithm is limited to three passes.

Based on the experiments conducted, it is found that swapping two blocks with a large difference in size or shape usually cause excessive overlaps. To resolve these overlaps, the subsequent compaction process usually has to make many 2-D moves which could lead to a considerable increase in wire-length. Therefore, pairs with large differences (> 100%) in sizes or shape, won't be swapped.

2.5.4 Evaluation of the Optimization Phases

To evaluate the contribution of each phase of the placement process, Fig. 2.22 shows the profiles of both the quadratic and half-perimeter cost functions for two macro-cell examples.

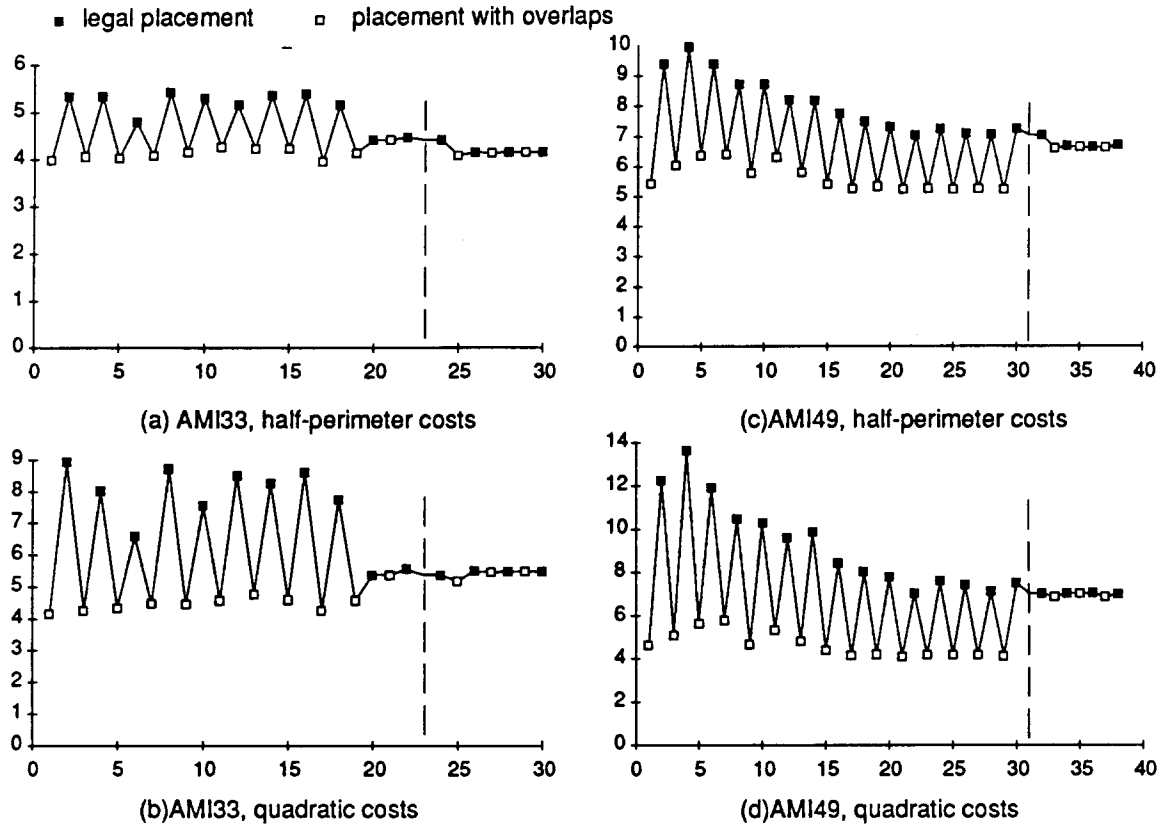


Fig. 2.22 The changes of the costs during the placement process. The first solid dot in each graph represents the first legal placement from the initial placement phase. The dots on the right of the dashed lines are results from the second refinement phase based on half-perimeter cost.

As shown by the these figures, the first legal placement is usually not very good because it is derived from a near optimal solution obtained without concerns about overlaps. The quadratic optimization in the first refinement phase reduces the wire length. Because it tries

to take into account some of the overlaps, the result after compaction can be better. However, sometimes, it may still be difficult to remove the overlaps and the compaction results may get worse than in the previous phase.

In Fig. 2.22, the two examples have different characteristics on the change of the wire lengths. The wire lengths of the intermediate legal placements change somewhat randomly on the smaller example, AMI33, but decrease quite consistently on the larger example, AMI49. This is because the wire lengths of these legal placements depends heavily on whether the illegal placements from the previous wire-length optimization phases are easy to compact or not. Usually, larger examples are easier to compact because there will be more possible ways to rearrange the cells into a dense layout. Furthermore, the larger the example, the less impact each 2-D move in the compaction process will have on the total wire length. On placing the smaller example, AMI33, the big jumps in cost after the 2-D compaction phases show that those illegal placements are difficult to compact. However, near the end of its first refinement phase, a 2-D compaction phase shows only minor increase of the costs. This is because the illegal placement happens to be very easy to compact.

The wire-length optimization based on the half-perimeter cost in the second refinement phase simply exchanges cells of similar sizes so the cells remain evenly-distributed without too much overlap. Therefore, the generated illegal placements are normally quite easy to compact, i.e. to generate a dense legal placement without overlaps, and there is no significant cost increase associated with the compaction process.

These figures also show that the second refinement phase can further reduce the half-perimeter wire-length while increasing the quadratic wire-length slightly. This demonstrates that this refinement phase is necessary to optimize the half-perimeter costs.

2.6 Results

There are not many macro-block placement benchmarks available and also not a lot of published results. The most commonly used ones have been the two MCNC benchmarks, AMI33 and AMI49. Table 2.5 shows our results on these two benchmarks as well as all the published results based on the half-perimeter metric that we know. Among them, GORDIAN [KSJ88] uses quadratic optimization to minimize the wire length and then uses exhaustive enumeration of slicing trees in the final stage to obtain legal placements. ATLAS [SB87] uses non-linear optimization to resolve overlaps while optimizing the wire length at the same time. TimberWolf [SSV84] uses penalty functions and simulated annealing to optimize wire length while resolving overlaps. Tamiya's approach [TFKM91] is also based on simulated annealing but it uses the rectangular dual to generate legal placements. Except for Tamiya's results, all the results have aspect ratios close to 1. To have a fair comparison

with Tamiya's results, we also generated layouts with aspect ratios close to the values reported by Tamiya.

Example			Program	Area (mm^2)	Aspect Ratio	Wire length (mm)	CPU time (sec)
Name	#Block	#Net					
Ami33	33	123	GORDIAN	1.48	na	72.4	5[a]
			TimberWolf	na	na	68.0	1014[a]
			ATLAS	na	na	52.0	714[a]
			Tamiya's	1.42	1.38	53.0	442[b]
			Ours (ratio=1.0)	**1.50**	1.01	**41.4**	**71**[c]
			Ours (ratio=1.4)	**1.48**	1.45	**44.1**	**80**[c]
Ami49	49	408	Tamiya's	43.44	1.19	772.4	1276[b]
			Ours (ratio=1.0)	**46.68**	1.00	**663.3**	**118**[c]
			Ours (ratio=1.2)	**45.89**	1.18	**725.7**	**90**[c]

Table 2.5 Results on MCNC macro-cell benchmarks. The wire lengths were estimated with a half-perimeter metric with no routing space between blocks. The CPU times were originally measured on different machines but were scaled to be roughly equal to the CPU times on a SUN SPARC-1 ([a]: microVAX time divided by 10, [b]: SUN 4/330 time without scaling, [c]: SUN SPARC-1 time). The TimberWolf result is obtained from [SB87]. The ATLAS result still has a small amount of overlap.

Table 2.5 shows that our results have better wire length than all the other published results. Our approach is faster and can achieve better wire length than approaches based on simulated-annealing and non-linear optimization. However, our area is slightly bigger because our program puts more emphasis on optimizing the wire length rather than the layout area. Our approach is slower than GORDIAN but our wire length is much better because of our multiple refinement steps interspersed with the re-spacing steps. The layout of one of our results is shown in Fig. 2.23.

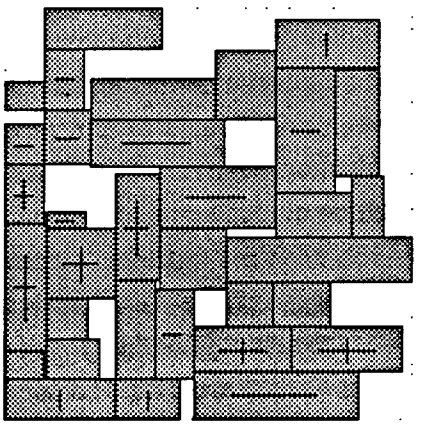


Fig. 2.23 Resulting layout of macro-cell benchmark AMI33.

To test the robustness and limitation of our algorithm, we also tried our placement tool on some standard-cell problems. Table 2.6 shows our results on the two MCNC standard-cell benchmarks, Primary1 and Primary2, as well as the results from PROUD[TKH88], one the best and fastest placement algorithms for row-based design styles. Table 2.6 also includes our results published in [TS91b], which are obtained with the zone-refining control strategy.

Program	Example				Wire length (mm)	CPU time (sec)
	Name	#Block	#Net	Area(mm^2)		
PROUD	primary1	752	904	5.4 x 4.25	1025	65.3
Ours(supercompaction)				(17 rows)	964	818.3
Ours(zone-refining)					995	152.4
PROUD	primary2	2907	3029	9.24 × 8.99	5197	598.3
Ours(supercompaction)				(29 rows)	5143	7953.8
Ours(zone-refining)					5120	1503.2

Table 2.6 Results on two MCNC standard-cell benchmarks. The wire lengths were estimated with a half perimeter metric with ample row spacing.

These standard-cell examples are substantially larger than typical macro-cell placement problems and most macro-cell placement algorithms cannot handle these problems effectively without non-trivial modifications. For our program, the problems are simply treated as a large macro-cell problem except that no orientation changes are allowed. The efficient row-based layout that is natural when all the cells have the same height, is routinely "discovered" by our program.

Even though our program is designed for a much more general purpose, the wire lengths of our results are somewhat better than those obtained by PROUD. However, not being able to take advantage of the special row-structure, our program is considerably slower than PROUD. The difference is quite large when the supercompaction control strategy is used but is less dramatic when the faster zone-refining control strategy is used. As mentioned in Section 2.3.2.1, the zone-refining control strategy usually causes many unnecessary 2-D moves which may increase the wire length. However, the two different compaction control strategies generate results with roughly equal wire-lengths on these examples. This is because the possible change of wire-length cost on each 2-D move is only a very small fraction of the total cost. The additional 2-D moves caused by the zone-refining control strategy won't degrade the results as much as similar moves in macro-cell benchmarks with only a few large blocks.

2.7 Summary

A new placement tool has been developed using a RULD-graph-based 2-D compactor to resolve overlaps between blocks. By using a combination of quadratic optimization techniques and greedy interchange and orientation changes, good results have been obtained. It shows that 2-D compaction based on a suitable data structure is a general and powerful approach for resolving overlaps in all phases of the placement process.

Chapter 3
Area Routing with Hierarchical Rip-up and Reroute

3.1 The Area Routing Problem

Routing is an important task in the synthesis of VLSI layouts. Traditionally, the routing problem has been divided into subproblems with special restrictions, such as channel routing [HS71] [RF82] [RSVS85] and switch-box routing [BP83] [Luk85]. However, newer manufacturing technologies provide more interconnect layers and allow the routing tasks to be completed in smaller regions. To take full advantage of these additional layers, the router has to be able to route over the cells. With such *porous* components, the traditional channel and switch-box routing techniques become insufficient because the pins and obstacles may be distributed throughout the routing region with essentially no restrictions. Such a routing problem is usually referred to as a *general area-routing problem.*

The area routing problem has originally been solved with the *maze routing* algorithm [Lee61] or its variations [Hig69] [Rub74]. These maze-routing-based approaches have two major problems. First, they are time-consuming and require a large amount of memory. Secondly, because the nets are routed one at a time, the nets routed later may be blocked by the nets routed earlier, and the router usually cannot complete the routing.

To speed up the routing process, a number of *hierarchical routing approaches* have been introduced. The most common form is using a two-level hierarchy - *global routing* and *detailed routing*. To further speed up the routing, multi-level hierarchies can be established by partitioning the routing region recursively [BP83] [Lau87] [LHT90]. Hierarchical routing is a *divide-and-conquer* approach, which works best when the subproblems are independent. However, such a clean partition is difficult to obtain when nets are crossing the boundaries between partitions, and various heuristics have to be introduced to deal with the interface between subproblems at the same level. In addition, hierarchical routing approaches also usually suffer from arbitrary decisions made at higher levels, which are usually based on some estimates and may turn out to be unfavorable at lower levels. In such situations, the router usually cannot complete the routing within the limited search space at lower levels.

The most important goal of any routing tool should be to complete as many nets as possible. To achieve this goal, a traditional approach is to rip up some routed nets to create

space for the blocked nets and then to reroute the broken nets latter; this is called the *rip-up and reroute* process. This is a difficult and expensive process because there are usually many possible rip-up and reroute sequences to be tried out. As shown in Fig. 3.1, a blocked net may have many possible paths. For each possible path, there may be several nets blocking the path. Some of these blocking nets can be rerouted easily but some of them may not. Testing each rerouting possibility in a recursive manner can lead to a very large search space very quickly. When a slow maze-router is used to explore all the possible paths, this process becomes extremely time-consuming.

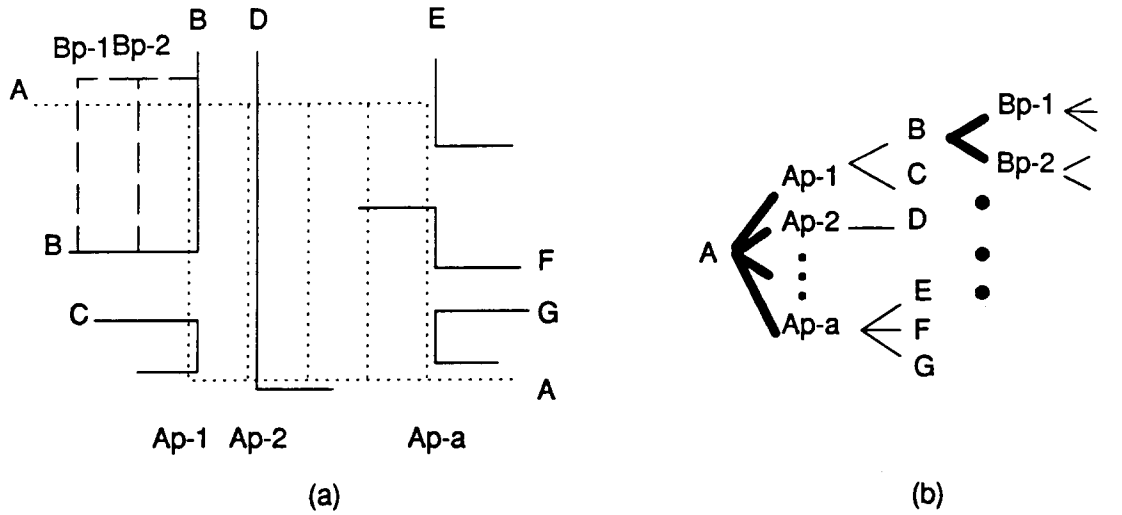


Fig. 3.1 (a) The blocked net *A* may have many possible paths - Ap-1, Ap-2, . . . , Ap-a. For each path, the nets blocking the path (e.g. B, C for Ap-1) have to be tested for rerouting. (b) The search tree for finding a feasible rerouting sequence can grow very fast.

To speed up the rip-up and reroute process, the router must *reduce the possible paths tried for each net* and/or *use fast rerouting operations to test the nets blocking the paths*. In [Shi87], good results were obtained by using a two-level search with fast modification-based rerouting algorithms. In [TS88], by extending the modification-based rerouting primitives into a more general form and conducting searches with more levels, the router can solve more difficult problems. However, all these rerouting operations can only work reasonably well in a small region on the detailed routing grid.

For large routing problems, a hierarchical approach can complete most of the nets very quickly. For the remaining blocked nets, rip-up and reroute algorithms may be used in small local regions on the detailed-routing grid. If no solution can be found in these local regions, the router has to expand the search regions, and the speed of the router may then slow down very quickly. This process usually becomes the bottleneck of the whole routing process.

In principle, the hierarchical routing approach can also be used to speed up the rip-up and reroute process. To conduct rip-up and reroute in a hierarchical way means that the router has to be able to move up to higher levels with coarser grids to conduct "rip-up and reroute" when it gets stuck at lower levels. In such a bottom-up backtracking process, the router has to address the interface problem between the data structures and algorithms of the different levels. This problem, combined with the original interface problem between subproblems at the same level, makes hierarchical rip-up and reroute a difficult and complex process. Therefore, most hierarchical approaches are designed only as fast, top-down construction processes.

In [IBSV89], a hierarchical router is proposed that can automatically enlarge the search region for the rip-up and reroute process when the router gets stuck at any level. However, this recovering process can be quite expensive when it is initiated at lower levels with very fine routing grids.

In [LHT90], a hierarchical router is proposed that can move up and down in the hierarchy by mapping wiring data between a set of two-dimensional maps with different grid sizes. However, the approach is designed primarily for speeding up the maze-search and does not show advantages in speeding up the rip-up and reroute process.

In [Lee90], several approaches have been proposed to conduct rip-up and reroute under a fixed routing hierarchy with pseudo-pins on the cut boundaries. The proposed approaches address the problem caused by excessive nets passing through congested partitions. However, there is still no clear solution for the problem caused by poor pseudo-pin assignment on the cut boundaries.

In [TS88], some form of global routing that extracts congestion information from the detailed routing result is used to speed up the rip-up and reroute process. However, the capability of the global router is limited and the router can efficiently handle only medium-size problems with up to a few hundred grid lines.

This chapter introduces a new area routing approach that can complete large routing tasks efficiently by using a hierarchical approach to speed up the rip-up and reroute process. By building a multi-level hierarchy with only a small number of grids at each level, the number of possible paths for each net can be limited, and all alternatives can be evaluated quickly. Therefore, the search for good rip-up and reroute sequences can be achieved reasonably fast. A unified routing database, shared by all levels, allows the router to move between levels easily. To make the interaction between different levels effective, a special data structure has been developed for the exchange of congestion information between levels.

Section 3.2 describes the multi-level routing hierarchy and its underlying data structure. Section 3.3 then describes the routing algorithms used in various phases, the overall routing

algorithm, as well as the congestion data structure. Finally, Section 3.4 presents some performance measures on actual routing problems.

3.2 Routing Hierarchy and Data Structure

This section first introduces the grid-based wiring model used by the router and also shows how the grid space is partitioned to construct the routing hierarchy. Then it presents the underlying data structures that allow the router to move easily between different levels of the hierarchy.

3.2.1 The Wiring Model

Automatic routers create wires and vias to implement nets. To ensure that these wires and vias can be manufactured on a real chip, some spacing rules between these elements have to be satisfied. Based on these spacing rules, there are two common wiring models - gridless and grid-based (Fig. 3.2). The gridless approaches allow wires and vias to be put anywhere in the layout as long as there are no spacing-rule violations. The grid-based approaches first create a grid with sufficient separations between grid lines and then put wires and vias only on the grid lines. Theoretically, the gridless approach can find a solution in a smaller space. However, the gridless approaches are normally much slower than the grid-based approaches because of the complexity of checking the spacing rules while searching for feasible paths. Therefore, most practical routers are grid-based, and so is the router presented in the chapter. To minimize the space between the wires generated by a grid-based router, layout compaction techniques [HP79] may be employed.

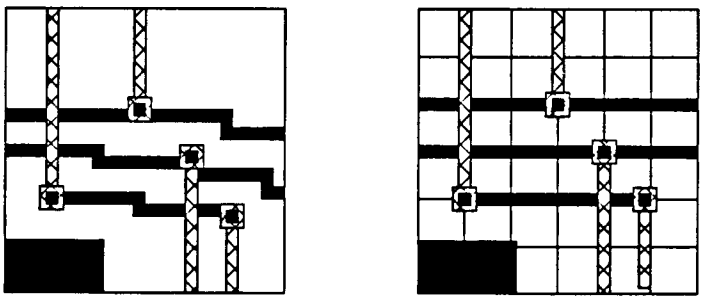


Fig. 3.2 (a) Gridless routing and (b) grid-based routing.

For vias and wires, there are normally three types of spacing rules, wire-wire, wire-via, and via-via. In most cases, the relations among these rules are:

$$RULE_{wire\text{-}wire} \leq RULE_{wire\text{-}via} \leq RULE_{via\text{-}via}.$$

Normally, the space between the grid lines is set equal to the wire-via rule so the spacing rules can be satisfied in most situations. If the via-via rule is greater than the via-wire rule, the router can either avoid putting vias in adjacent grid points or ignore these rules at all, leaving the problem to the following compaction phases. For the router presented in this chapter, the latter approach is adopted to simplify the routing problem.

In accordance with the above grid model, the router assumes all the input geometries, including pins, wires, obstacles, etc., are on a predefined grid. This grid is normally referred to as the *detailed routing grid*, to be distinguished from the grid formed by the cut-lines created in the global routing phase. The router creates wires and vias only on the grid lines, which are also called the *detailed routing tracks*, or simply *routing tracks*. Each *wire segment*, or simply *wire*, occupies a portion of a routing track. Wires or vias from different nets cannot share the same portion a routing track on the same layer. At the global routing level, the router measures the routability based on the number of *free tracks*. A routing track is *free* if the portion being considered is not occupied by any wires or obstacles.

3.2.2 Partitioning Scheme

There are two basic methods to partition a rectangular routing region recursively - *two-way partitioning* and *four-way partitioning* (Fig. 3.3).

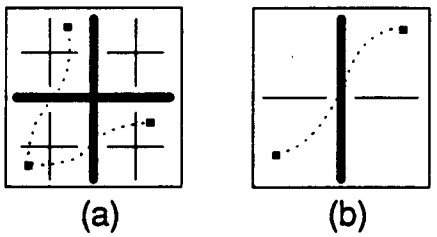


Fig. 3.3 (a) 4-way partitioning and (b) 2-way partitioning. The thick lines are the highest-level cut lines and the thin lines are next level down. The points and dotted curves show nets crossing the highest-level cut lines.

As shown in Fig. 3.3, nets that cross the highest-level cut lines have to be processed again at the next level down. Without breaking these nets, the subproblems cannot be processed independently and the hierarchical routing may become fairly complex and inefficient. A common approach to building a clean hierarchy is to create *pseudo-pins* on the cut lines to break the nets into several parts that can be processed independently. However, it is difficult to find good pin assignments that make all subproblems easy to route, and it is even more difficult to recover from a bad pin assignment made at high levels. Furthermore, pseudo-pins usually fracture long nets that cross many partitions into several segments linked by undesirable jogs. In [Lau87], a *linear-assignment* algorithm is used to find an optimal pin assignment for a special cost function. However, the cost function is based mainly on wire

length and does not take into account important internal congestion information for each partition.

In [BP83], a "N to $2 \times N$" partitioning scheme was proposed. This is essentially a two-way partitioning scheme in which all the subproblems on the same horizontal or vertical slices are *grouped* together to be processed simultaneously (Fig. 3.4). Under such a grouping scheme, no pseudo-pins are required. Furthermore, long nets crossing multiple subproblems won't be broken into many segments by the pseudo-pins and can be processed with a global view.

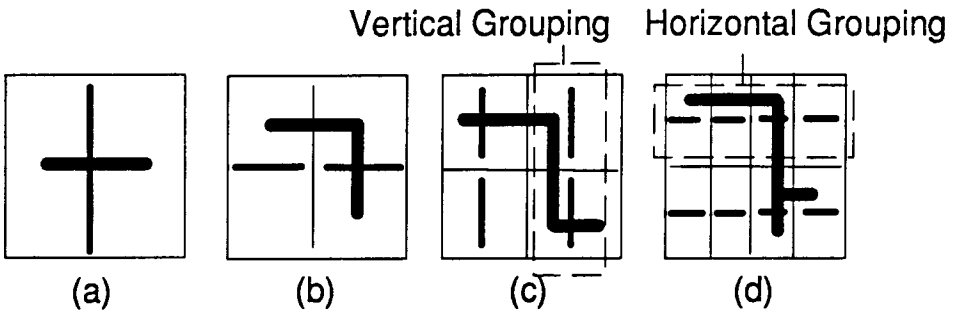


Fig. 3.4 The partitioning scheme. (a) First vertical cut with a horizontal wire. (b) First horizontal cut. (c) Vertical cuts with vertical grouping. (d) Horizontal cuts with horizontal grouping. In these figures, the thickest lines represent wires; the medium thick lines indicate latest cuts; and the thin lines represent cuts made at higher levels.

In comparison with the clean partitioning scheme based on pseudo-pins, this grouping approach may get slower at lower levels because the $2 \times N$ routing region may have a fairly large N. However, this is justified by the advantage to have a global view in processing long nets crossing many partitions. To minimize the interface problems between partitioned regions, this partitioning scheme is adapted to build a multi-level hierarchy. In the resulting routing hierarchy, the levels are characterized by two numbers, (*horizontal-level, vertical-level*). At level (h,v), the partitioned regions cover exactly 2^h vertical grid lines and 2^v horizontal grid lines except those at the right and upper boundaries of the routing region, which may cover a smaller area. To move from level (h,v) down to $(h-1,v)$, the router makes horizontal cuts at distances from the bottom equal to $2^{h-1}+n \times 2^h$, n=0,1,... . Similarly, to move from level (h,v) down to $(h,v-1)$, the router makes vertical cuts at distances from the left boundary equal to $2^{v-1}+n \times 2^v$, n=0,1,... . The router always tries to cut partitions on the direction with the larger dimension. For example, if $h \geq v$ at level (h,v), the router will make horizontal cuts to move to level $(h-1,v)$; otherwise, the router will make vertical cuts to move to level $(h,v-1)$.

This partitioning method chooses cut-lines differently from other hierarchical approaches, which normally make the first cut at the center the routing region. By making the cuts at the positions that are powers of 2, uneven partitions may be created at upper and righ edges of the routing region Basically, there is no significant difference between these two approaches except that uneven partitions may introduce more errors in estimating the wire lengths. On the positive side, this partitioning method allows the router to derive quickly the routing hierarchy, i.e. the partitioning boundaries, without creating additional data structures to store partitioning information. This is very important so that the router can move efficiently between different levels during the rip-up and reroute process.

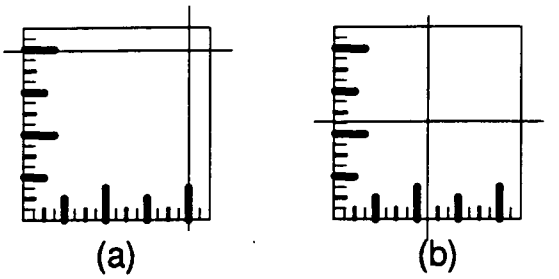


Fig. 3.5 (a) Making cuts at the positions that are powers of 2; and (b) making cuts to produce even partitions.

In [Lee90], some intelligent cut-path selection algorithms are proposed to create partitions that are easier to route. However, these algorithms create irregular partitions with zig-zag cut paths. These irregular partitions make it much more expensive to determine the enclosing partitions of arbitrary wires or obstacles. Because our router tries to resolve the congestion by moving between different levels of the hierarchy and by changing the hierarchy dynamically, straight cut-paths are used to simplify the computation.

Our recursive partitioning process measures routability based on the number of nets and free tracks crossing the cut lines. This measure works reasonably well when the number of free tracks is much greater than 1. However, as the partitioned regions become smaller, this measure becomes less precise and the routing algorithm becomes ineffective. Therefore, this recursive partitioning scheme, which is called *global routing*, normally stops at level (3,3). Then a *detailed router* takes over, working directly on the detailed routing grid with an explicit and exact measurement of routability, i.e. "*can the wiring be completed without any conflicts*".

When the global routing process stops, the whole routing region is partitioned into a 2-D array of *global routing cells* (Fig. 3.6) and wires are assigned to regions covered by one or several cells. In most traditional approaches, these subregions are routed one at a time by a detailed router with pseudo-pins created on the boundaries between the subregions, but this assignment of pseudo-pins is difficult. Inspired by the *grouping strategy* used in global routing, the use of pseudo-pins is avoided by grouping the detailed routing regions dynami-

cally while ignoring the partitioning boundaries created by the global routing algorithm. For each wire segment generated by the global routing algorithm, the detailed router first assigns the whole segment to a single detailed routing track in the partitions traversed by the segment. This assignment does not take into account conflicts caused by overlapping wires. To resolve these conflicts, the detailed router then tries to *reroute* each involved wire segment in a dynamically selected rectangular region surrounding the wire segment (Fig. 3.6). With such a grouping scheme, there is no need to break wires with pseudo-pins.

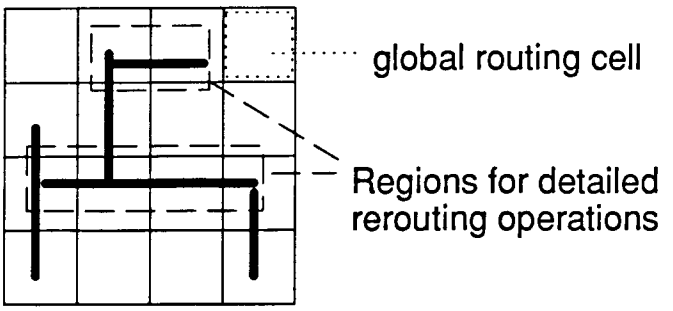


Fig. 3.6 Dynamic grouping for the detailed routing.

3.2.3 Data Structure

A key idea of our routing algorithm is to move among coarser and finer levels to complete the routing task with efficient rip-up and reroute operations at different levels of abstraction. It is very important for the router to transform data between these different levels with as little overhead as possible. To preserve the data of each level separately is not feasible because it would require too much memory and management overhead to maintain the consistency of the whole database. Instead, a unified database has been developed to be shared by all the levels. To make such sharing possible, all wiring information is stored in a form compatible with the final detailed wiring description. For higher-level routing, the data corresponding to the coarser-grid is extracted and abstracted on the fly.

The primary elements of the routing database are wire segments that are either horizontal or vertical. Every wire segment is assigned to a horizontal or vertical detailed routing track. To speed up the search for wires in a certain region, one or more doubly-linked lists are formed for the wires on the same line. Each linked list contains a set of wires without overlaps so they can be sorted based on their positions. For a final legal layout, each detailed routing track needs exactly one such linked list for each interconnect layer. For the intermediate results with conflicts, each track may need several linked lists for one layer. In practice, because the router distributes wires evenly in the routing region, the wiring density won't get so uneven that one track has to carry a large number of linked lists.

In our implementation, wires of different layers can be stored in the same linked list. This is because the wires of a particular layer run predominantly in the same direction and each track usually requires no more than $N/2$ linked lists for a legal routing result of an N-layer problem. By mixing wires of different layers together, the number of linked lists on each track can be even lower, especially in regions with low wiring density.

With the sorted wire lists, the key mechanism to speed up the data access is a *cache pointer* associated with each sorted list that points to the wire that is just accessed. Because the router works on the routing region in a proper order, e.g. from left to right, most search regions are very close to the previous search regions. Therefore, only a few wires need to be checked if the search starts from the current cache pointers. Fig. 3.7 shows the structure of the linked list.

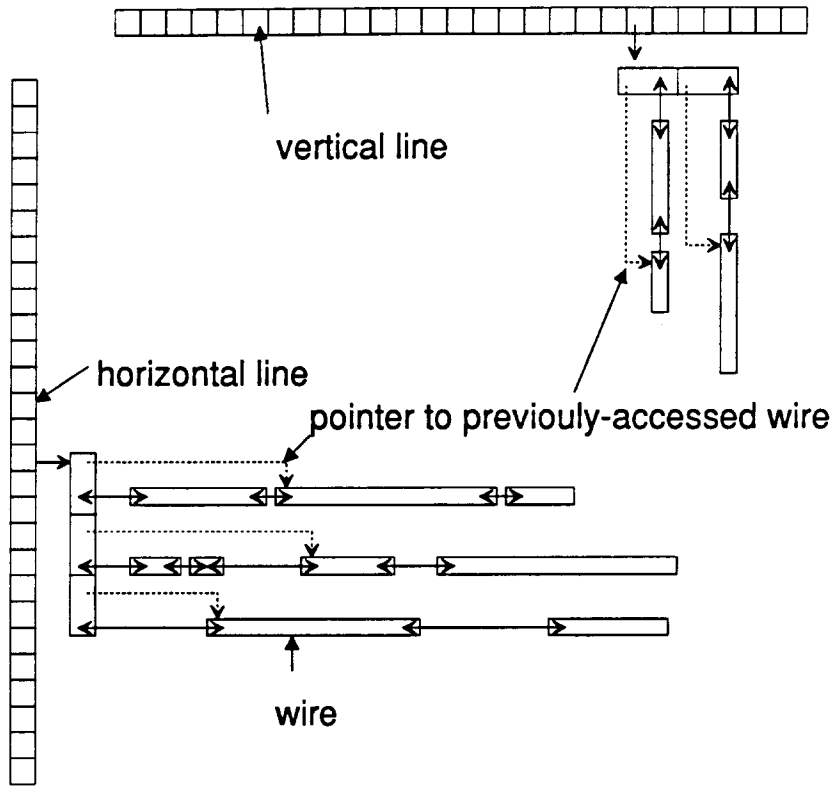


Fig. 3.7 The data structure for wires.

With such a data structure, we can find all the wires in a $2 \times N$ strip for the global routing task quickly and build a proper map for the $2 \times N$ routing. However, representing global routing data on detailed routing tracks introduces a new problem - the same data abstraction must be obtained for both the top-down and bottom-up processes.

In the top-down process, the global routing algorithm looks at a very coarse grid and creates only one wire to connect two elements lying in the same row or column in the $2 \times N$ strip. These two elements may be on different detailed routing tracks. To connect

two such elements, a *stretched link* is created to maintain connectivity. The number and length of these stretched links are minimized by simple heuristics which choose a suitable track passing through one of the elements to be connected (Fig. 3.8). In the bottom-up process, most of the nets are implemented with many wires and some of them may not touch any of the higher-level cut-lines. In conducting the higher level routing, the wires not touching cut-lines are ignored so the data abstraction will be the same as that in the top-down process. In our implementation, the global routing algorithm carries out all these transformations automatically without creating a new data structure to store temporary routing results.

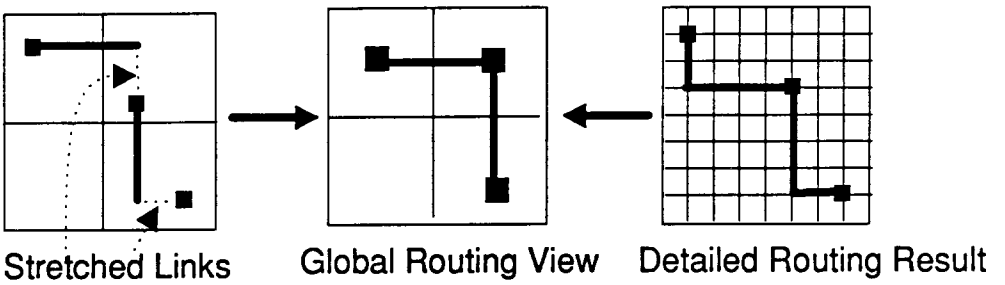


Fig. 3.8 The use of stretched links.

With this unified data structure, no extra transformation is needed to map the global routing results to a form that can be used by the detailed routing. The detailed router only has to convert all the stretched links into actual wires before it starts.

3.3 Routing Algorithms

The router uses three basic routing algorithms to complete the task, *the global construction algorithm, the global rerouting algorithm,* and *the detailed rerouting algorithm.* For a given routing problem, the router partitions the region with alternating horizontal and vertical cuts. When the router partitions a region of N cells into $2 \times N$ cells, the *global construction algorithm* builds an initial solution with near-optimal wire length. In such a strip of $2 \times N$ cells, each *edge* (i.e. the boundary between the cells) has a *capacity* measure that is the number of free detailed routing tracks crossing it. The global construction algorithm ignores these capacity constraints, but the *global rerouting algorithm* then tries to rearrange the wires so that there is no *overloaded edge* whose capacity is lower than the number of wires crossing it.

If the global rerouting algorithm fails to resolve all the overloaded edges, the router backtracks to higher levels, using the global rerouting algorithm to ease the congestion. After these congestion problems have been resolved, the router re-enters the top-down process. When this top-down process reaches level (3,3) successfully, the *detailed rerouting*

algorithm takes over to try to assign every wire to a conflict-free detailed routing track. The router will backtrack to higher levels if the detailed rerouting algorithm cannot resolve all the conflicts. The overall algorithm of the router is outlined in Fig. 3.9

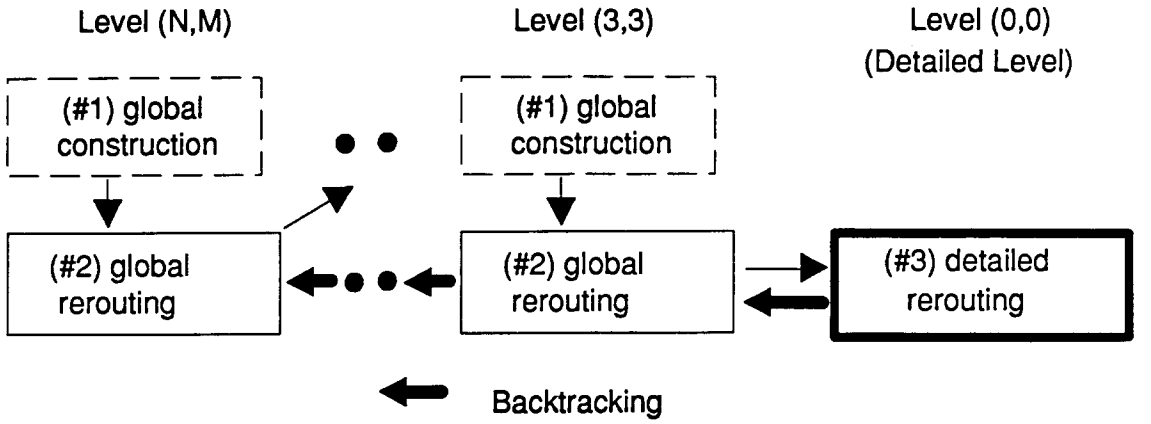


Fig. 3.9 The use of the three routing algorithms in completing a routing task.

These three algorithms as well as the overall control strategy will be explained in detail in the following sections. *Without loss of generality, the global construction algorithm and the global rerouting algorithm will be explained in the context of a horizontal* $2 \times N$ *strip with a horizontal cut-line.*

3.3.1 Global Construction Algorithm

When a horizontal strip of N cells is partitioned into $2 \times N$ cells, the router first assigns all the pins and vertical wires into one or both of the horizontal slices based on their absolute y-positions. Then a pattern router replaces each horizontal wire segment crossing several cells by one or more new horizontal segments assigned to one of the two slices and by the necessary vertical segments to keep the net connected (Fig. 3.10). The pattern router tries to minimize the number of additional vertical wire segments crossing the cut line while ignoring the capacity constraints on the edges between the cells.

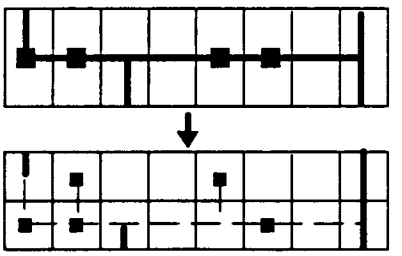


Fig. 3.10 Partitioning a horizontal $1 \times N$ strip into a $2 \times N$ strip.

For each horizontal wire segment, the pins and vertical segments to be connected may appear in a column in three possible constellations: in the lower cell only, in the upper cell only, or in both cells (Fig. 3.10). Among the 3×3 possible cases of connecting elements (pins and wires) in adjacent occupied columns, six have only one minimum-cost solution (Fig. 3.11(a)). For the other three cases, the two possible solutions have the same number of vertical wire segments, and the connecting wires are labeled as *switchable* (Fig. 3.11(b)). For a multiple-pin net, adjacent switchable segments are grouped into a long segment to be considered simultaneously. This long segment will be assigned to the lower (upper) slice if there are more elements in the lower (upper) slice to be connected (Fig. 3.11(c)). If there are an equal number of pins in the lower and upper slices, the segment will be assigned randomly to one of the two positions.

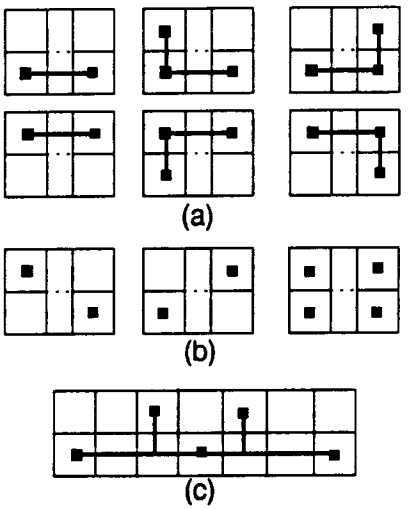


Fig. 3.11 (a) The six cases with one minimum-cost solution. (b) The three cases with switchable wire segments. (c) Four adjacent switchable segments are grouped into one segment and assigned to the lower slice because there is one more pin in the lower slice.

3.3.2 Global Rerouting Algorithm

The initial solution constructed by the pattern router may have many overloaded edges. The rerouting algorithm tries to reduce the number of overloaded edges with two connectivity-preserving rerouting routines: *wire-push* and *maze-routing*. The wire-push operation moves a horizontal wire segment from one slice to the other in the $2 \times N$ strip (Fig. 3.12(a)), creating jogs only at columns that contain elements (pins or wires) connected to this net. To create a path with arbitrary jogs, a simple maze-router optimized for the $2 \times N$ search space is used(Fig. 3.12(b)).

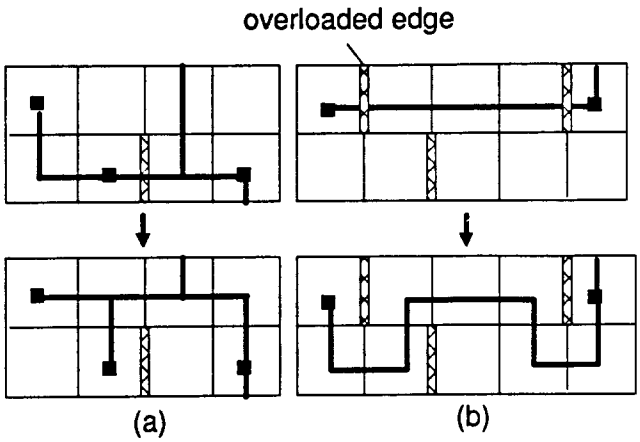


Fig. 3.12 Rerouting operations in 2 × *N* strips: (a) wire-push (b) maze-routing.

To reduce the number of overloaded edges, the wire-push and maze-routing operations are selected with a *greedy strategy* based on the cost of a path. The overall rerouting algorithm can be outlined in the following C-like pseudo-code:

```
For_all_overloaded_edge (E) {
  do {
      For_all_wires_crossing_edge_E (w);
          evaluate_push_cost (w);
      }
      w = wire_with_minimum_push_cost;
      C = push_cost_of (w);
      if ( C < 0 ) {
          push_wire (w);
      }
  } while ( C < 0 && edge_is_overloaded (E);
  if ( edge_is_overloaded (E) {
      For_all_wire_crossing_edge_E (w) {
          if ( try_maze_routing(w) == OK ) {
              if ( ! edge_is_overloaded (E) {
                  break;
              }
          }
      }
  }
}
```

In this process, the procedure evaluate_push_cost computes the difference between the *configuration costs* of the two possible positions. The configuration cost for a net is computed based on the following formula:

$$\text{Configuration\_Cost} = \sum_{e} \text{cost\_on\_edge (e)};$$

where

$\sum_{e}$: all edges touched by wires of the net;

cost_on_edge(e) $= \frac{1}{h}$ estimated_length(e) + congestion_measure(e) ;

estimated_length(e) = distance between the centers of the two regions separated by the edge;

$h = \frac{1}{2}$ (perimeter of the routing region);

congestion_measure(e) = 1 if e is overloaded when the net is implemented ;
= 0 otherwise.

This cost function consists of two parts, *wire length cost* and *congestion measure*. The wire length cost is scaled so that it is always less than the congestion measure if the edge is overloaded. The congestion measure is determined by whether the edge is overloaded or not instead of by the difference between the number of crossing wires and the capacity of the edge. Such a measure can prevent the router from simply re-distributing the congestion without finding a feasible solution free of overloaded edges. For example, if edge *A* is overloaded with one wire and edge *B* is overloaded with five wires, moving two wires from edge *B* to edge *A* can distribute the congestion evenly but won't solve the problems. With the binary-valued congestion measure, a net crossing edge *A* or edge *B* has the same configuration cost so no wires will be moved from edge *B* to edge *A*. Instead, other moves will be tried or backtracking will be initiated to solve the problem.

Based on such a congestion measure, minimizing the total configuration costs of all nets can minimize the number of overloaded edges. This rerouting process repeats until no edge is overloaded or no other advantageous moves can be found. If some edges are still overloaded, backtracking to the next higher level is initiated with proper congestion information passed along.

In the bottom-up backtracking process, this cost function is modified slightly to take into account the congestion information passed from the lower level routing. The modification will be explained in more detail later.

3.3.3 Detailed Rerouting Algorithm

The global construction and rerouting algorithms put all the wires on the detailed routing grid while using some stretched links to maintain connectivity. Because wires are assigned to the grid lines with simple heuristics, there are usually many *conflicts* caused by wires competing for the same track or by wires routed over obstacles. The detailed router first

replaces all the stretched links with explicit wires and vias, using just one wire or two wires forming an L-shape for each stretched link. Then it tries to resolve the conflicts with two rerouting operations: *wire-push* and *maze-routing*. These two operations are essentially the same as those used in the global rerouting algorithm, except that these two operations work in a rectangular region on the detailed routing grid rather than on a $2 \times N$ strip. These operations are also similar to those used in [TS88] and are illustrated in Fig. 3.13.

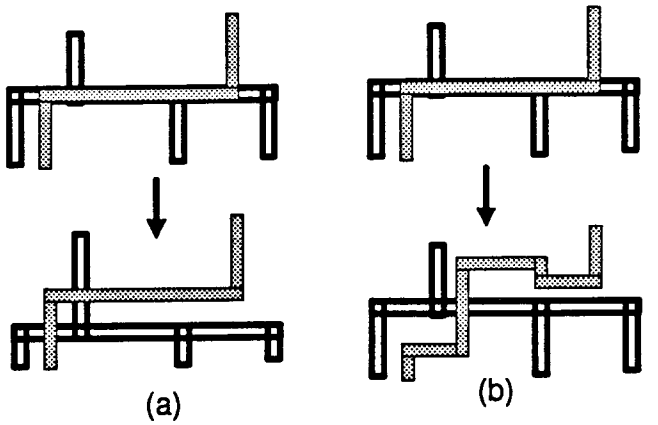


Fig. 3.13 Rerouting primitives for detailed routing: (a) wire-push (b) maze-routing. The patterns of the wire segments are used to distinguish different nets but not different layers. Normally, different layers are used for wires in different orientations.

Similar to the global rerouting algorithm, the detailed rerouting algorithm tries to reduce the number of conflicts by selecting rerouting operations based on a greedy strategy. The greedy algorithm can be described in the following C-like pseudo code:

```
For_all_conflicts (w1, w2 ){
    evaluate_push_cost (w1);
    evaluate_push_cost (w2);
    w = wire_with_minimum_push_cost;
    C = push_cost_of (w);
    if ( C ≤ 0 ) {
        push_wire (w);
    } else {
        if ( try_maze_route (w1) == FAIL ) {
            try_maze_route (w2) ;
        }
    }
}
```

In this procedure, w_1 and w_2 are normally two wires created by the router. Sometimes one of them may be a fixed obstacle, then only the actual wire will be tried for rerouting operations. The procedure evaluate_push_cost evaluates the neighboring 16 detailed routing tracks, eight on each side, of the wire under consideration. This range is set because the global routing stops at level (3,3) and leaves partitions with eight grid lines on each side. For each of the 16 possible positions, the cost is computed based on the following formula:

$$\text{push\_cost}(w,p) = \sum_{o} \text{conflict\_cost}(w,o) ;$$

where

$\sum_{o}$: for every obstacle o on position p;

$$\text{conflict\_cost}(w,o) = h \quad \text{if } o \text{ is fixed obstacle;}$$
$$= \text{length}(o) + \text{history\_cost}(o) \quad \text{otherwise;}$$
$$\text{history\_cost}(o) = \text{push\_count}(o) * \frac{h}{10} ;$$
$$h = \frac{1}{4} (\text{perimeter of the routing region}) ;$$

In computing the push-cost, the *conflict cost* is primarily an indicator for the difficulty of rerouting the wire that causes the conflict. For wires created by the router, the conflict cost is computed based on the length of the wire, because longer wires are usually more difficult to reroute. Fixed obstacles cannot be rerouted, thus they are assigned a cost higher than most movable wires.

This greedy rerouting algorithm works like a depth-first search algorithm, trying to find a sequence of moves to resolve each conflict. To prevent the rerouting process from working on a few short wires repeatedly, every wire carries a *push-count*, which is increased by one whenever the wire is pushed. Wires with higher push-counts will be less likely to be pushed again because it is more expensive to have conflicts with these wires. The weight of this history cost is set to make a wire look like a fixed obstacle if the wire has been pushed several times. The value 10 has been chosen based on some simple experiments. With this push-count, the greedy algorithm can find a very long sequence of moves without getting trapped into a loop. However, it is usually very expensive to accumulate many local moves to resolve a conflict. Therefore, the router only tries to find a rerouting sequence of four moves for each conflict. That is, after the router works on all conflicts for four times, the router backtracks to higher levels and uses the global rerouting algorithm to make more dramatic changes.

3.3.4 Control Strategy and Congestion Data Structure

With the three basic routing algorithms, the following C-like pseudo-code outlines how the router selects the cut orientation and moves from one level to another:

```
    ori  = horizontal ;
    ori2 = vertical ;
    mode[ori]  = top_down;
    mode[ori2] = top_down;
    level [horizontal] = log2 ( horizontal_size ) + 1 ;
    level[vertical] = log2 (vertical_size ) + 1 ;
again:
    while ( level[ori]  >= detailed_level && level[ori2] >= detailed_level) {
        if (level[ori] > level[ori2]) {
            change_ori = YES;
        } else if ( mode[ori2] == bottom_up && mode[ori] == top_down ) {
            change_ori = YES;
        } else {
            change_ori = NO ;
        }
        if ( change_ori ) swap_integer ( &ori, &ori2) ;
        if ( mode[ori] == bottom_up) {
            if ( ! change_ori) level[ori] ++ ;
        } else {
            level[ori] -- ;
        }
        route_nx2 (ori, shift_NO);
        if ( check_congestion(ori) == FAIL) {
            route_nx2 (ori, shift_YES) ;
        }
        if ( check_congestion (ori) == FAIL ) {
            mode[ori] = bottom_up ;
        } else {
            mode[ori] = top_down ;
        }
        if ( check_congestion (ori2) == FAIL ) {
            mode[ori] = bottom_up ;
        }
    }
    detail_route();
    if ( check_congestion (horizontal) == FAIL ) {
        mode[horizontal] = bottom_up ;
    }   if ( check_congestion (vertical) == FAIL ) {
        mode[vertical] = bottom_up ;
    }
    if ( mode[vertical] == bottom_up || mode[horizontal] == bottom_up )
        goto again;
```

In this procedure, the variable ori determines the cut orientation. Let h = level[horizontal] and v = level[vertical], the procedure route_nx2(horizontal, shift_NO) makes horizontal cuts at $2^h+n\times 2^{h+1}$, n=0,1,....., to move from level (h+1,v) to level (h,v). When it is called as route_nx2(horizontal, shift_YES), it makes horizontal cuts at $n\times 2^{h+1}$, n=0,1,..... This is equivalent to shifting the $2\times N$ routing frame by half of the size of the original partitions to break the artificial boundaries created by higher-level cut lines.

The first time a slice is cut, the global construction algorithm is called first, followed by the global rerouting algorithm. Afterwards, only the rerouting algorithm is used. If the rerouting algorithm cannot resolve congestion in the *horizontal* (vertical) direction, the router increases the *horizontal* (vertical) level so it can search in a larger region without increasing the complexity. Fig. 3.14 shows an example of the change of levels in routing a macro-cell example.

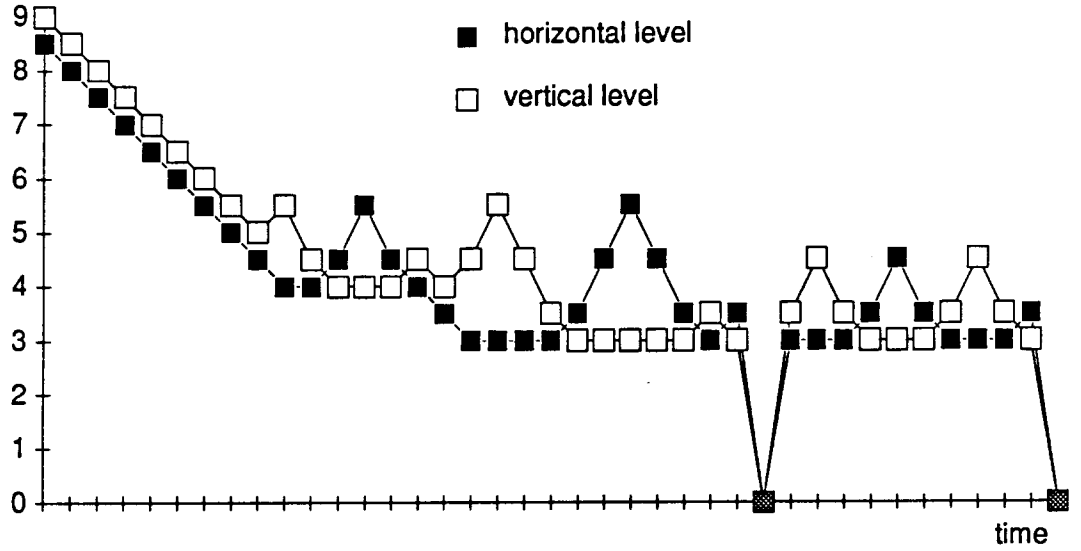


Fig. 3.14 The change of the levels in routing a macro-cell example. To make the picture more informative, the level of the cut-orientation is increased by 0.5. For example, level (3.5, 3) means that the router is moving from level (4,3) to level (3,3). The detailed routing takes place when the level is (0,0).

To record the congestion information, *congestion elements* are created, which serve as the main mechanism for congestion information exchange between levels. When the global rerouting algorithm fails to resolve an overloaded edge, it creates congestion elements to cover the overloaded edge. If a *vertical* (horizontal) edge with capacity C in a $2\times N$ map has W *horizontal* (vertical) wire segments passing through ($W > C$), a *horizontal* (vertical) congestion element is created with an *overload count* (W–C). The congestion element is labeled as *active* initially. After W–C wires are moved away from the congestion element, the congestion element is then labeled as *passive*. Similarly, when the detailed rerouting

algorithm fails to resolve a conflict between two elements on a detailed routing track, a corresponding congestion element is created to cover the section of the track on which they overlap (Fig. 3.15). The overload count for such a congestion element is 1. If more than two elements occupy the same section of a track, a congestion element will be generated for each pair of elements.

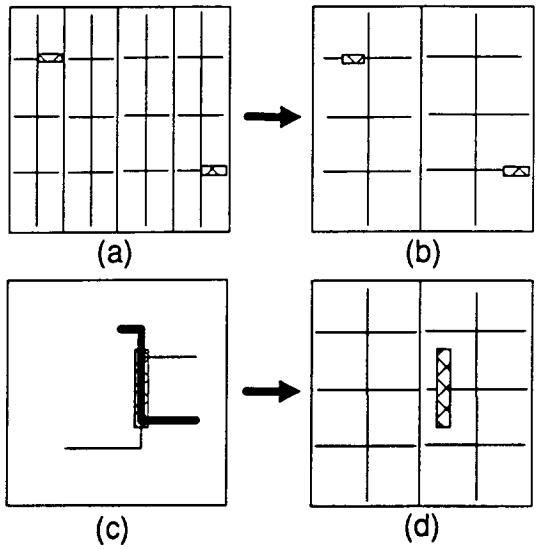


Fig. 3.15 (a) Two vertical congestion elements are created to cover overloaded edges; and (b) they change the congestion measure of the edges at next higher level. (c) One vertical congestion element is created to cover two overlapping vertical edges; and (d) it changes the congestion measure in global routing.

With these congestion elements, the procedure check_congestion simply checks the presence of congestion elements. If there are active congestion elements in the direction under consideration, the procedure returns the value FAIL and will cause the router to shift the $2 \times N$ routing frame or enter the bottom_up mode. When the router moves up to higher levels, these congestion elements change the *congestion measure* (see page 54) of the edges touching them (Fig. 3.15). The congestion measure in computing the configuration cost for a net is modified according to the following formula:

congestion_measure(e) = 1 if e is overloaded when the net is implemented, or there are active congestion elements on e;
= 1.5 if there are passive congestion elements on e that once caused wires of the net to move;
= 0 otherwise.

This new cost function treats edges with active congestion elements as overloaded edges even though they may not be overloaded. Therefore, wires will be moved away from these

congestion elements, which will then be labeled as passive. After the move, the edge will not have any active congestion elements and wires may move back to the edge. To prevent the router from moving wires back to their original positions, each congestion element records the nets that are pushed away because of its presence and generates a high congestion cost (1.5) when the router tries to push back any wires of the recorded nets. Fig. 3.16 shows how the congestion elements are used in a backtacking process. These passive congestion elements are removed when the router starts the detailed rerouting algorithm which does not use any information from these congestion elements at all.

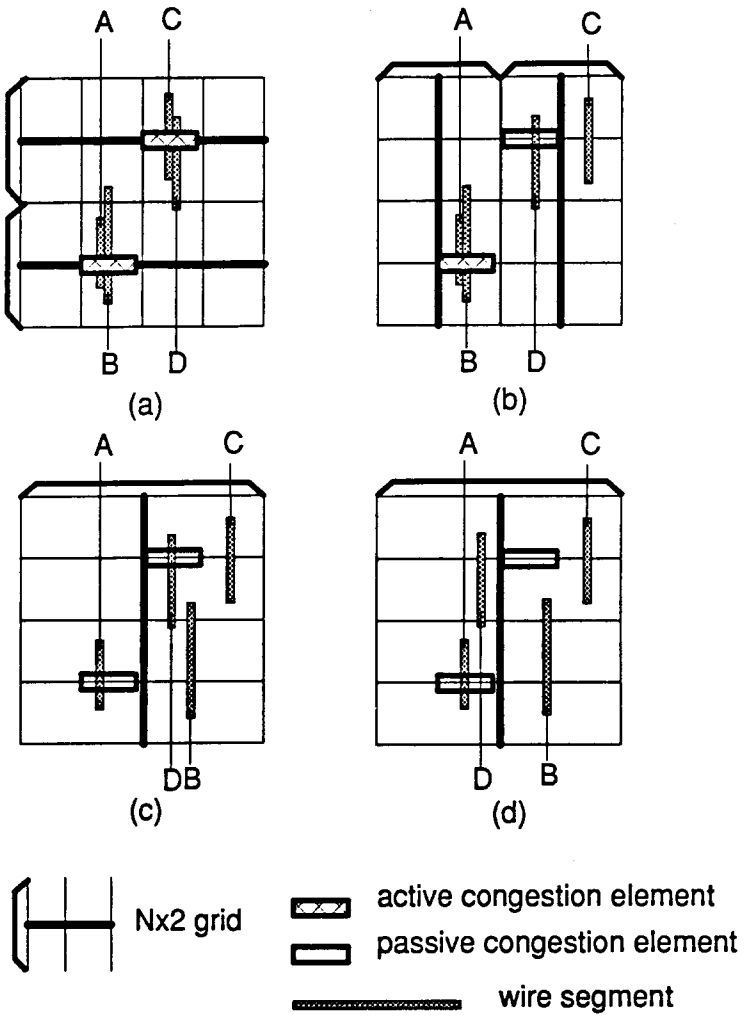


Fig. 3.16 The use of congestion elements in backtracking. (a) $2 \times N$ routing with horizontal cuts. Two vertical congestion elements are created. (b) Backtracking to the previous level with vertical cuts. One congestion element is resolved by pushing wire segment C away. (c) Backtracking to next higher level. No edge is overloaded at this level. The congestion element causes wire segment B to be pushed away. (d) More rerouting operations at the same level. Wire segment D can be pushed to the left but wire segment B won't because the passive congestion element causes the move of B. The vertical wire segments in these figures should be connected by horizontal wire segments, which are not shown here for simplicity.

3.4 Results

The main application targets of this router are large area routing problems. However, very few results have been published for this kind of problems and there are no well-established benchmarks. Therefore, a series of synthetic area routing problems have been created with randomly generated net lists. These random problems are made to be reasonable close

to the routing problems in actual large gate array designs. First, a set of small cells of the same size (twelve by six tracks) are created. For each problem, a 2-D array of cells is formed from these small cells and a list of nets is generated to connect pins of these small cells. Among the generated nets, most of them contain only two pins, while only a few nets contain more than ten pins. In addition, pins of the same nets are usually kept close to each other as would be the case in most well-placed layouts.

An important issue in benchmarking the routers is the difficulty of the routing examples. This is especially important when the rip-up and reroute process is involved, As a rough estimate, these randomly generated routing problems are characterized by their average and maximum *cut-densities*. The cut-density is defined, for each cut line crossing the whole routing region, as the percentage of occupied tracks on the cut line. To make these examples of approximately equal difficulty, the maximum cut density for each problem is adjusted to be about the same by inserting additional space between rows or columns of cells. This additional space is distributed into channels between rows and columns based on a simple estimate of the local congestion. Table 3.1 shows the characteristics of these random examples with the cut densities measured from the actual routing results from our router. Fig. 3.17(a) shows the layout of one of the examples.

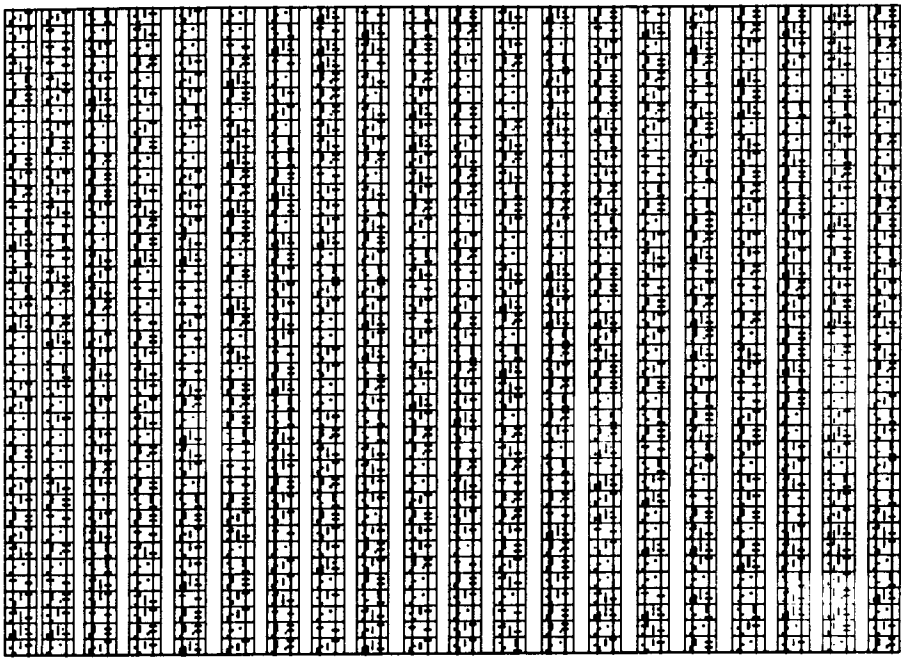
Table 3.2 summarizes the results on these routing problems generated by our multi-level hierarchical router, and Fig. 3.17(b) shows one of the resulting layouts. Table 3.2 also shows the results obtained by two other area routers, MIGHTY [Shi87], a detailed router designed for small-sized problem, and CODAR [TS88], a detailed router with some global routing capability. Because MIGHTY cannot handle large routing problems, many medium-size problems have been generated for MIGHTY and the example "r13" is the largest one that MIGHTY can complete.

Name	Area	# Nets	Cut-Density (Routed)			
			horizontal		vertical	
			average	maximum	average	maximum
r5	60x60	62	56	72	52	82
r10	138x130	232	62	77	49	72
r13	203x209	409	53	66	43	60
r20	338x240	822	54	68	45	64
r30	595x360	1878	53	68	43	60
r40	854x480	3327	54	69	43	62
r50	1129x600	5189	54	69	44	62
r60	1532x720	7453	53	65	43	61

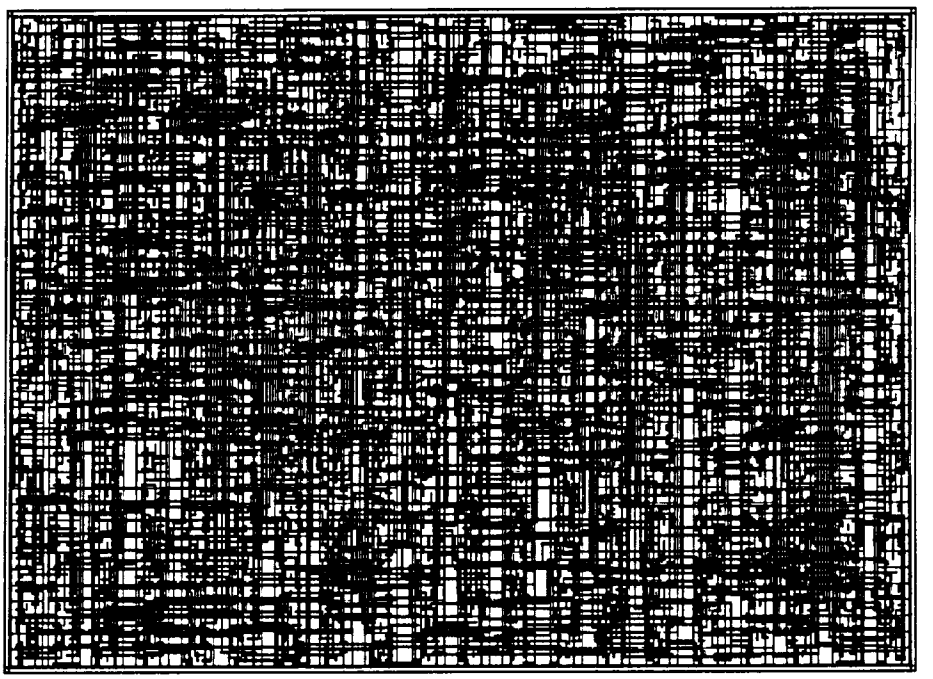
Table 3.1 Characteristics of the randomly generated area routing problems. The cut-density is measured in percentage points from actual routing results generated by our multi-level hierarchical router.

Name	Mighty		Codar		**M-L Hier. Router**		
	CPU time	wire length	CPU time	wire length	CPU time	wire length	# iterations
r5	18.3	2609	4.4	2648	3.6	2692	1
r10	268.6	14072	24.7	14490	17.1	14830	1
r13	1044.8	29049	45.8	29713	28.1	30493	1
r20	fails	-	151.9	61373	90.1	61924	2
r30	fails	-	585.5	164246	247.8	163673	2
r40	fails	-	1639.3	325086	631.8	324238	2
r50	fails	-	3896.3	543116	1353.2	545322	2
r60	fails	-	9054.8	877170	2503.8	883513	2

Table 3.2 Results on the randomly generated area-routing examples. The wire-lengths are measured in units of the detailed routing grids. The CPU-times are measured in seconds on a SUN SPARC-1. Our multi-level hierarchical router (M-L Hier. Router) moves between global and detailed routing algorithms and the "# iterations" is the number of complete global-detailed loops used to complete the routing task.



(a)



(b)

Fig. 3.17 (a) The area-routing problem r20. (b) The routing result.

Table 3.2 shows that our multi-level hierarchical router is faster than the other two on these routing problems. For a clearer analysis of the complexity of these routing approaches, Fig. 3.18 shows the CPU-time v.s. the total wire length for the three routers. Among the three routers, MIGHTY is designed for small detailed routing problems, and these examples are simply far beyond its original design goal. However, MIGHTY's data is a good approximation of the routing complexity without any hierarchical approach. With some integrated global routing capability, CODAR can handle medium- to large-sized problems. However, its two-level hierarchy is still not enough for large-sized problems and the complexity is growing faster as the problems are getting larger. With the multi-level hierarchy, the complexity of our router is still kept to be only slightly higher than linear complexity, even for large routing problems.

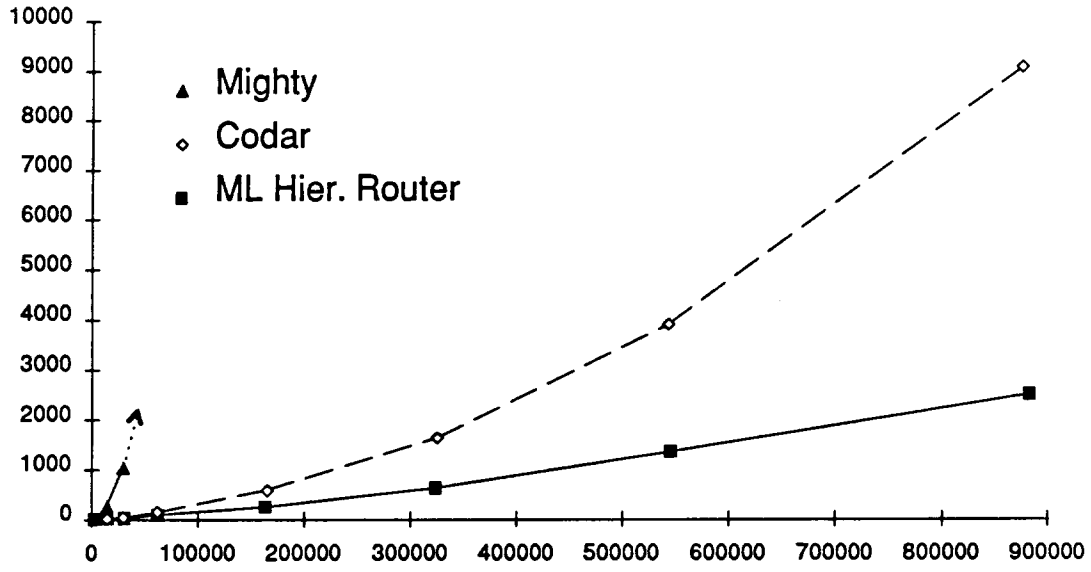


Fig. 3.18 The CPU-time v.s. the wire length for area routing problems.

3.5 Summary

A new area-routing approach has been presented. The router combines hierarchical routing with rip-up and reroute techniques, accomplishing most of the routing task with modification-based rerouting operations at various levels of abstraction. Therefore, the router can complete difficult routing problems. Because the routing region is partitioned recursively into a multi-level hierarchy, the router runs efficiently with a very small search space at each level. A unified data structure permits the router to switch easily between levels, so that it can employ efficient rerouting operations at higher-levels to resolve congestion. Considerable speed-up has been shown over other approaches to area routing.

Chapter 4
Integrated Placement and Routing

4.1 Routing and Placement Adjustment

In most placement and routing systems, the cells are first placed and then routed. Most placement algorithms optimize the wire length and find a good placement that defines the relative positions between cells. To enable the router to complete all the nets without any conflicts, a *placement adjustment* phase is required to allocate enough space between cells. The simplest placement adjustment approach is to move the cells based on the actual routing result. This approach is normally used in the standard-cell designs because the wires are restricted to the channels between rows of cells and the height of each channel can be adjusted easily without disturbing the routing results of other channels. Usually, the channels are routed one by one and the placement adjustment is in the vertical direction only.

The problem is much more difficult for macro-cell layouts when the routing channels can be perpendicular to each other and the cells need to be moved in both horizontal and vertical directions. Without a slicing structure, which is usually too restrictive and causes too much wasted space, it is normally impossible to find a proper order of channels such that the channels routed later can be adjusted without disturbing the channels routed earlier. In [DAK85], L-shaped channels are introduced to find such a linear order of the routing channels. However, it is very difficult to adjust the L-shaped channels with only manhattan wires [Che87]. Even though some algorithms have been proposed to minimize the number of L-shaped channels [CW91], it is still difficult to eliminate all the L-shaped channels. The problem becomes even more difficult when wires can go over the cells and no channel structures can be defined. In such an area routing environment, it is normally impossible to adjust the space between cells without disturbing the wiring generated earlier.

When it is difficult to adjust the placement based on the actual routing result as in the macro-cell layouts or in an area-routing environment, a common approach is to allocate routing space in advance then route the whole chip with cells in fixed positions. Usually, the space is allocated based on some statistical analysis or rough routing results. However, it is normally impossible to predict the space requirements precisely without actually conducting the routing task. Allocating too little space, may cause the router to fail, and requires that the whole chip be routed again with a new placement. To avoid this time-consuming process, most systems allocate excess space initially which usually results in larger chips. Sometimes, enough routing space is allocated but it is not distributed properly,

resulting in local congestion. This may force the router to detour some nets, thus increasing the delay of certain signals.

Routing with fixed placements usually results in wasted space and/or conflicts in congested areas. A typical remedy to avoid routing the whole problem again after adjusting the placement is to use symbolic layout compaction [HP79] to obtain a dense layout while resolving all the design-rule violations. This approach simplifies the placement adjustment and routing problems but shifts the burden to the compactor. Unfortunately, it is usually very difficult for the compactor to get a dense layout from a routing result generated with too much routing space. As shown in Fig. 4.1, when there is plenty of routing space, the router may produce two different types of results: some can be compacted into a very dense layout with a simple 1-D compactor but others can not. To generate a dense layout from a layout like Fig. 4.1(a), a sophisticated compactor that can change the relative positions of elements is required. In [Hoj90], a layout modification phase that employs rerouting techniques has been proposed as a pre-processing step of the compactor. However, such a system has very high time complexity because it has to deal with gridless routing and compaction at the same time. Thus it can only handle small problems effectively.

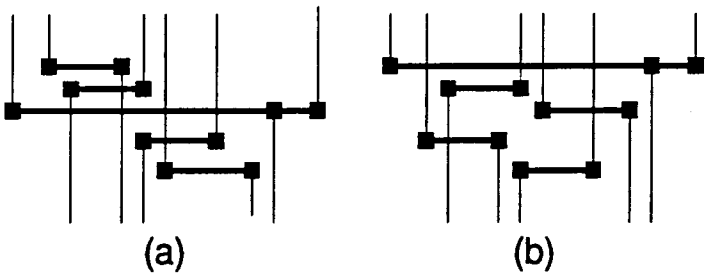


Fig. 4.1 The routing result that is (a) difficult to compact and (b) easy to compact in the vertical direction.

To avoid relying on a compactor to minimize the wasted space, a tight integration between the placement adjustment and routing phases seems to be the only solution. A straightforward integration would be to iterate between routing and placement adjustment phases. Because routing is usually a very time-consuming process, it will be very expensive to remove all the wires while adjusting the placement and then to route the whole problem all over again. The only efficient approach would be to carry out both the routing and placement adjustment phases incrementally. Incremental modification of the routing is essentially a rip-up and reroute process, which is usually also an expensive process. To make such an integrated approach successful, the rip-up and reroute operations have to be carried out efficiently and the placement adjustment phase also has to minimize the rip-up and reroute operations required after the cells are moved. Yet another key issue of such an integrated scheme is how to adjust the placement based on the routing results. It seems that

these problems have not previously been studied in depth, and the author is not aware of any existing system that uses such an integrated placement and routing approach.

The router introduced in the previous chapter is based on efficient rip-up and reroute operations. These efficient rip-up and reroute operations have made such an integrated placement and routing scheme feasible. Based on the router, a new placement adjustment algorithm has been developed specifically to work with the router. By iterating between the routing and placement adjustment phases, a dense layout can be achieved efficiently. The basic ideas of this integrated approach will be highlighted in the next section, followed by more detailed descriptions of the data structure and algorithms.

4.2 Basic Idea

The input to the integrated placement and routing system is assumed to be a legal placement like those obtained by the algorithms shown in Chapter 2. That is, there are no overlaps between cells but there is also no additional routing space allocated between cells. With all the cells being movable, the goal is to implement all the nets without any conflicts in the smallest area. As shown in the previous section, it is difficult to compact a sparse routing result into a dense layout. Therefore, very little space will be allocated initially between the cells to force the router to try to find the best possible solution in the minimum amount of the space. Additional space is added only where it is necessary.

From a placement without enough routing space, the router first connects all the nets while ignoring some of the congested regions and conflicts between nets. Then based on the congestion measure derived from such an actual routing result, the placement is adjusted by inserting space between the cells to ease the congestion. When the cells are moved during this placement adjustment phase, wires will be dragged properly to maintain most of the existing routing result. Then the router will try to complete all the nets again using the fast rerouting operations. If the router still cannot complete the routing task, the placement can be adjusted again. Such an adjustment-routing loop can be repeated until all the nets are implemented without any conflicts.

The router used in such a loop is essentially the hierarchical area router introduced in the previous chapter. However, some non-trivial modifications are required to allow the router to work in different modes. Section 4.3 shows how the area router is enhanced. Then Section 4.4 shows how the placement adjustment algorithm moves the cells based on the routing results while moving and stretching wires to preserve most of the routing result. Finally, the overall algorithm will be described and various experiment results will be presented.

4.3 Area Routing

When the system iterates between the placement adjustment and routing phases, the area router is used in two different modes. When the router is called the first time, very little routing space is allocated between cells. In this situation, the router has to ignore the congestion that can be resolved by the placement adjustment phase. It will be waste of time for the router to try very hard to find a solution without any conflicts because additional space can be added to ease the congestion. In fact, in most cases, it is impossible for the router to complete the task without adjusting the placement. In the second situation after the placement is adjusted based on the congestion analysis, the router has to try to minimize the number of conflicts. However, the placement is still adjustable so the router can still use the placement adjustment algorithm to fix some locally congested areas that are difficult to route.

Based on this idea, the major enhancement required for the router is the routines for evaluating the congestion situation. When the placement is adjustable, there are basically two types of congestions: one type can be resolved by adding space between cells, but the other can not. Basically, a congested region that falls between moveable cells can be resolved by adjusting the placement while a region that is congested because it is totally covered by a cell has to be resolved by the router.

As presented in Chapter 3, the hierarchical router works on a shared routing database with three routing algorithms - global construction, global rerouting, and detailed rerouting algorithms. These algorithms are presented as the tools for solving routing problems with enough routing space allocated among the cells. It turns out that, to allow the router to evaluate the congestion situation without trying to complete 100% of the nets, only the global rerouting algorithm requires some enhancement. The global construction algorithm ignores all the capacity constraints, and the detailed rerouting algorithm allows the presence of conflicts, so these two algorithms don't need any modifications at all.

As shown in Section 3.3.2, the global rerouting algorithm makes decisions based on a configuration cost that is computed from the congestion measure of the edges of the $2 \times N$ routing frame. The congestion measure is derived from the number of free tracks and wires passing through the edge under consideration. When the router is called with the option to ignore the adjustable congested regions, the congestion measure of an edge will be zero if the edge touches any space between movable cells. If the edge is completely covered by a cell, the normal congestion measure will then be used. With such a differentiated congestion measure, the edges touching the space that can be adjusted will never be treated as overloaded. Congestion elements with proper overload counts are still created for these edges, but these congestion elements will be marked differently and ignored during the

routing process. These elements are used in the placement adjustment phase to figure out the capacity requirements between the cells. Fig. 4.2 shows the two different types of edges of the $2 \times N$ routing frame in the global rerouting algorithm.

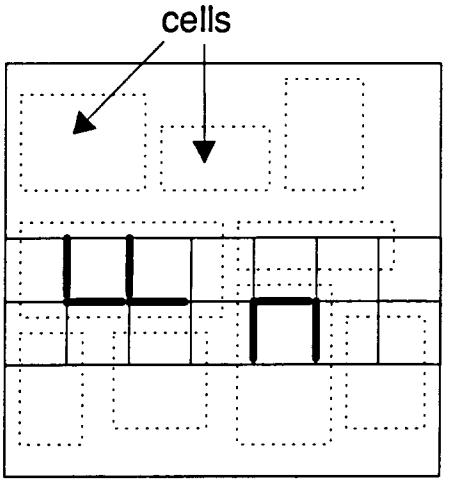


Fig. 4.2 Two different types of edges in the $2 \times N$ routing frame. The congestion on the **bold** edges (i.e. when the edges are overloaded) cannot be resolved by placement adjustment. The congestion on the other edges can be resolved by moving cells.

4.4 Placement Adjustment

When the router fails to complete all the nets for the given placement, the placement adjustment phase has to move cells around to create room for the router. The task of the placement adjustment phase is to take the existing routing result, including wires and congestion elements, and then generate a new placement while preserving the existing routing results as much as possible. The next section first shows how the required space is derived from the congestion analysis from the existing wiring. Then the algorithm to move cells with all the attached wires will be presented.

4.4.1 From Congestion to Space Requirement

The first step of the placement adjustment phase is to estimate the required space between adjacent cells. These space requirements represent the spacing constraints between the adjacent cells, which can then be enforced with a 1-D compaction/spacing algorithm. The RULD-graph and the associated 1-D compaction algorithm introduced in Chapter 2 are used for this purpose. In fact, when all the cells are tightly packed together, the traditional horizontal and vertical constraint graphs [HP79] are almost the same as the RULD-graph and could be used instead. However, the RULD-graph is used because it also can handle inputs that is not yet densely packed. As shown in Chapter 2, a traditional 1-D compaction

approach can easily destroy the relative positions between cells if the cells are not evenly distributed.

In the RULD-graph, adjacent cells are connected by edges. When the RULD-graph is used to resolve overlaps among cells during the placement process, the *explicit constraints* derived from these edges represent the spacing constraints that require the two connected cells to be separated by a distance greater than or equal to zero. To take into account the additional routing space between the cells, the RULD-graph is enhanced so that the edge can also represent spacing constraints greater than zero. The 1-D compaction algorithm that works with the RULD-graph can take this additional spacing information into account with only minor modifications.

The key problem is now how to convert the congestion information into spacing constraints for the edges in the RULD-graph. The congestion information found in the routing process is represented by the congestion elements distributed in the routing region. For each edge in the RULD-graph, its spacing constraint is computed based on the congestion elements touching the *region covered by the edge*. For an edge between block A and B, the region covered by the edge is a rectangular region defined as:

$(\min(A_{xc},B_{xc}),\max(A_b,B_b))$ to $(\max(A_{xc},B_{xc}),\min(A_t,B_t))$ if edge is horizontal;

$(\max(A_l,B_l),\min(A_{yc},B_{yc}))$ to $(\min(A_r,B_r),\max(A_{yc},B_{yc}))$ if edge is vertical;

where

t: top, b: bottom, l: left, r: right,

xc: x-position of center,

yc: y-position of center.

The interpretation of these equations is shown in Fig. 4.3. A special case exists for these equations when $\max(A_b,B_b) > \min(A_t,B_t)$ for a horizontal edge or $\max(A_l,B_l) > \min(A_r,B_r)$ for a vertical edge (Fig. 4.3(c)). In these cases, the covered region of the edge will be an empty set and its spacing constraint won't be increased at all.

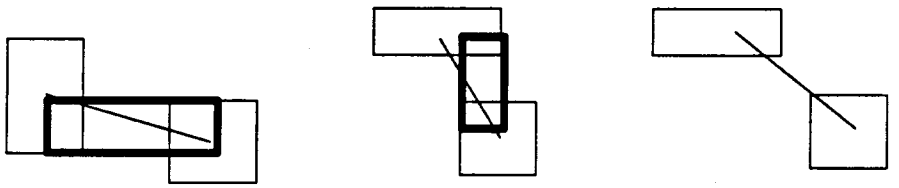


Fig. 4.3 The covered area (the bold rectangles) for (a) a horizontal edge and (b) a vertical edge; and (c) an edge whose covered area is empty.

In computing the required spacing constraint for each edge, the region covered by the edge is first computed. Then for all the congestion elements inside the covered region, the congestion element with the maximum overload count is located. The spacing constraint of the edge will then be increased by the overload count of this congestion element. The *maximum* overload count is used instead of the sum of overload counts of all the congestion elements because the congestions at different locations can sometimes be resolved simultaneously by increasing the spacing between cells by the maximum overload count (Fig. 4.4(a)). There are also some situations (Fig. 4.4(b)(c)) that the router may not be able to find a solution with the increased space. In these cases, the additional space will be added later in the following phases.

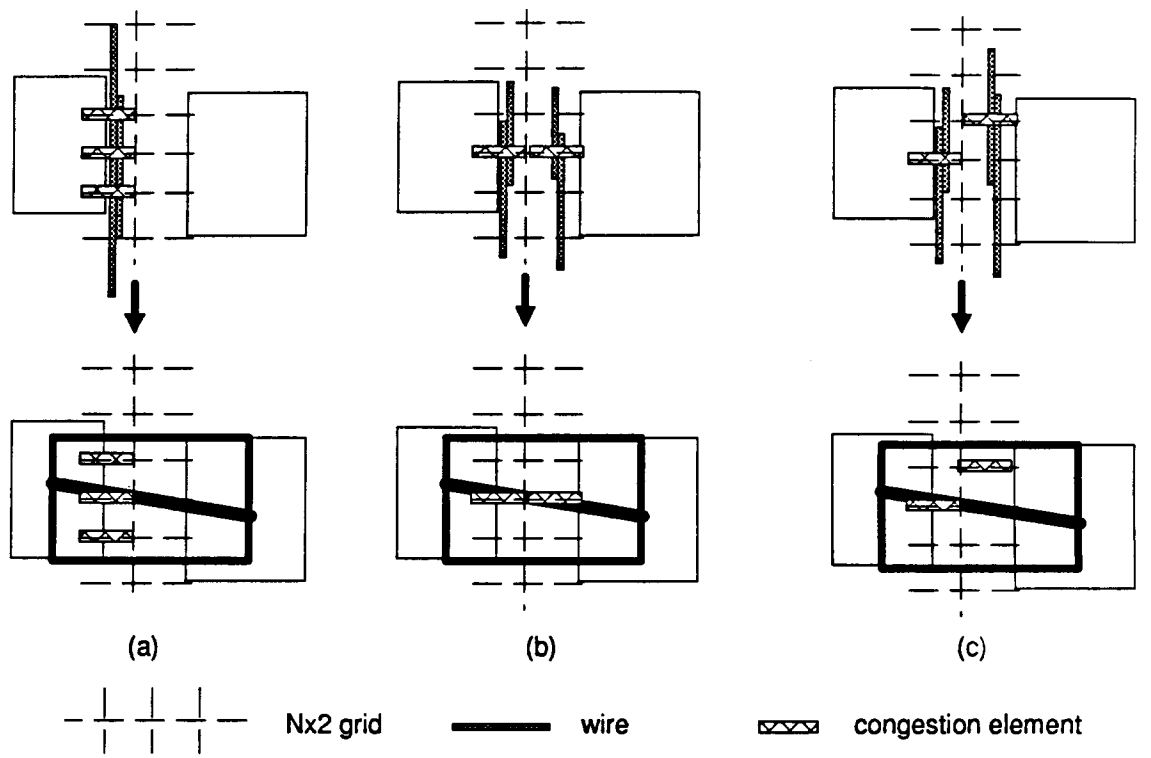


Fig. 4.4 Assuming the overload count of every congestion element shown be 1. Increasing the spacing between the two cells by the maximum overload count (=1), may resolve the congestion in case (a) but may not in case (b) or (c). The wires shown in these figures are possible candidates to be moved to resolve the congestion.

With these additional spacing constraints added to all the edges of the RULD-graph, the desired locations for all the cells can be computed quickly with the 1-D compaction algorithm introduced in Chapter 2.

4.4.2 Moving Cells

After the compactor has computed the desired locations of all the cells, the next step is to move each cell to its desired location. This would be a trivial task if there were no wires present. However, the wires connecting pins on different cells make the task very difficult, because when the cells are moved, some nets may get broken and some may get involved in new conflicts with the others (Fig. 4.5(a)(b)).

A possible solution for avoiding breaking nets and creating new conflicts is to use *layout compaction* techniques to push all the elements, including wires and cells. However, such a compaction process on the whole chip can be very time-consuming. Furthermore, because layout compactors normally preserve the relative positions between elements, the possible changes that can be achieved tend to be very limited. Usually some cells have to be moved, resulting in larger layout area (Fig. 4.5(c)). Since there will be a rerouting phase following the placement adjustment phase, these problems can be solved by the rerouting operations. In many situations, this approach can actually find a solution in a smaller space without pushing the affected cells (Fig. 4.5(d)).

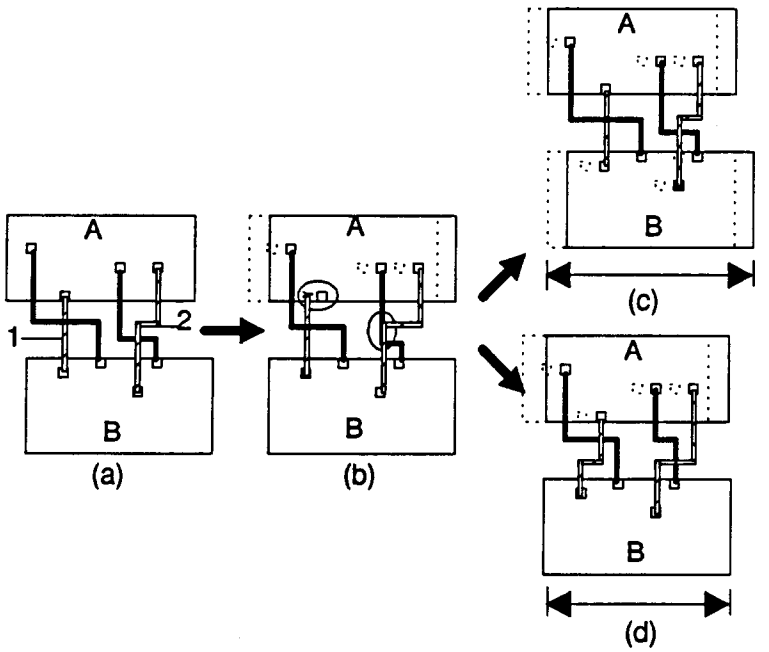


Fig. 4.5 (a) The original position of two cells. (b) When cell A is moved to the right, net-1 is broken and net-2 has a new conflict with another net. (c) Using compaction that maintains the original topology will push cell B to the right, increasing the total area. (d) Use of rerouting operations can find a solution without moving cell B. (In these figures, the patterns of the wire segments are used to distinguish different nets but not different layers. The wires in the same orientation are on the same layer.)

To use the router to reconnect the broken nets and resolve new conflicts, proper information needs to be passed to the router. This is achieved by modifying the routing data structure directly. Because the wiring data structure allows the presence of conflicts, there is no need to process the new conflicts caused by the cell moves. However, the connectivity information of the broken nets needs to be passed to the router. This connectivity information is recorded by adding a *stretched link* to connect the two separated parts of each broken net. As presented in Chapter 3, these stretched links are normal data elements in the routing data structure and no special treatment is required.

With such a general routing data structure, the task of moving cells is somewhat simplified. However, it is still very important to minimize the number of new conflicts and stretched links in order to simplify the task required in the following rerouting phase. This is achieved by a *wire-adjusting* algorithm that drags the wires with the moving cells. This wire-adjusting algorithm comprises two phases. In the first phase, wires are moved based on the moves of their neighboring cells to minimize the number of new conflicts. Then in the second phase, wires are stretched or shrunk to minimize the number of stretched links.

4.4.2.1 Moving Wires

In the first phase of the wire-adjusting algorithm, wires are moved based on the moves of their neighboring cells. The goal is to avoid breaking too many connections and to minimize the number of new conflicts. The basic idea is to divide the whole routing region into many non-overlapping partitions such that each partition contains exactly one cell. For all the wires touching a certain partition, the desired move will be the move of the cell in the partition. When a wire touches two or more partitions, the wire will have several desired moves.

Ideally, if every wire has only one desired move or all its desired moves are the same, there won't be any broken connections or new conflicts after making these desired moves. However, many wires may touch different partitions with different desired moves. The selection of the desired move is based on the move patterns of the cells. When the router fails to complete all the nets, the new positions of the cells are decided by the 1-D compactor using the new spacing constraints derived from the congestion analysis. Because the spacing constraints of all the edges in the RULD-graph can only increase or remain the same, and the 1-D compactor always pushes all cells to the lower-left corner, cells can only move upward or to the right with increased critical paths to the lower-left corner. To maintain the same move patterns for wires and cells to keep the original routing result, wires with several desired moves will select the minimum desired moves in both x– and y–directions. Because the desired moves for all the partitions are always positive, this selection process does not need to check the directions of the moves.

Based on the proposed idea, the remaining problem would be to partition the routing region such that each partition contains exact one cell. However, as shown in Fig. 2.15, such a partition cannot always be found with the RULD-graph. Even without the situation shown in Fig. 2.15, the non-zero spacing constraints also make it impossible to find such a partition in many cases. To solve the problem, a two-step process is taken to approximate such a partition.

In computing the desired moves for wires, a *surrounding region* is first computed for every cell. Using the attached edges in the corresponding RULD-graph, the surrounding region of a cell is computed based on the following equations and demonstrated in Fig. 4.6. For each cell, the desired moves for the wires that touch its surrounding region will be the move of the cell.

Surrounding region : (x_0-x_l, y_0-y_d) to (x_1+x_r, y_1+y_u)

with

(x_0,y_0) to (x_1,y_1) : region originally occupied by the cell

$$x_l = \frac{1}{2} \min \text{ (spacing constraints of edges going left),}$$

$$x_r = \frac{1}{2} \min \text{ (spacing constraints of edges going right),}$$

$$x_u = \frac{1}{2} \min \text{ (spacing constraints of edges going up),}$$

$$x_d = \frac{1}{2} \min \text{ (spacing constraints of edges going down).}$$

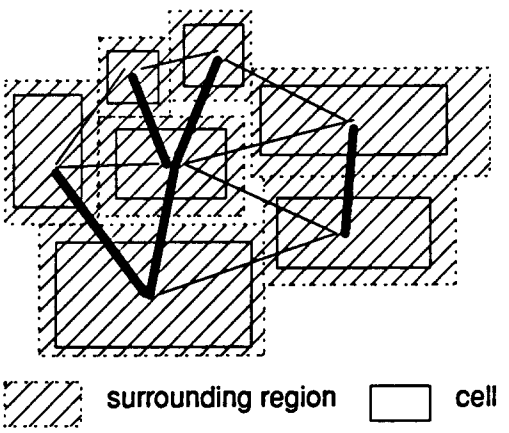


Fig. 4.6 The surrounding regions for the cells.

The surrounding regions of all the cells won't overlap. However, as shown in Fig. 4.6, these surrounding regions won't cover the whole routing region. In the second step of computing the desired move for all the wires, the region covered by the edge of the

RULD-graph, as defined in Section 4.4.1, are used again. A slight modification here is that the surrounding region of each cell, instead of its actual shape, is used in computing this region. Fig. 4.7 demonstrates how these regions fill the space not covered by the surrounding regions of the cells. For a wire touching the region covered by an edge, its desired move will be the move of the cell connected by the left end (for a horizontal edge) or the lower end (for a vertical edge) of the edge.

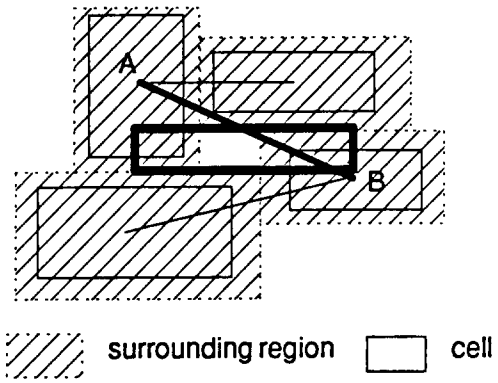


Fig. 4.7 The region (the bold rectangle) covered by the edge (A,B) can fill the empty space not covered by the surrounding region of all the neighboring cells.

With the surrounding regions of all the cells and the regions covered by all the edges, the whole routing region can be covered. Every wire will have at least one desired move. For wires with multiple desired moves, the minimum x-direction move and the minimum y-direction move will be chosen as the final move. After moving all the wires with the selected move, wires with multiple desired moves may break connections or cause new conflicts.

4.4.2.2 Stretching and Shrinking Wires

The second phase of the wire-adjusting algorithm reconnects some of the broken nets with simple operations to minimize the number of the stretched links. This algorithm tries to extend or shorten the wires to eliminate the use of stretched links. Basically, these operations can reconnect all the broken connection between perpendicular wires but not between parallel wires or between wires and pins. Because most of the connection in routing results are between perpendicular wires, these wire-stretching/shrinking operations can remove most of the stretched links. Table 4.1 shows how the number of the conflicts and stretched links are changed when the cells are moved.

	case #1		case #2		case #3	
	# wires = 1824		# wires = 1895		# wires = 1837	
	#str.-links	# con-flicts	#str.-links	#conflicts	#str.-links	#conflicts
before move	0	35	0	8	0	3
after moving wires	428	-	243	-	97	-
after stretching wires	59	193	22	95	7	33
after rerouting	0	8	0	3	0	0

Table 4.1 The effect of the placement adjustment algorithm on the number of stretched links (str.-links) and conflicts. These numbers are extracted from three placement adjustment phases in processing a macro-cell example.

Table 4.1 shows that the wire-stretching phase can eliminate most of the stretched links created when the wires are moved. It also shows that only a small portion of the wires will get involved in new conflicts after the placement adjustment phase. With the increased routing space, most of these conflicts can be resolved quickly by the succeeding rerouting phase.

4.5 Overall Algorithm

Based on the enhanced router and the placement adjustment algorithm, the overall algorithm can be described in the following C-like pseudo-code.

```
main()
{
   build_RULD_graph();
   add_space_around_cell();      /* only a small amount of space */
   1D_compact_RULD_graph();
   loop = 0 ;
   do {
      route(loop);
      compute_spacing_constraints_from_congestion();
      1D_compact_RULD_graph();
      move_wires_with_cells();
      loop ++ ;
   } while (! finished() ) ;
}

route(loop)
{
   if (loop == 0 ) {
      global_route(top_level, IGNORE_ADJ_CONGESTION) ;
      return ;
   } else if ( loop == 1 ) {
      global_route(top_level, REROUTE_ONLY);
      detail_route();
      return ;
   } else { /* loop = 2 */
      detail_route();
      for (i=0; (! complete) && ( i < (loop-1) ) ; i++ ) {
         global_route(detailed_level, REROUTE_ONLY);
         detail_route();
      }
   }
}
```

In this routine, the procedures build_RULD_graph and 1D_compact_RULD_graph are the ones presented in Chapter 2. The procedure route is the enhanced area router. When it is called the first time, it ignores all the congestions in the adjustable area and returns before conducting any detailed routing. The second time, it repeats the hierarchical routing from the highest level while using only global rerouting algorithms. Right after the router tries the detailed routing algorithm once, it returns, and the placement adjustment routine then tries to move the cells again based on the detailed routing result. Afterwards, the procedure route tries harder and harder, iterating between the global rerouting and detailed rerouting algorithms several times, before it returns and resorts to the placement adjustment routines. This loop repeats until all the nets are implemented with proper wires and vias without any conflicts. Fig. 4.8 shows how the layout is changed during such an iteration loop.

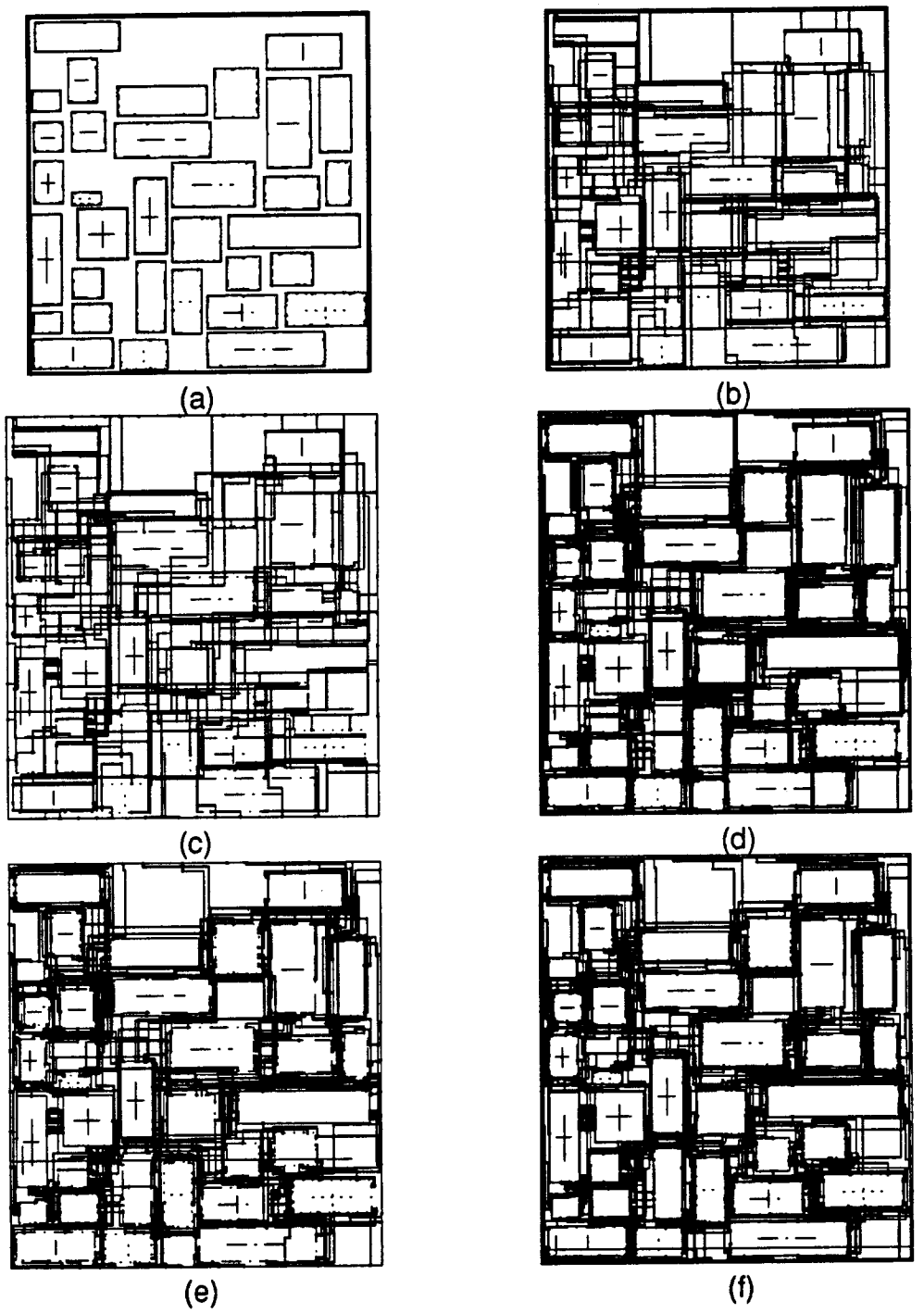


Fig. 4.8 The change of the layout in the placement and routing loop. (a) Initial layout after inserting a small amount of routing space around each cell. (b) After first global routing phase. (c) After the placement adjustment phase. (d) After another pass of global routing and then detailed routing. (e) After another placement adjustment phase. (e) Final result after several iterations.

4.6 Results

The integrated placement and routing scheme introduced in this chapter is a very general approach that is quite different from traditional approaches. To evaluate this new approach, a prototype system has been developed and compared with different approaches on various kinds of problems.

4.6.1 Comparison with Fixed Placement and Compaction

An alternative to complete routing for a given placement is to allocate additional routing space in advance and then use a layout compactor to minimize the wasted space of the routing result. This approach, which will be referred to as the *Place-Route-Compact* or simply *P-R-C* approach, is compared with the integrated placement and routing approach presented in this chapter.

This comparison is made on the two well-known MCNC macro-cell benchmarks, AMI33 and AMI49. For each benchmark, an initial placement without any routing space allocated between cells is first obtained using the placement algorithm introduced in Chapter 2. For the P-R-C approach, simple heuristics are used to allocate a wiring zone surrounding each cell. For each cell, the extended wiring zone for an edge E is computed based on the following formula:

$$\text{extension(E)} = S \times \sqrt{\text{pin\_number}(E) \times 2}$$

where

$\text{pin\_number}(E)$ = number of pins on edge E;

S = an adjustable scaling factor.

For each benchmark example, several scaling factors are tested and the three smallest ones that can generate enough space to allow the router to complete all the nets are presented. The area router presented in Chapter 3 is used here. The routing results are processed with an industrial symbolic-layout compactor to squeeze out the wasted space. To make a fair comparison, the final results from the integrated placement and routing approach are also processed by the same compactor. Table 4.2 shows the results of this experiment and Fig. 4.9 and Fig. 4.10 compare the layouts of the two different approaches.

Example	Approach (scaling factor)	before compaction		after compaction	
		area	wire-length	area	wire-length
ami33	**Integ. P&R**	**2.71**	**124.2**	**2.41**	**113.6**
	P-R-C(2)	2.99	132.2	2.52	117.0
	P-R-C(3)	3.96	153.6	2.59	126.5
	P-R-C(4)	5.07	179.5	2.96	135.7
ami49	**Integ. P&R**	**55.37**	**1000.7**	**51.99**	**953.2**
	P-R-C(5)	66.72	1068.2	54.97	960.3
	P-R-C(6)	72.69	1110.2	56.42	972.1
	P-R-C(7)	77.78	1155.8	57.35	982.3

Table 4.2 Results from the two different approaches. For the P-R-C approach, three different scaling factors are used to generate placements for routing. The unit is mm^2 for the areas and *mm* for the wire-lengths.

Table 4.2 shows that when additional space is allocated for the router initially, the compactor can squeeze out a considerable amount of wasted space but the results are not as small as those generated by the integrated placement and routing approach introduced in this chapter. This verifies the statement made in Section 4.1 that the router, given too much space, may generate results that are difficult to compact.

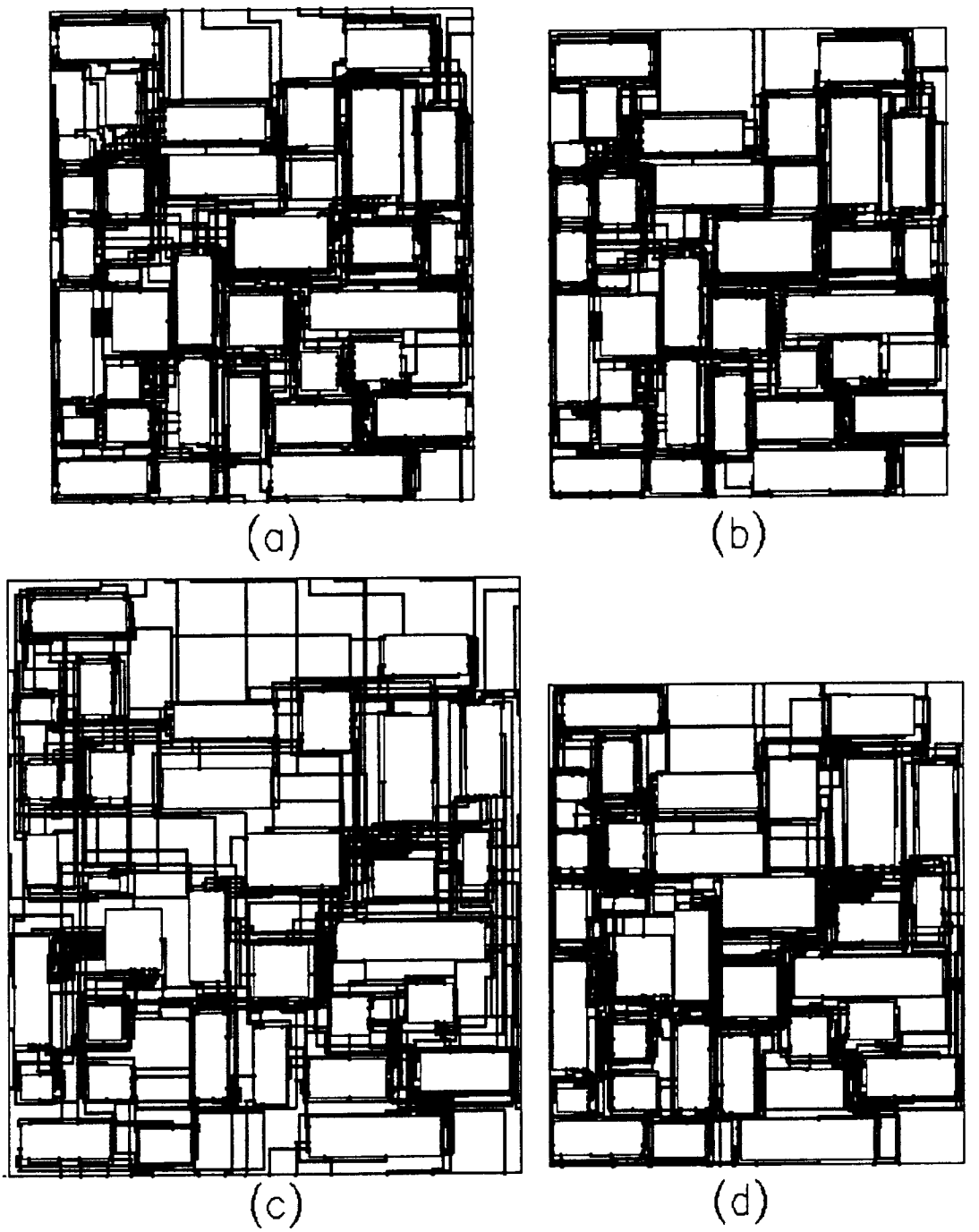


Fig. 4.9 Comparison of the integrated placement and routing approach (a) (b) and the Place-Route-Compact approach (c) (d) on AMI33. (a) (c) are layouts before compaction and (b) (d) are compacted results. The scaling factor of the P-R-C approach is 3.

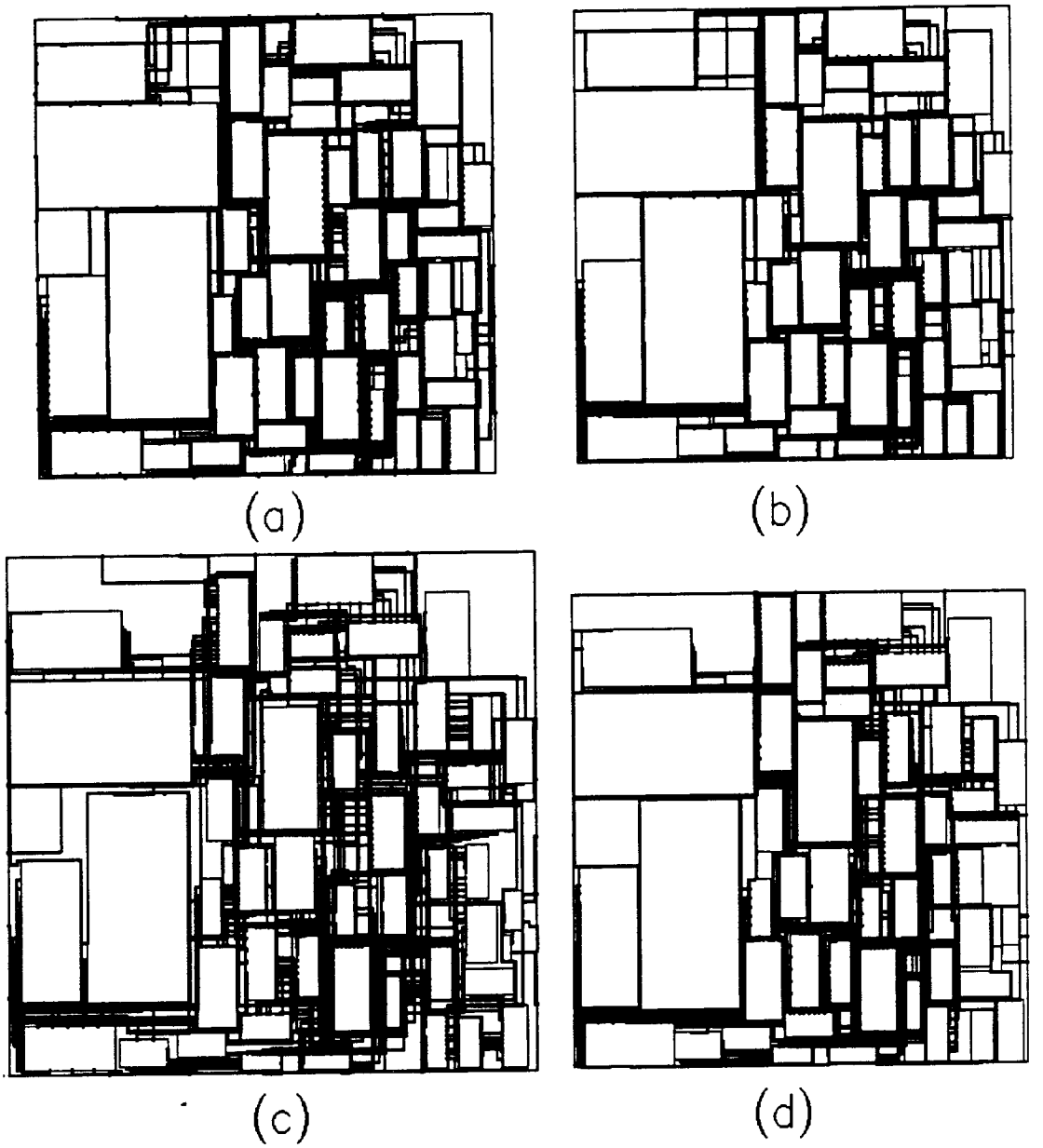


Fig. 4.10 More comparison of the integrated placement and routing approach (a) (b) and the Place-Route-Compact approach (c) (d) on AMI49. (a) (c) are layouts before compaction and (b) (d) are compacted results. The scaling factor of the P-R-C approach is 6.

4.6.2 Macro-cell problems with Over-the-Cell Routing

The primary application targets of this integrated system are macro-cell problems with over-the-cell routing. Even though there have been real designs implemented in this style, there are still no benchmark examples available in the public domain. To the best of our knowledge, there are also no published result on this kind of problem. Therefore, several artificial examples have been created to evaluate the system.

The artificial examples are derived from the two MCNC benchmarks - AMI33 and AMI49. Because the two benchmarks are originally designed for channel-based design style, it would create problems that are too easy by simply allowing the wires to go over the cells. For these two examples, the original net lists are used while only the cells are modified. For each cell in the design, it is first scaled by a random number from 0.8 to 1.0 and then filled with randomly generated rectangular obstacles. The total area of these random obstacles is controlled to be in the range of 20% to 60% of the cell area, with the percentage also generated randomly for each cell. Finally, every pin on the cell is shifted randomly to an arbitrary location in the cell that is not covered by any obstacles. Fig. 4.11 demonstrates how a cell with obstacles and pins inside is generated from a cell only with pins on the boundary.

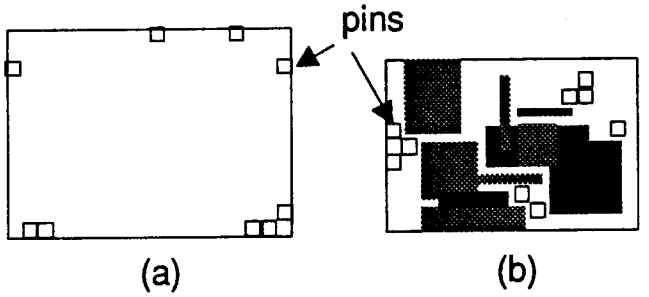
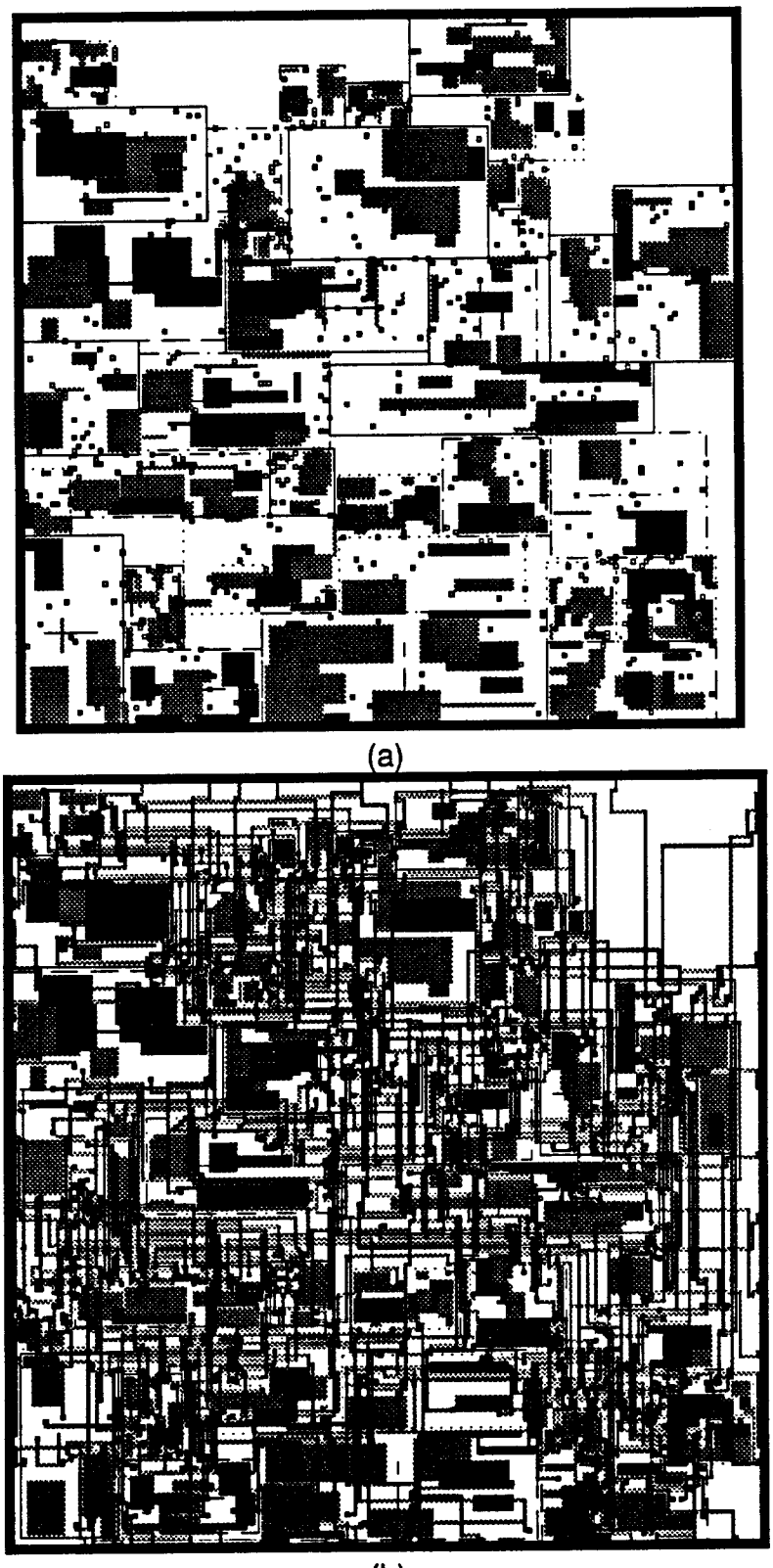


Fig. 4.11 An example of the randomly generated cells. (a) The original cell and (b) the generated cell with obstacles and pins inside.

Fig. 4.12 and Fig. 4.13 show the layouts of the initial placements and final results of these two randomly generated problems. Table 4.3 summaries the results for these two problems.

Example	Initial Area	Final Area	Wire Length	CPU time	# Placement Adjustments
AMI33otc	1135x1120	1239x1239	113848	232	4
AMI49otc	5978x5887	6062x6258	985327	2844	10

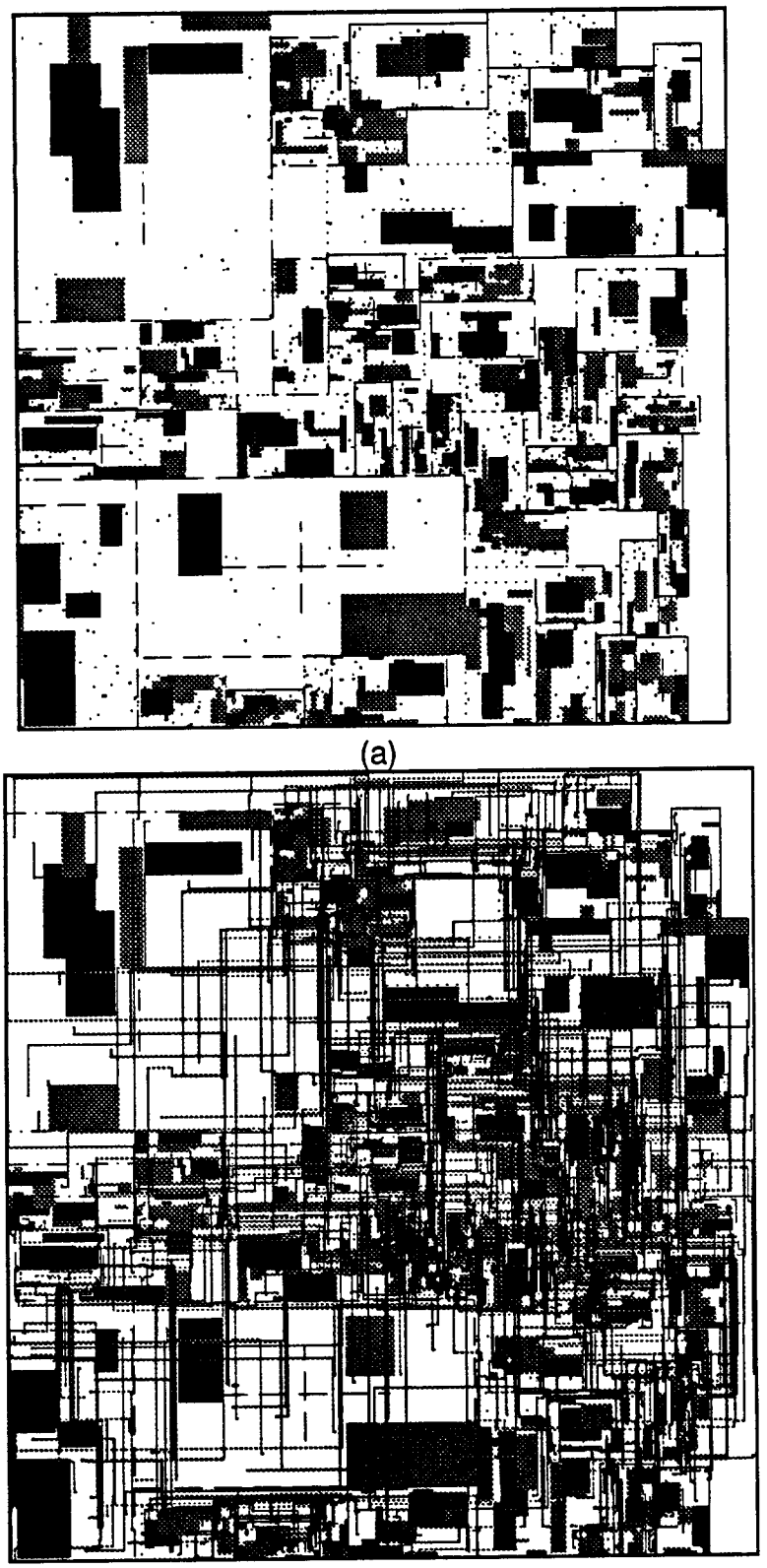
Table 4.3 Results of on macro-cell examples with over-the-cell routing. All the linear dimensions are in μm. The CPU-times are measured in seconds on a SUN SPARC-1.



(a)

(b)

Fig. 4.12 The result of AMI33otc: (a) initial placement and (b) final result.



(a)

(b)

Fig. 4.13 The result of AMI49otc: (a) initial placement and (b) final result.

4.6.3 Channel-based Macro-cell Problems

Even though this system is designed for handling problems with over-the-cell routing, the system can also handle traditional channel-based macro-cell designs. Table 4.4 summaries our results of the two MCNC macro-cell benchmarks, AMI33 and AMI49, and Fig. 4.14 shows the final layout of AMI33. Table 4.5 compares the compacted results of our integrated placement and routing approach with other published results.

Example	Initial Area	Final Area	Wire Length	CPU time	#Placement Adjustment
AMI33	1442x1491	1575x1694	124196	183	5
AMI49	7105x6713	7763x7133	1000692	3030	6

Table 4.4 Results of on macro-cell examples. The final results here are before compaction. All the linear dimensions are in μm. The CPU-times are measured in seconds on a SUN SPARC-1.

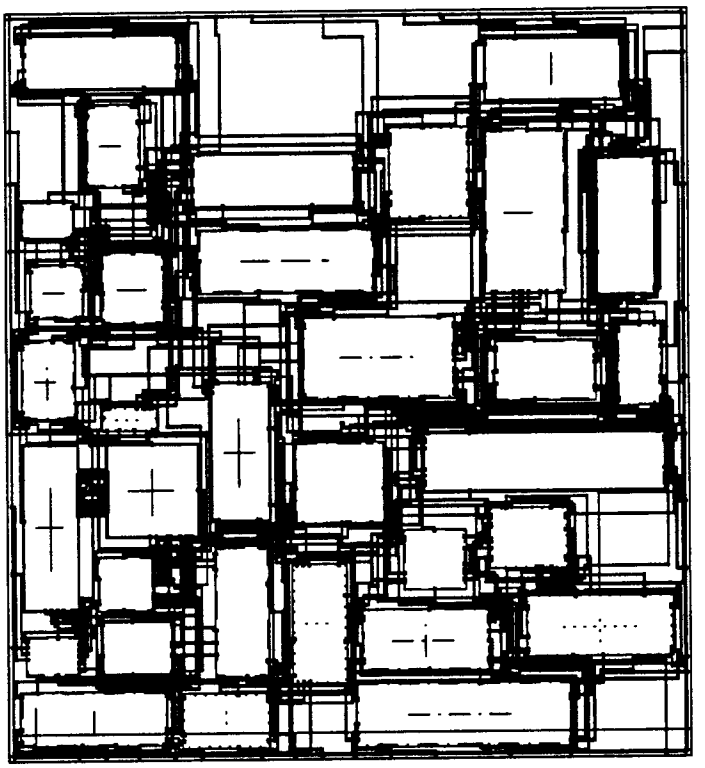


Fig. 4.14 Resulting layout of AMI33 (before compaction).

Example	System	Area	Wire Length
AMI33	Delft P&R	2.60	152
	Industrial 2	3.12	135
	Bear	2.83	131
	Industrial 1	2.94	125
	Mosaico	2.71	118
	TimberWolf	2.57	105
	MBP	2.42	91
	Integ. P&R	**2.41**	**113**
AMI49	MBP	48.79	904
	Integ. P&R	**51.99**	**953**

Table 4.5 Comparison of the placement and routing results of MCNC macro-cell benchmarks. The results of the integrated placement and routing approach (Integ. P&R) are from compacted layouts. All the numbers here are obtained from [SS90] except the results of MBP, which are published in [USS90]. The unit is mm^2 for the areas and *mm* for the wire-lengths.

Among the various approaches shown in Table 4.5, the results on the Delft P&R and the Mosaico systems were obtained with the use of interactive manual placement; furthermore, Mosaico also used interactive global routing modifications. Both TimberWolf and MBP used the simulated-annealing approach for placement and channel-based global and detailed routers to complete the routing. All these methods take advantage of the knowledge of the channel structure of the wiring area, and have been optimized and fine-tuned for this type of problem. Thus it is not surprising that our results are not the best among them because our system is designed to handle much more general problems that cannot be handled by any of these systems. However, our results are fairly close to the best published results and are better than those of some systems designed specifically for this problem class.

4.7 Summary

An integrated placement and routing approach has been developed. By using efficient incremental routing and placement adjustment algorithms, the system can iterate between the two algorithms to generate optimized results with reasonable time complexity. This iterative optimization loop can produce layouts better than the approaches that rely on layout compactors to minimize the final area. Furthermore, such a general approach can handle problems that cannot be handled by other approaches.

Chapter 5
Discussions and Conclusions

This dissertation presents new approaches for compaction, placement, and routing in layout synthesis and optimization. The major research effort has been focused on integrating these various algorithms to provide smaller layouts with shorter total wire length. The major contribution of this dissertation is to demonstrate an explicit way to integrate the various algorithms via carefully designed combined data structures. The results show that various layout routines can work cooperatively in an iterative optimization loop, which traditionally has been very difficult and/or typically required considerable amount of human intervention. The key ideas will be reviewed in the following sections.

5.1 Integrated Data Structure

The first major contribution of this dissertation is the use of a concise integrated data structure to allow the different algorithms to work together effectively. Typically, different algorithms require different data structures to work in the most efficient way. However, when separate data structures are used for different algorithms or tools, data that needs to be communicated from one routine to another must be transformed from one format to the other. Furthermore, if there is duplicated information in different data structures, it also requires special routines to maintain the consistency of the shared data. The overhead of data transformation and consistency maintenance can be quite expensive.

The basic approach presented in this dissertation is to use an integrated data structure that keeps only the highest-level of data abstraction. The goal is to minimize the size of the data structure and the overhead of modifying the data structure. Any information that can be derived on the fly is not stored, even though it may be necessary to recompute the data many times. This approach manifests itself in several places in this dissertation.

The first demonstration of this idea is the use of the RULD-graph for 2-D compaction. Traditional compaction algorithms require two constraint-graphs for the two compaction directions, horizontal and vertical. Most compactors, 1-D or 2-D, have to iterate between the two directions. However, the constraint-graph used in one compaction pass usually cannot be used again in the following passes because the changes of the location of the blocks in the other direction make the constraint-graph invalid. The RULD-graph combines the horizontal and vertical constraint-graphs into a single data structure. All the necessary horizontal and vertical constraints can be derived from it very quickly. Even though deriving

and enforcing the implicit constraints is more expensive than using the traditional constraint-graph, its cost is small compared to that of rebuilding the complete constraint-graphs several times. This allows the compactor to compute 2-D moves efficiently and to iterate between horizontal and vertical compaction easily.

Another example is the unified wiring data structure presented in Chapter 3. Traditionally, the global router and the detailed router have been working independently. Special track-assignment algorithms are then required to convert global routing results for the use of the detailed router, and no information flows in the inverse direction. By using the same wiring data structure for both the global and detailed routing algorithms, no special data conversion is required. This same data structure also integrates and provides the means of communication for the different levels of the global router. In addition, the data structure also carries the important congestion information that can be shared by all the routing algorithms. Even though there is a certain overhead using the relatively low-level wiring data structure for the global routing algorithms, the advantage of allowing the router to move between different levels makes the router overall much more efficient.

This wiring data structure is also used for integrating the routing and placement adjustment algorithms. The placement adjustment algorithm uses the congestion information to estimate the required space among cells. After moving the cells, the placement adjustment algorithm modifies the wiring data structure directly. The *stretched link* provides a convenient way for the placement adjustment algorithm to pass connectivity information back to the router. Because all the information exchange between the routing and placement adjustment phases goes through a well-defined data structure, the transition between the two different algorithms is very simple and smooth. Therefore, it is possible to iterate between the two algorithms to achieve better overall results.

5.2 Iterative Optimization Using Cooperative Algorithms

The second major contribution of this dissertation is to demonstrate that iterating between algorithms with different functions can produce better overall results. VLSI design is a very complexity task that traditionally has required several manual iterations to achieve a satisfactory result. Similarly, the CAD tools for automating this process should also be able to iterate freely. However, iterating sequences of CAD tools or algorithms have not been practical in the past because of the high time complexity of the individual algorithms and the lack of good feedback mechanisms.

The basic idea demonstrated in this dissertation is to use fast and effective incremental updating routines to work on an integrated data structure. Instead of building complex estimation algorithms or statistical model to provide information from lower-level tools to higher ones, simple and efficient algorithms are used to conduct the actual low-level tasks

to collect real factual data. All the tools may start with imprecise estimates. As the system iterates between different tools, more precise information is collected by each tool and this information can be taken into account by the other tools. The overall result can be improved gradually until a satisfactory result is achieved. This idea is demonstrated in several places in this dissertation.

The first demonstration of the iterative optimization loop is the placement algorithm that iterates between the wire-length optimization algorithms and the 2-D compaction algorithm. Traditionally, the task of resolving overlaps is integrated into the wire-length optimization algorithm as some cost functions that estimate the area of the final layout. However, the estimate is not always precise, and the additional cost terms usually make the optimization routines much more complex and slow. By dividing the placement process into wire-length optimization and overlap-resolving phases, each phase can be made more efficient. It is then possible to iterate between the two phases to explore many good legal solutions quickly and choose the best one.

The second demonstration of the iterative optimization loop is the process of integrated global and detailed routing. Traditionally, the global and detailed routing tasks are conducted sequentially. In this approach, the global router have to build complex model to estimate the capacity requirement for detailed routing. However, a very precise model is normally not available. By using efficient incremental rerouting routines in both the global and the detailed routing phases, the router can switch between the two algorithms, choosing the most efficient approach to ease the congestion and resolve the conflicts. Even though very simple capacity models are used initially by the global router, the efficient rerouting routines can quickly correct the original estimates based on the feedback from actual detailed routing results. The overall algorithm is still very efficient even though it iterates between two algorithms several times.

The third demonstration of the iterative optimization loop is the integrated routing and placement adjustment scheme. This is probably the first ever successful attempt to iterate between these two processes automatically to complete the task. This integration is made possible primarily by the rerouting operations that can incrementally modify the wiring in an efficient manner. Again, this approach uses initially simple and imprecise congestion information to space the cells. The placement is then adjusted gradually based on the exact routing results. Even though simple capacity models are used initially, the final result can still be densely packed with the congestion evenly distributed.

5.3 Conclusion

In the past few years, many VLSI CAD tools and algorithms have been developed or proposed. Even with these powerful tools and novel algorithms, it is still very time-con-

suming to accomplish a high-quality design. One of the major problem has been the lack of interaction between the various tools, especially the good feedback mechanisms from the lower-level tools to the higher ones. The detailed router may not be able to complete a good global routing result. A highly-optimized placement result may not be routable. This problem is not limited to the domain of the physical design. A highly-optimized net-list from a logic synthesis tool may be very difficult to place and route, resulting a very large and/or slow chip.

The techniques and results presented in this dissertation further support the conjecture that various CAD layout routines can work cooperatively in an iterative optimization loop to generate smaller layouts with shorter total wire length. Such an iteration loop is made possible with a carefully designed combined data structure and efficient incremental modification routines. This basic principle can be extended to combine more CAD algorithms into an optimization loop to generate high-quality chip designs with little human intervention.

References

[BHP83] M. Burstein, S.J. Hong, and R. Pelavin. Hierarchical VLSI layout: Simultaneous placement and wiring of gate arrays. In *Proc. of IFIP VLSI-83*, 1983.

[Boy87a] D.G. Boyer. Symbolic compaction benchmarks - introduction and ground rules. In *Proc. ICCD-87*, pages 186–191, Oct. 1987.

[Boy87b] D.G. Boyer. Symbolic compaction benchmarks - results. In *Proc. ICCD-87*, pages 209–217, Oct. 1987.

[BP83] M. Burstein and R. Pelavin. Hierarchical wire routing. *IEEE Trans. CAD of ICs and Systems*, CAD-2(4):223–234, Oct. 1983.

[Bre77] M.A. Breuer. Min-cut placement. *J. Design and Fault Tolerant Computing*, 1(4):343–362, Oct. 1977.

[Che87] H.H. Chen. Routing L-shaped channels in nonslicing structure placement. In *Proc. 24th Design Automation Conf.*, pages 152–158, 1987.

[Chr89] W.A. Christopher. Mariner: A sea-of-gates layout system. Technical Report M89/83, UCB/ERL, June 1989.

[CW91] Y. Cai and D.F. Wong. On minimizing the number of L-shaped channels. In *Proc. 28th Design Automation Conf.*, pages 328–334, 1991.

[DAK85] W.M. Dai, T. Asano, and E.S. Kuh. Routing region definition and ordering scheme for building-block layout. *IEEE Trans. CAD of ICs and Systems*, CAD-4(3):189–197, July 1985.

[FCW67] C.J. Fisk, D.L. Caskey, and L.E. West. ACCEL: automated circuit card etching layout. *Proc. of the IEEE*, 55(11):1971–1982, 1967.

[GV83] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1983.

[Hig69] D.W. Hightower. A solution to line-routing problem on the continuous plane. In *Proc. 6th Design Automation Workshop*, pages 1–24, 1969.

[Hoj90] R. Hojati. Layout optimization by pattern modification. In *Proc. 27th Design Automation Conf.*, pages 632–637, 1990.

[HP79] M.Y. Hsueh and D.O. Pederson. Computer-aided layout of LSI circuit building blocks. In *Proc. 1979 International Symposium on Circuits and Systems*, pages 474–477, 1979.

[HS71] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment. In *Proc. 8th Design Automation Conf.*, pages 214–224, 1971.

[HWA73] M. Hanan, P.K. Wolff, Sr., and B.J. Agule. Some experimental results on placement techniques. In *Proc. 13th Design Automation Conf.*, pages 214–224, 1973.

[IBSV89] M. Igusa, M. Beardslee, and A. Sangiovanni-Vincentelli. ORCA: A sea-of-gates place and route system. In *Proc. 26th Design Automation Conf.*, pages 122–127, 1989.

[Joh87] F.M. Johannes. Use of triangulation for global placement. In *Proc. VLSI'87, Vancouver*, pages 151–160, Aug. 1987.

[Kah89] A.B. Kahng. Space-filling curve techniques for CAD/CAM. In *Proc. Advanced Research in VLSI Conference, Caltech, Pasadena*, pages 261–277, 1989.

[KK88] K. Kozminski and E. Kinnen. Rectangular dualization and rectangular dissection. *IEEE Trans. Circuits and Systems*, 35(11):1401–1416, Nov. 1988.

[KPS89] B. Korte, H.J. Prömel, and A. Steger. Combining partitioning and global routing in sea-of-cells design. In *Proc. ICCAD-89*, pages 98–101, 1989.

[KSJ88] J.M. Kleinhans, G. Sigl, and F.M. Johannes. Gordian: A new global optimization / rectangle dissection method for cell placement. In *Proc. ICCAD-88*, pages 506–510, 1988.

[KW84] G. Kedem and H. Watanabe. Graph-optimization techniques for IC layout and compaction. *IEEE Trans. CAD of ICs and Systems*, CAD-3(1):12–20, Jan 1984.

[Lau79] U. Lauther. A min-cut placement algorithm for general cell assemblies based on a graph representation. In *Proc. 16th Design Automation Conf.*, pages 1–10, 1979.

[Lau87] U. Lauther. Top down hierarchical global routing for channelless gate arrays based on linear assignment. In *Proc. of IFIP VLSI-87*, 1987.

[Lee61] C.Y. Lee. An algorithm for path connection and its application. *IRE Trans. on Electronic Computers*, EC-10:346–365, Sept. 1961.

[Lee90] B.D.N. Lee. Experiments in hierarchical routing of general areas. Technical Report M90/17, UCB/ERL, March 1990.

[LHT90] Y.-L Lin, Y.-C. Hsu, and F.-S. Tsai. Hybrid routing. *IEEE Trans. CAD of ICs and Systems*, CAD-9(2):151–157, 1990.

[Luk85] W.K. Luk. A greedy switch-box router. *Integration*, 3:129–149, 1985.

[Oht85] T. Ohtsuki, editor. *Layout Design and Verification*, volume 4 of *Advances In CAD for VLSI*. North-Holland, 1985.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry, An Introduction*. Springer-Verlag, 1985.

[RF82] R.L. Rivest and C.M. Fiduccia. A 'greedy' channel router. In *Proc. 19th Design Automation Conf.*, pages 418–424, 1982.

[RJ89] H.C. Ranke and F.M. Johannes. Macrocell placement by global optimization with uniform cell distribution. In *Proc. VLSI'89, Munich*, pages 423–432, Aug. 1989.

[RSVS85] J. Reed, A. Sangiovanni-Vincentelli, and M. Santomauro. A new symbolic channel router: YACR2. *IEEE Trans. CAD of ICs and Systems*, CAD-4(3):208–219, 1985.

[Rub74] F. Rubin. The Lee path connection algorithm. *IEEE Trans. on Computers*, C-23(9):907–914, 1974.

[SB87] L. Sha and T. Blank. Atlas - a technique for layout using analytic shapes. In *Proc. ICCAD-87*, pages 84–87, 1987.

[Shi87] H. Shin. *Two-Dimensional Routing and Compaction in Computer-Aided Design of Integrated Circuits*. PhD thesis, U.C. Berkeley, 1987.

[SK88] P.R. Suaris and G. Kedem. An algorithm for quadrisection and its application to standard cell placement. *IEEE Trans. Circuit and Systems*, CAS-35(3):294–303, 1988.

[SLS87] E. Shragowitz, J. Lee, and S. Sahni. Placer-router for sea-of-gates design style. In *Proc. ICCD-87*, pages 330–335, 1987.

[SLW83] M. Schlag, Y.Z. Liao, and C.K. Wong. An algorithm for optimal two-dimensional compaction of VLSI layout. *Integration*, 1:179–209, 1983.

[SS90] W. Swartz and C. Sechen. New algorithms for the placement and routing of macro cells. In *Proc. ICCAD-90*, pages 336–339, 1990.

[SSV84] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. In *Proc. 1984 Custom Integrated Circuit Conference*, pages 522–527, 1984.

[SSVS86] H. Shin, A. Sangiovanni-Vincentelli, and C.H. Séquin. Two dimensional compaction by zone refining. In *Proc. 23rd Design Automation Conf.*, pages 115–122, 1986.

[Sze86] A.A. Szepieniec. Integrated placement/routing in sliced layouts. In *Proc. 23rd Design Automation Conference*, pages 300–307, 1986.

[TFKM91] Y. Tamiya, M. Fujita, T. Kakuda, and Y. Matsunaga. Macro cell placement based on a rectangular dual representation. In *Proc. 1991 Physical Design Workshop*, 1991.

[TKH88] R.-S. Tsay, E.S. Kuh, and C.-P. Hsu. PROUD: A fast Sea-Of-Gates placement algorithm. In *Proc. 25th Design Automation Conf.*, pages 318–323, 1988.

[TS88] P.-S. Tzeng and C.H. Séquin. Codar: A congestion-directed general area router. In *Proc. ICCAD-88*, pages 30–33, 1988.

[TS91a] P.-S. Tzeng and C.H. Séquin. A data structure for resolving overlaps in macro-block placement. In *Proc. Physical Design Workshop*, 1991.

[TS91b] P.-S. Tzeng and C.H. Séquin. Macro-block placement using efficient 2-d compaction. In *Proc. Advanced Research in VLSI Conference, Santa Cruz*, pages 178–191, March 1991.

[Tsa89] R.-S. Tsay. *Partitioning, Placement, and Routing Algorithms for High Complexity Integrated Circuits.* PhD thesis, U.C. Berkeley, 1989.

[USS90] M. Upton, K. Samii, and S. Sugiyama. Simulated annealing placement for mixed macro cell and standard cell layouts. In *Proc. International Workshop on Layout Synthesis*, 1990.

[VKLS90] J. Valainis, S. Kaptanoglu, E. Liu, and R. Suaya. Two-dimensional IC layout compaction based on topological design rule checking. *IEEE Trans. CAD of ICs and Systems*, CAD-9(3):260–275, March 1990.

[WMND88] W.H. Wolf, R.G. Mathews, J.A. Newkirk, and R.W Dutton. Algorithms for optimizing two-dimensional symbolic layout compaction. *IEEE Trans. CAD of ICs and Systems*, CAD-7(4):451–466, April 1988.

[Xio89] X.-M. Xiong. Two-dimensional compaction for placement refinement. In *Proc. ICCAD-89*, pages 136–140, 1989.