# TWO-DIMENSIONAL ROUTING AND COMPACTION IN COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS

by

Hyunchul Shin

Memorandum No. UCB/ERL M87/92

25 October 1987

# TWO-DIMENSIONAL ROUTING AND COMPACTION IN COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS

by

Hyunchul Shin

Memorandum No. UCB/ERL M87/92

25 October 1987

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# TWO-DIMENSIONAL ROUTING AND COMPACTION IN COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS

Ph. D. Hyunchul Shin Department of EECS

[illegible]

Chairman of Committee

## ABSTRACT

Routing and compaction are two essential steps in the physical design of VLSI circuits. Several approaches have been proposed in the recent past for these two tasks. Most of the work in routing has been performed on channel routing, i.e., on routing rectangular area where the pins to be connected lie on two parallel edges of the region. For restricted layout methodologies such as standard cell, gate array, and macro-cell arranged in a slicing structure, channel routers can be used to route the entire chip. However, there are cases where these restricted approaches can not be used for efficiency reasons. Two-dimensional routers are often necessary for the macro-cell design style and for routing problems in which the routing regions are irregular. The routing regions that the technique proposed in this thesis can handle are very general: their boundaries can be described by any rectilinear set of edges, the pins can be on or inside this boundary, and obstructions can be of any shape in any layer. The technique is based on an algorithm that routes the nets in the routing region incrementally using minimum cost paths, and allows modifications by ripping-up nets already routed when an existing shortest path is "far" from optimal or when there is no room for a needed path. Some modification steps (called weak modification) relocate segments of nets already routed

to find shorter paths or to make room for a blocked net. The rip-up and reroute steps (called strong modification) remove segments of nets already routed to make room for a blocked connection; these steps are invoked only if weak modification fails. The algorithm has been proven to complete in finite time and its complexity has been analyzed. Many test cases have been run, and on all the examples known in the literature the router has perfomed as well as or better than existing algorithms. In particular, the Burstein's difficult switchbox example has been routed using one less column than the original data. In addition, the router has routed difficult channels such as Deutsch's example in density and has performed better than or as well as YACR-II on all the channels available to us.

Compaction is a critical step in a symbolic layout system. It can be used in module generators as well as in post-processors for placement and routing systems. Most of the compactors in practical use today rely on one-dimensional compaction techniques, i.e., on methods that compact the layout by pushing elements in one direction at a time. This method is fast but it can not always produce compact layouts that can rival those of a human designer. Two-dimensional compaction allows more complex "moves" of the components of the layout and can produce better results. A new two-dimensional compaction strategy has been proposed based on zone-refining. The zone-refining layout compaction technique bears a strong similarity to a technique used in the purification of crystal ingots. Individual circuit components or small clusters of components are peeled off row by row from the pre-compacted layout, moved across an open zone, and reassembled at the other end of this zone in a denser configuration. In this process both coordinates of the moved components are altered and jogs are introduced in the connecting wires between them to produce the needed flexibility for placing components into optimal positions. This general approach provides a flexible framework. Without lateral movements of the components it degenerates to a one-dimensional compactor. At the other extreme, simulated annealing techniques can also be employed within the zone-refining process. This permits tradeoffs of run-time and final layout density.

## Acknowledgements

I would like to thank my research advisor, Professor Alberto L. Sangiovanni-Vincentelli, for his continuous patience, guidance, and support. The discussions with him were invaluable. I am also grateful to Professor Carlo H. Sequin for the useful suggestions and encouragement, to Professor A. Richard Newton for general advice, and to Professor James Sethian for reading this thesis.

Mark Bales developed a compactor, Python, in 1982, which helped me understand compaction routines at the beginning of Zorro project. Discussions with Wayne Wolf at Bell Laboratories on critical path shearing are acknowledged. Several references on routing were available from Marek-Sadowska. Klaus Rohe from Siemens, West Germany, patiently used the early version of Mighty router before it could be extensively debugged to be practical and helped me in finding a few bugs. I also thank my fellow graduate students in CAD group. While it is impossible to name everyone who helped me during my years at U.C. Berkeley, I must mention Rick Spickelmier and Tom Quarles, who helped me with many computer related problems. Discussions with Jeff Burns, Faye Marron, Fabio Romeo, Richard Rudell, Young Kim, Hyun J. Shin, Seung Hwang, Hong Park and many others are acknowledged.

The research for this thesis was supported in part by the Defense Advanced Research Projects Agency under contract number N00039-83-C-0107, Hughes Aircraft Semiconductor Division, AMI, Intel, and the MICRO Project of the State of California.

Last, but not least, I thank my parents, brother, and sisters for their trust and encouragement. I am most indebted to my wife, Meejung, for her continuous love and understanding, and to my daughter Angela for bringing new happiness to my family.

# Table of Contents

1. INTRODUCTION .......... 1

1.1 DETAILED ROUTING BASED ON INCREMENTAL WIRING MODIFICATIONS .......... 2

1.2 TWO - DIMENSIONAL LAYOUT COMPACTION USING 'ZONE REFINING' .......... 5

2. DETAILED ROUTING BASED ON INCREMENTAL ROUTING MODIFICATIONS .......... 9

2.1 PREVIOUS WORK .......... 10

2.2 FORMULATION OF THE PROBLEM AND GENERAL APPROACH .......... 15

2.3 THE ALGORITHM .......... 18

2.3.1 Data Representation .......... 18

2.3.2 The Basic Algorithm .......... 18

2.3.2.1 Overall algorithm .......... 25

2.3.3 Finding a Shortest Path .......... 26

2.3.3.1 Algorithm Find_path .......... 28

2.3.4 Confirmation of a Path .......... 29

2.3.5 Weak Modification .......... 30

2.3.6 Strong Modification .......... 35

2.4 COMPLEXITY OF THE ALGORITHM .......... 37

2.5 PARTITIONING OF LARGE PROBLEMS .......... 40

2.5.1 Pseudo-Pins .......... 40

2.5.1.1 Pseudo-Pin Generation .......... 43

2.5.1.2 Setting Parameters in the Cost Matrix .......... 44

2.6 OTHER VARIATIONS TO THE BASIC ALGORITHM .......... 45

2.6.1 Pre and Post Processing .......... 45

2.6.2 Equivalent Pins .......... 46

2.6.3 Floating Pins .......... 46

2.6.4 Obstacles and Prerouted Nets .......... 47

2.6.5 Critical Nets .......... 47

2.6.6 Routing on the Boundaries .......... 48

2.7 EXPERIMENTAL RESULTS .......... 48

2.8 SENSITIVITY OF THE ALGORITHM .......... 57

2.9 SUMMARY .......... 58

3. TWO-DIMENSIONAL LAYOUT COMPACTION USING 'ZONE-REFINING' .......... 60

3.1 PREVIOUS WORK ON TWO-DIMENSIONAL COMPACTION .......... 63

3.2 COMPACTION BY 'ZONE-REFINING' .......... 65

3.3 BOX PACKING .......... 69

3.3.1 XY Adjacency Graph .......... 71

3.3.2 Zone-Refining Algorithm .......... 71

3.3.2.1 Initialization .......... 72

3.3.2.2 Selection .......... 72

3.3.2.3 Placement ........ 72

3.3.2.4 Updating the XY Adjacency Graph ........ 73

3.3.3 Sample Results of Box-packing Algorithms ........ 74

3.3.4 Pitch Matching ........ 76

3.4 CIRCUIT LAYOUT COMPACTION ........ 78

3.4.1 Data Structures ........ 79

3.4.1.1 Floor and Ceiling ........ 79

3.4.1.2 Coarse Grid Structure ........ 80

3.4.1.3 Permanent and Non-permanent Arcs ........ 83

3.4.2 Initialization ........ 84

3.4.3 Cluster Selection ........ 84

3.4.4 Cluster Placement ........ 85

3.4.5 Updating the Constraint Graph ........ 91

3.4.6 Optimizing the Connections ........ 95

3.4.6.1 Sliding Connections ........ 96

3.4.6.2 Automatic Jog Introduction ........ 96

3.4.6.3 Merging Connections of the Same Net ........ 100

3.4.6.4 Reducing Wire Length ........ 101

3.4.7 Use of Simulated Annealing Technique ........ 103

3.5 NON-TRIVIAL DESIGN RULES ........ 104

3.6 HIERARCHICAL COMPACTION ........ 105

3.6.1 Hierarchical compaction by routing ........ 106

3.6.2 Hierarchical compaction by abutment ........ 107

3.6.3 Combined Hierarchical Compaction Strategy ........ 107

3.7 VARIATIONS .......... 112

3.7.1 Back-Tracking .......... 112

3.7.2 Inter-pass Processing .......... 112

3.7.3 Zone-Refining with Critical-Path Shearing .......... 113

3.8 APPLICATION AND RESULTS .......... 115

3.8.1 Layout Compaction by Zone-Refining .......... 115

3.8.2 Results on Benchmark Examples .......... 120

3.9 SUMMARY .......... 126

4. CONCLUSIONS AND FUTURE WORK .......... 128

4.1 EXTENSIONS AND FUTURE WORKS .......... 128

4.1.1 Extensions to Mighty .......... 129

4.1.1.1 Multi-layer Routing .......... 129

4.1.1.2 Other Extensions to Mighty .......... 131

4.1.3 Extensions to Zorro .......... 132

4.2 CONCLUSIONS .......... 133

BIBLIOGRAPHY .......... 135

Appendix A: Mighty User's Guide .......... A1

Appendix B: Zorro User's Guide .......... B1

# List of Figures

2.1 Routing area with obstacles. .......... 17

2.2 The number of floating segments of several cases. .......... 20

2.3 An example to show the routing order and the schedule. .......... 23

2.4 Examples of weak modifications. .......... 31

2.5 Channel example routed by Mighty. .......... 54

2.6 Burstein's difficult switchbox routed by Mighty. .......... 54

2.7 Burstein's difficult switchbox routed by two other routers. .......... 55

2.8 Deutsch's difficult channel routed by Mighty. .......... 56

2.9 CMOS gate(NAND (NOR X2 X3 ... X16) (NAND X1 X2 ... X15)) routed by Mighty. .......... 56

2.10 Sensitivity of the algorithm on cost parameters. .......... 57

3.1 Interference that needs to be resolved with a movement perpendicular to the axis of compaction. .......... 62

3.2 Zone-refining. .......... 66

3.3 Example of box packing in progress. .......... 70

3.4 Example of box packing. .......... 75

3.5 Pitch Matching. .......... 77

3.6 The use of space-partitioning in constraint graph construction. .......... 81

3.7 An example of placement. .......... 89

3.8 An example of sub-optimal placement. .......... 90

3.9 Constraint graphs with and without merging. .......... 92

3.10 Sliding connection. .......... 96

3.11 Two cases of automatic jog introduction. .................... 98

3.12 Jogged wire bundles after zone-refining with automatic jog introduction. .................... 99

3.13 Merging contacts. .................... 101

3.14 Clean-up removes an U-shape connection. .................... 103

3.15 A single leafcell with compacted core. .................... 109

3.16 Assembly of compacted leafcells. .................... 110

3.17 Completed hierarchical compaction of cell assembly. .................... 111

3.18 Results of layout compaction using zone-refining. .................... 118

3.19 Results of zone-refining with a fixed-width constraint. .................... 119

3.20 Compacted result of mul4x4. .................... 122

4.1 An example of three layer routing by Angelar. .................... 131

# List of Tables

2.1 Types of detailed routers .......... 13

2.2 Routing of Burstein's difficult switchbox .......... 49

2.3 Routing of switchbox examples .......... 49

2.4 Dependence on the orientation of the routing area .......... 50

2.5 Routing of Deutsch's difficult channel .......... 52

2.6 Routing of channel examples .......... 52

2.7 Routing results of channel examples .......... 53

2.8 Beautification by post-processing .......... 53

3.1 Run-times on different values of Kg .......... 83

3.2 Run-time comparisons between Zorro1 and Zorro2 .......... 94

3.3 Comparison of two zone-refining approaches .......... 114

3.4 Results of the layout compaction .......... 116

3.5 Compaction of benchmark examples .......... 121

3.6 Comparisons on benchmark examples .......... 125

1.

# Chapter 1

# Introduction

In a layout which defines the components that make up the electrical circuit to be built, electrical elements are described as one or more polygons - in many cases simply as rectangles. In MOS technology, the elements that are commonly fabricated are 1) enhancement and depletion mode transistors, 2) wires to connect other elements, and 3) vias or contacts to make connections between different layers. Translating a circuit description into a layout is a tedious and error-prone job. To be considered correct, layouts must satisfy at least two checks. First they must represent the desired electrical circuit in the sense that all device sizes must be correct and the wires must form the desired connections. Second, they must obey the layout rules dictated by the manufacturing process, e.g. the minimum dimensions of elements and the minimum spacing between them. The simplest way to produce a design-rule-correct layout is to place components widely separated, but this increases the manufacturing cost and degrades the performance of the resulting circuit.

As the number of circuit components on a chip increases, an integrated CAD system is needed to prevent an exponential increase in the required design time [23].

To reduce design complexity, mask layouts are commonly obtained by a sequence of steps. The first phase is *floor-planning* during which the overall structure of a chip is decided. The second phase is *global and detailed routing* in which all the interconnections are made between the placed blocks. The final step is *compaction or spacing* which produces a manufacturable layout in a minimum area. *Detailed routing* and *layout compaction* are two important parts in automatic cell generation as well as in the generation of a mask layout for an entire chip from symbolic design. Since each of the above steps is so complicated that finding the global optimum in any one step is NP-hard, most approaches are trying to find a good solution rather than a globally optimal one.

Following a brief introduction of the new routing and compaction algorithms in this chapter, the detailed routing algorithm and its implementation are described in detail in Chapter 2. The goal of the routing algorithm is the completion of all the connections with a minimum number of vias and minimum wire-length. In Chapter 3, a new layout compaction algorithm is introduced. The input is a symbolic layout, such as a sticks representation. The output is a detailed layout in which all components are placed as densely as possible while still satisfying the geometrical design rules. The goal of this algorithm is to minimize the overall area of the layout. Finally, conclusions and extensions are given in Chapter 4.

## 1.1. DETAILED ROUTING BASED ON INCREMENTAL WIRING MODIFICATIONS

In slicing-structure designs, for example, standard-cell designs, all detailed routing can be performed by channel routers. However, for the macro-cell design style and for routing problems where the routing regions are irregular, so called "two-dimensional" routers, rather than channel routers, are necessary. In this section, a new routing technique that can be applied for general two-layer detailed routing problems including switch boxes, channels, and partially routed areas, is described. The routing regions that can be handled are very general; their boundaries can be described by any rectilinear chain of edges, the terminals can be on the boundaries of the region or inside it, and this region can contain obstructions of any shape and size.

The input is composed of a routing area and a net-list. The routing area is given as a rectagon possibly with obstacles through which wires can not pass. A net-list is a list of all terminals (pins) to be connected and of their locations. The output is a list of wires and vias to form all the necessary connections within the routing area.

Most of conventional detailed routers do not allow backtracking so that once a path is implemented as an interconnection then this connection can not be modified later. These routers try to predict where congested regions are and which nets are difficult to connect and give priorities based on this estimate. Of course, the estimate is only approximate, and problems may develop anyway and force the router to use additional rows and/or columns. In addition, some of the interconnections may be unnecessarily far from being optimal with respect to their length and number of vias.

Several approaches include backtracking or routing modifications. Since connecting a path is not final but the path can be modified later, these routers can concentrate on performance issues such as wire-length and via minimization, as well as a completion of connections. Only a few detailed routers have been published in this category. Most of the approaches wait until a completely blocked net appears and use only one type of modification which is rip-up and re-route. Furthermore, they assume single layer of interconnection [91], or require human intervention [95].

The proposed technique routes incrementally based on the shortest path and allows *modifications and removal of nets* when an exiting path is *far* from optimal or when a path can not be found. The modification steps (also called weak modification) relocate some segments of nets already routed to find a shorter path. The rip-up and re-route steps (called strong modification) remove segments of nets already routed to make room for a blocked connection and are invoked only if weak modification fails. Since all the modification routines are symmetric - they are implemented in four directions, the routing results do not depend on the orientation of the routing region. The history of weak modifications made is stored to prevent an infinite loop and to try new types of modifications when the modification routine is called several times to find a path of a net. The algorithm has been rigorously proven to terminate in finite time and its complexity has been analyzed.

Many test cases have been run and on all the examples known in the literature the router has performed as well as or better than existing algorithms. In particular, the Burstein's difficult switch box example has been routed using one less column than the original data. In addition, the router has routed difficult channels such as Deutsch's example [17] in density and has performed better than or as well as YACR-II in all the channels available to us.

To route long channels or highly irregular routing regions, global information is used during the shortest path searching. The idea is to guide the selection of the paths by placing *pseudo-pins* inside the routing region for a number of critical nets. A linear assignment algorithm based on matching of a bipartite graph is used to minimize the sum of costs in placing the pseudo-pins. This scheme adds some global information about the routing region to the maze routing phase.

Future work includes the extension of this approach to routing of multi-layer routing regions of macro-cells and printed circuit boards, and to routing of sea-of-gates. Extending this algorithm for multi-layer (more than two layers) routing of various wire-width nets is straight-forward; however, the run-time may be considerably increased, since finding a shortest path with various wire-width in multilayer routing area will take a great deal of time. The main focus will thus be on the efficiency of the algorithm. The pseudo-pin generation routine must be extended to handle routing areas that have very irregular shapes or that contain many obstructions.

## 1.2. TWO - DIMENSIONAL LAYOUT COMPACTION USING 'ZONE REFINING'

Layout compaction plays an important role in the automatic or semi-automatic generation of integrated circuits. Fully automated layout systems gave discouraging results when compared to manually packed designs [92]. Still automation is necessary for its speed, and for getting "correct-the-first-time" designs. Symbolic layout, for which compactors are very important, has been used for productivity increase and technology independency. The automatic design of any electrical digital circuit consists of two major phases. In the first phase, the designer builds the circuit from abstract blocks such as gates, registers and latches. The result of the first phase is usually schematic diagram or a description of circuit blocks and their interconnection. The circuit description is logical in nature. In the second phase, the circuit is built from chosen modules, which are fully determined in size, shape, and internal structure.

When the library cells are stored in symbolic form, it is not necessary to store several cells for each logic block which have different output power and sizes. Instead, the user may set the output power by adjusting the size of each device, and then run a compactor to produce a compact cell in a desired shape.

Layout compaction consists of rearranging the geometries of a design with limited topological modifications to generate a layout that satisfies all the design rules and is as compact as possible. This is a very tedious and time consuming task that is preferably off-loaded to automated procedures. Furthermore, conventional algorithms cannot yet match the quality of layouts produced by human designers.

The most general form of compaction involves moving the geometries of the design in the horizontal and vertical coordinates simultaneously. The problem of generating a correct layout of minimum area with this set of moves has been shown to belong to the class of NP-hard problems. For this reason, most of the compactors proposed in the past decompose this 2-dimensional problem into an alternating sequence of independent one-dimensional compaction steps [4, 37]. However, one-dimensional algorithms are unable to recognize situations where small movements of a component in one dimension could significantly affect the amount of compaction possible in the perpendicular dimension.

Several two-dimensional approaches have been published. One approach starts from a totally collapsed layout and removes the spacing violations one by one [81]. Another approach transforms the compaction problem to an integer programming problem [45]. A third method is based on simulated annealing techniques. All these methods have exponential complexity on the number of components in the layout and are very expensive in CPU time [13].

A new algorithm for two-dimensional compaction has been developed, which combines a generalized one-dimensional compaction procedure with sophisticated lateral movements of the elements to be compacted. These lateral movements will 'shake' the individual components into a more densely packed arrangement. The technique bears a strong similarity to the technique of 'zone-refining' used in the purification process of crystal ingots [72]. Individual circuit components or small clusters of components are peeled off row by row from the precompacted layout, moved across an open zone, and reassembled at the other end of this zone in a denser configuration. In this process, both coordinates of the moved components are altered and jogs may be introduced in the connecting wires between them to produce additional flexibility for placing components into optimal positions. The constraint graphs in both the x- and y-direction are used and updated concurrently.

The current implementation of compaction by zone-refining can handle NMOS or CMOS technology. For efficient use of area, it can *merge/overlap* contacts or wires belonging to the same electrical node. It can also bend wires automatically, and can shorten the wire length during compaction.

Since the zone-refining process requires a lot of local movements, the constraint graph modification takes most of the CPU time. Data structures and mechanisms are developed to handle several metal layers, and to reduce the computing time. Solving the longest path problem on constraint graphs takes usually less than 3 percents of the total run-time, because only a small number of clustered components are moved at once. An efficient constraint graph generation is possible by an efficient search of pairs of components to be constrained. The most efficient and natural way of neighbor search is to access components from their x/y-coordinates. In the latest version of implementation, all the components are attached on space cells divided by coarse grid. With this grid approach, searching time is bounded by a constant and sorting time is linear to the number of components provided components have finite size; whereas in general, sorting time is $O(\ n\ \log n\ )$ and searching time is $O(\ \log n\ )$ where $n$ is the number of components considered. As a consequence of the new data structure, run-time has been reduced to 1/5 to 1/10 of that of the first implementation.

The pitch-matching of cells is important in standard cell design or when connection by abutment is desirable. If each cell is independently compacted, a large routing area may be needed between the cells. By adding more constraints during compaction this routing area can be reduced. The current implementation of zone-refining can generate layouts with desired width or height; however, placing terminals at proper positions to make connections by abutment is a more difficult problem. This problem can be avoided by compacting all the cells at once. More study is necessary to handle general cases.

Merging and bending wires are essential for an efficient use of area. Merging of equi-potential elements may save 20 to 30 percent of area; however, this may generate partially overlapped components which must be removed before manufacturing. Bending wires by automatic jog-generation may save 5 to 10 percent of area; however, this may result in unnecessary "U-shape" wires which must be removed to obtain good electrical performance.

In the current implementation, all the connections are kept through the zone-refining process by stretching or shrinking wires. An alternate approach for moving elements is *rip-up and re-routing*, in which all the wires connected to the elements to be moved are removed, and then re-routed after the desired movements. In this approach, the re-routing must be guaranteed by calculating the exact routing space before each movement. The exact calculation of the routing space is not a simple operation [63], and more study is necessary.

Future work includes *pitch-matching* of separately compacted cells in hierarchical design environment, *clean-up routine* to remove "U-shape" wires and partially merged contacts, *exploring other algorithms of placements of elements* in the zone, handling non-trivial design rules, and *extensions* to handle 45 degree wires.

2.

# Chapter 2 Detailed Routing Based on Incremental Routing Modifications

Channel routers have been most successful in performing the detailed routing task for a variety of design styles. However, for the general macro-cell or building-block design style, the routing regions cannot be defined so that a channel router can route all of them unless some restrictive assumptions are made on the shape of the cells and on the placement technique used [35, 43]. In our approach to macro-cell placement and routing, we have to place and route cells whose bounding boxes are not rectangles and we do not wish to enforce slicing structures to capitalize on the power of simulated annealing algorithms. Hence, we are confronted with the problem of routing regions with irregular boundaries and non convex shapes which are delineated by polygons. This approach would not be feasible if a powerful two-layer two-dimensional router were not available. This chapter describes an algorithm and its implementation that allows a very efficient routing of any two-dimensional region with pins on or inside the boundaries of the region. Because of the interest of designers in modifying incrementally their design and in performing part of the routing manually, we have developed the router to take care of partially routed regions.

This chapter is organized as follows. Previous work is described in Section 2.1. The problem formulation and general approach are described in Section 2.2. Section 2.3 outlines the main ideas of our approach and the description of algorithms. The computational complexity of the algorithm is analyzed in Section 2.4. Section 2.5 describes the partitioning of large problems. Section 2.6 contains other variations of the basic algorithm used for channel routing and special cases. In Section 2.7, routing results of switchbox and channel examples are evaluated. In Section 2.8, the sensitivity of the algorithm on parameters is discussed. Finally, summary is given in Section 2.9.

## 2.1. PREVIOUS WORK

A brief survey of detailed routers published can be described as follows.

The first detailed router reported in CAD of IC is probably the maze router studied by Lee in 1961 [55]. The router implements connections net by net. This router starts a search from a starting terminal and the frontier of the search-region expands like a wave front until it hits the target terminal. The strongest point of this router is that it can always find a path if there exists one. The weak points are that it is rather slow and that the routing results strongly depend on the routing order. This approach can be extended to a line-search algorithm [29, 30].

The second routers are greedy. The greedy approach is suggested by Rivest and Fiduccia [75] and modified by other authors [50, 62, 94]. In this approach, connections are implemented column by column usually from the left-most column to the right and the routing results may depend on the scan direction.

The third approaches are channel routers using track/row assignments [73, 110]. These routers assign each net to a row/track and then implement vertical connections. Most of the routers in this category assume that routing area is a rectangle and that terminals are on top and bottom edges of the rectangle. Some routers [73] are extended to handle terminals on three edges of the routing area.

The fourth approach that gives good results is hierarchical routing [11]. In this approach, the routing area is divided into several subcells and global paths between the subcells are determined for all the nets that have terminals in more than one sub-cell, then each subcell is divided again and routing paths are determined at a more detailed level. This hierarchical dividing and routing continues until all the interconnections have been explicitly determined. The major drawback of this approach is that at each stage the decision is made ignoring the information contained at lower stages.

The fifth approach is rule-based. Several rules can direct the router [65]. A force directed router can also be seen as a rule-based system [26]. In this case, attracting and repulsing rules decide the routing path.

The sixth approach is derived from knowledge-based expert systems [40]. In this approach, hundreds of routing rules carefully chosen determine where the interconnections will be placed. However, trying not to impede complete connections takes so much run-time that this approach can handle only small routing problems.

The final type is rip-up and rerouting. Since the proposed router belongs to this class, rerouting methods are described in detail later in this section.

Some of these are switchbox routers [25, 33, 40, 62, 65], but the quality of the results or the running time needed by the routers have not been fully satisfactory. The most successful switchbox router, Weaver [40, 41], is based on Knowledge-Based Expert Systems (KBES). It includes many experts and several hundred rules to guide the routing. For this reason, running time is very long and the time needed to develop the program was also very long.

Table 2.1 summarizes well known detailed routers into a few types and compares them in several respects - routing area, directionality in routing, CPU time, capability of modifications. Routing area is the type of routing region the corresponding router can handle. A channel is the most restricted form of routing area which has fixed pins (terminals) on top and bottom edges of the region. A switchbox may have pins on all four edges of the rectangular routing area. Rectilinear polygonal region is the most general form of routing region and there may be obstacles in it. Directionality shows how flexible to use each layer in either direction (horizontal or vertical) to make necessary connections. CPU time shows efficiency of the algorithms. Dynamic modification is the capability to modify existing connections.

| Router types | routing area | directionality on each layer | CPU time | dynamic modifications |
|---|---|---|---|---|
| Maze (Lee 61, Hightower 69, Hsu 82) | poly. | flex. | mod. | no |
| Greedy (Rivest 82, Hamachi 84, Hsieh 85) | box | poor-mod. | fast | no |
| VCG based (Yoshimura 82, YACR 85) | chan. | poor-mod. | fast | no |
| Hierarchical (Burstein 83) | box | poor-mod. | fast | no |
| Rule based (Marek-Sadowska 85, Hasan 87) | box | poor-mod. | mod. | no |
| KBES (Joobbani 85) | box | flex. | slow | no |
| Reroute (LAMBDA 83, MIGHTY 86) | poly. | flex. | mod. | yes |

| | |
|---|---|
| chan. : | channel |
| box : | switchbox |
| poly. : | rectilinear polygonal region |
| flex. : | flexible |
| mod. : | moderate |

Table 2.1 : Types of detailed routers

In all cases except the last row in the table, the routers tried to predict where congested regions were and which nets were difficult to connect and gave priorities based on this estimate. Of course, the estimate is only approximate and problems may develop anyway and force the router to use additional rows and/or columns. In addition, some of the interconnections may be unnecessarily far from being "optimal" with respect to their length and the number of vias.

Only a few rerouting algorithms have been reported, mostly for the routing of printed wiring boards. In 1974, F. Rubin [78] suggested an iterative process where all failed connections are routed allowing crossings with a substantial scoring penalty and then the connections with the greatest number of crossing are ripped up. Later, this approach has been extended using penalty functions by R. Linsker [60]. In 1977, P. Agrawal and M. Breuer [2] suggested a backtracking algorithm which can possibly enumerate all the alternate connections for each net. However, complete backtracking to find an optimal routing is so complicated that it can be used only for small-sized problems. In 1982, W. Dees and P. Karger [16] reviewed rip-up and rerouting techniques and suggested two types of modifications - rip-up and shove-aside. Shirakawa et al. [91] has reported a rerouting scheme for single-layer printed wiring board with two-pin nets only. Recently Suzuki et al. [95] reported that the Lee algorithm [55] is most useful and powerful when applied to interactive rip-up and reroute. However, no published results are available, to the best of our knowledge, on automatic general-routing-modification and rerouting for two-layer detailed routing with polynomial worst-case complexity. Most of the previous rerouting approaches have one or more of the following weaknesses.

1 Rip-up and rerouting is the only modification routine while relocating part of existing connections can frequently solve the problem [60, 78, 91, 95].

2 Modification is tried only after a completely blocked net appears. But there may be many "bad" quality connections at this time [16, 95].

3 Modification is interactive, and human intervention is necessary [95].

4 Rip-up techniques can be potentially uncontrollable when considering computational complexity [ 2, 16].

5 The focus is on the routing of two pin nets on a single layer of printed wiring board [2, 16, 91].

## 2.2. FORMULATION OF THE PROBLEM AND GENERAL APPROACH

The new router is based on an incremental routing technique that favors nets with several pins widely scattered in the routing region. The novel part of the algorithm is the modification of connections already made for the nets in the routing region to allow blocked or "bad" quality connections to find better solutions. These modifications are of two kinds: a weak modification step pushes aside existing connections without removing them to find a shorter path or to make room for a blocked connection, and a strong modification step, invoked when weak modification fails, removes blocking connections. Thus, we have concentrated our attention on developing a router which is an "expert" in modifying and rerouting a partial solution if problems occur. Note that in our opinion, the most distinguished difference between a human expert and existing routing tools is the ability of rerouting or modifying the existing connections.

The router described in this chapter implements a general incremental routing modification strategy for rectilinear polygonal routing regions. This router has obtained excellent results on a number of test cases, outperforming not only existing two dimensional routers but even channel routers when applied to "standard" channels. Hence, Mighty can be considered a general purpose two layer router, being able to route any irregular regions with floating as well as fixed pins on the boundaries of the region. Mighty works on symbolic grids and is an integral part of MOSAICO [7], a general macro-cell floor planning, placement, and routing system where it is used to route L-shaped and staircase-shaped channels.

The terms used throughout this chapter are defined as follows.

**channel or routing area:** A rectagonal (rectilinear polygonal) region between circuit blocks that can be used for interconnections (for example, see Fig. 2.1).

**pin:** A terminal of a cell facing the routing area, which is to be connected to a set of other terminals. All pins are on grid points because the router works on symbolic grids.

**net:** A set of pins to be connected and their associated connections.

**component:** A set of pins and wire-segments of a net which have been interconnected. Each unconnected pin is a trivial component.

**horizontal (vertical) segment:** A piece of wire running horizontally (vertically) on a layer described by two end points. Nets are connected by wire segments.

**track** or **row:** The symbolic routing area in the horizontal direction.

**column:** The symbolic routing area in the vertical direction.

**contact** or **via:** An interconnection of two segments on two different layers, placed at the row and column location the segments have in common.

**path:** A set of vias and segments implementing the interconnection between two components of a net.

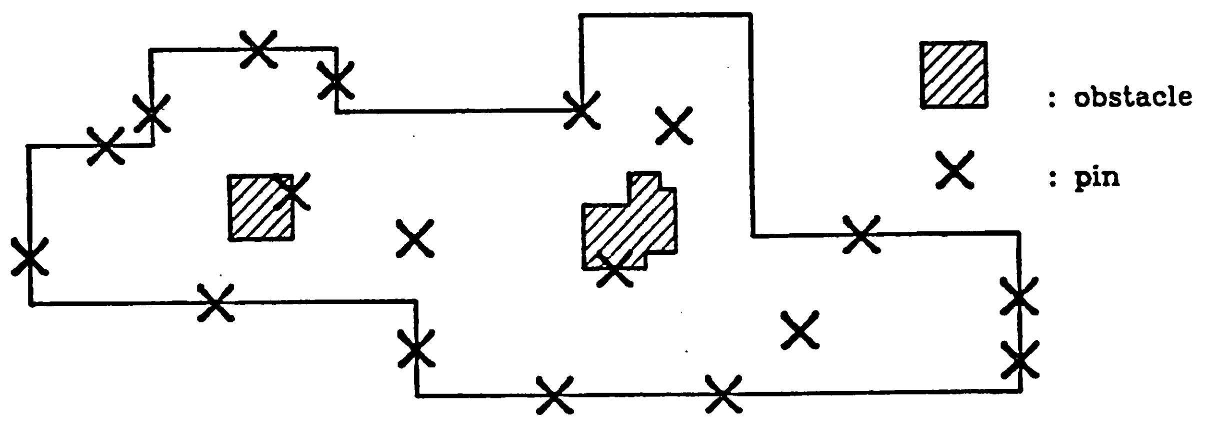


**Figure 2.1.** *Routing area with obstacles.*

The routing area is a rectilinear polygon with two layers of interconnection as shown in Fig. 2.1. Pins may be inside or on the boundaries and some of them may be floating on the boundaries. There may be obstacles of any shape and size in the routing region.

The problem we have to solve is:

> Connect all the components of each net in the routing region such that each available grid point on each layer is used by at most one net.

The primary objective is to complete the connections using minimum routing area. The secondary objective is to minimize the number of vias and the wire length of each net. Other factors such as preference of one layer to the other or consideration of coupling with neighboring nets can be included by appropriately setting cost parameters.

## 2.3. THE ALGORITHM

The main algorithm is described in Section 2.3.2. The detailed algorithms are described in the following sections. In the description of the algorithms, all the parameters begin with a capital letter.

### 2.3.1. Data Representation

Efficiency was the major concern for the current implementation. To represent the routable area, a grid-map has been used for each layer. A grid-map contains all available symbolic grid points on a layer. Pins are also marked in another grid-map for easy access of their locations during modification of nets. Paths are kept as a linked list of segments for each net.

### 2.3.2. The Basic Algorithm

The overall flow of the main algorithm is presented in this section.

The basic algorithm consists of four main parts: a path finder that searches for a shortest (minimum cost) path among components, a path confirmer that implements a particular path proposed by the path finder, a weak modifier that relocates existing interconnections to find a shorter path when the existing one is "far" from optimal, and a strong modifier that removes some connections from the routing region to allow the completion of a blocked path.

As a pre-processing step, the algorithm extends all the pins on the boundaries of the region inside by one unit to avoid possible connections along the boundaries of the routing region. Then the path-finding phase begins. The nets are processed in the order they are entered. From each pin of the net we start a maze router search to find the minimum-cost path that interconnects any two pins of the net, assuming that all the other nets are not present. The cost used to direct the search is based on the model in which a layer is mainly used for horizontal interconnections and the other for vertical interconnections. For example, extension of a path in the vertical direction on the horizontal layer, while possible, is penalized. Changing a layer is also penalized to minimize the number of vias. As soon as a path connecting two pins of the net is found the search is stopped. The path connecting the two pins is recorded on an ordered list organized in increasing cost. Note that the path is not finalized yet.

When all the nets have been processed by the path-finder, the path-confirmer takes over. The paths on the ordered list are popped and examined. If the path is feasible, i.e. if no other interconnection has occupied the same locations as the present path, the path is implemented. Otherwise the path finder is invoked again and a feasible minimum-cost path connecting any two unconnected components of the net is sought. Note that when this path is looked for, the interconnections already laid out are taken into consideration. Because of the paths that have been implemented before, the new path of the net under examination may be far from the optimal solution. One of the main ideas of our algorithm is to make sure that poor intermediate solutions are not accepted. Hence, if the new path is found to be unacceptable, the modification phase is entered.

It is important to choose a good criterion to classify paths as acceptable or unacceptable. The criterion we selected is based on the number of "bends" that the path has to include. In particular, we use the following definitions.

**floating segment:** a segment which is not connected to a pin or a pseudo pin (see Fig. 2.2).

**good path:** a path which has less than $n$ floating segments. ($n$ = 3 to 5 seems to be a good range to use.)

**bad path:** a path which is not a good path.

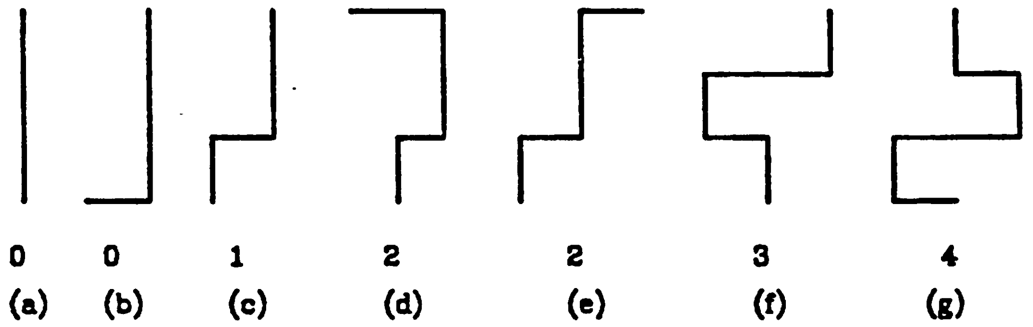


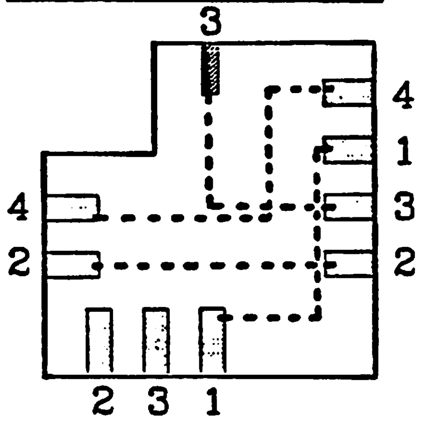
**Figure 2.2.** *The number of floating segments of several cases. With $n = 3$, (a) to (e) are good paths, while (f) and (g) are bad paths.*

Different criterion for good path can be used. The above is chosen since it is simple and can be checked efficiently.

If a path is found and it is a good path, then it is scheduled again. Otherwise, the modification phase is entered. The weak modifier is first called to push other nets around to make a good feasible connection possible for the net under consideration. If no solution is found, then the strong modification phase is entered. Each of these implies a modification of the existing interconnections - some interconnections have to be either pushed away or removed. In both phases, a variety of alternative "good" interconnections for the net under consideration are examined, the cost of each solution is computed considering the cost of relocating or removing existing nets, and the minimum-cost solution is selected. If such solution is above a preset limit, then the router ends its search with a failure. The process is iterated until either a solution is found or a failure is reported.

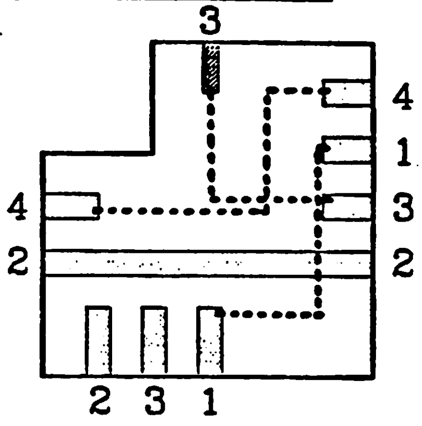
To show the schedule and routing order, a simple example is shown in Fig. 2.3. The cost of a via is assumed to be 30, while the cost of a unit length of wire is 2 ( if in preferred direction ) or 50 ( if in non-preferred direction ). Since the cost of a via is large, straight connections are made first and paths with two vias are implemented last. In Fig 2.3.(a), the minimum-cost path of each net is found as shown by the dotted lines, and all the three nets are scheduled in the increasing order of their costs. Note that net 2 and 3 have three components and that shortest paths connecting any two of the three components are found and scheduled. In (b), net 2 is popped and its path is implemented. In (c), net 2 has two components, so a shortest path which interconnects the bottom pin with the partially routed net is found and net 2 is re-scheduled. In (d), a path of net 3 has been implemented. In (e), net 3 is re-scheduled with a shortest path connecting its two components. In (f), net 2 is completely routed and three nets are in the queue. In (g), net 3 is completely routed and two nets are in the queue. In (h), net 1 is completely routed. However, the path of net 4 is completely blocked in the figure. Simple rip-up and reroute does not help in this case as shown in Fig. 2.3 (i) and (j). In (i), the blocking connection (net 3) has been removed, and net 4 is routed as shown in (j). Now net 3 is blocked and the pin on the right-side edge can not be connected to other pins of net 3. Since Mighty uses weak modifications, the via of net 3 is moved by one grid space to the right and a path for net 4 is found, as shown in (k). In (l), the routing is completed and the queue is empty.

| net | 2 | 3 | 1 | 4 |
|---|---|---|---|---|
| cost | 8 | 38 | 70 | 72 |



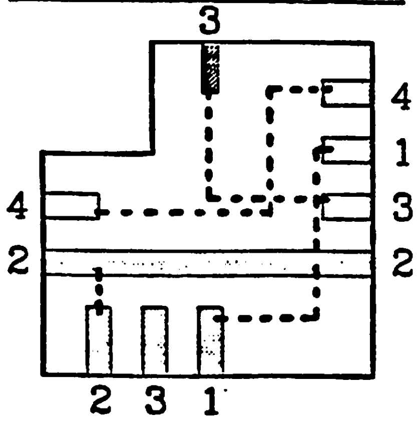
(a)

| net | 3 | 1 | 4 |
|---|---|---|---|
| cost | 38 | 70 | 72 |



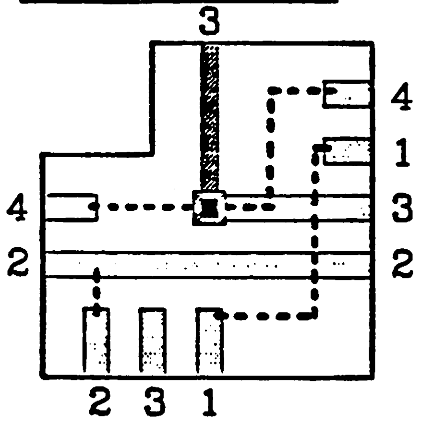
(b)

| net | 3 | 2 | 1 | 4 |
|---|---|---|---|---|
| cost | 38 | 50 | 70 | 72 |



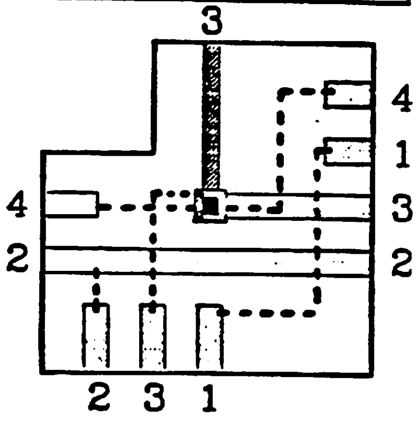
(c)

| net | 2 | 1 | 4 |
|---|---|---|---|
| cost | 50 | 70 | 72 |



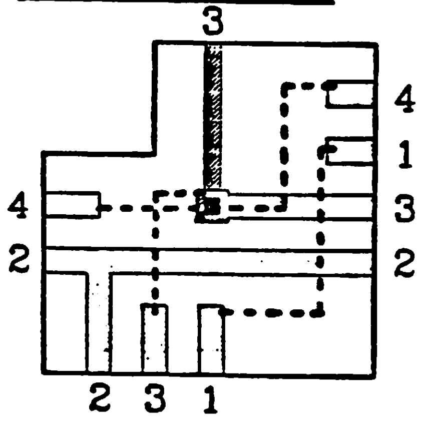
(d)

| net | 2 | 3 | 1 | 4 |
|---|---|---|---|---|
| cost | 50 | 66 | 70 | 72 |



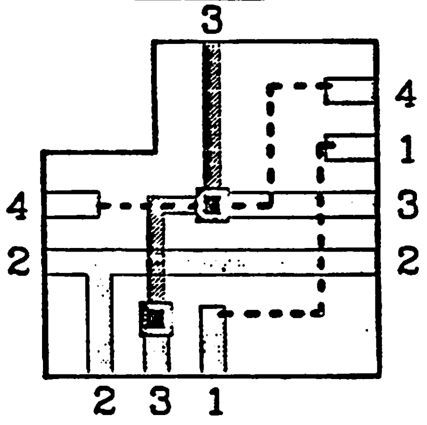
(e)

| net | 3 | 1 | 4 |
|---|---|---|---|
| cost | 66 | 70 | 72 |



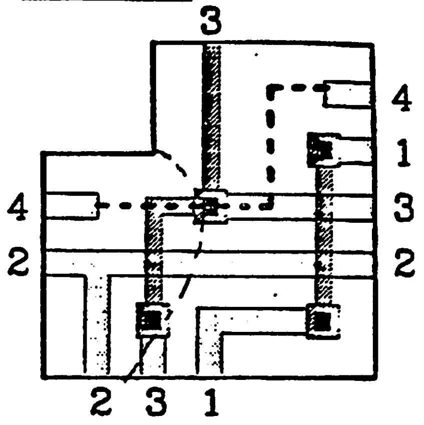
(f)

| net | 1 | 4 |
|---|---|---|
| cost | 70 | 72 |

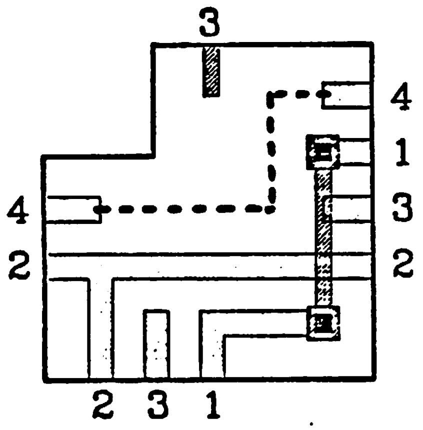


(g)

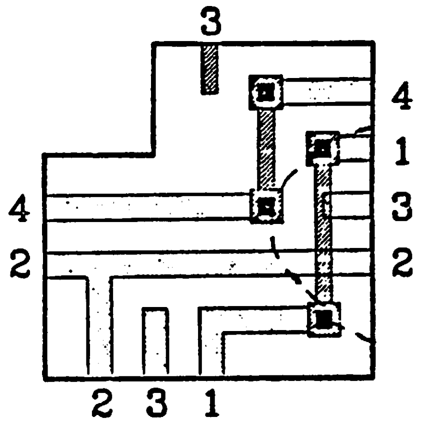
| net | 4 |
|---|---|
| cost | 72 |



(h)

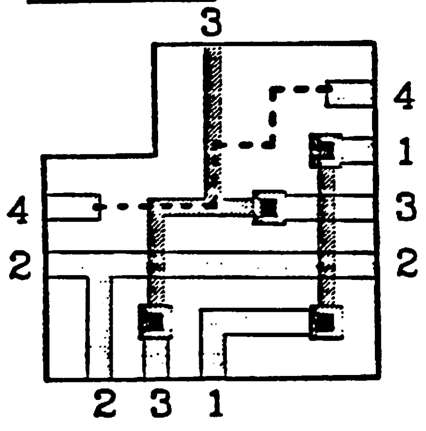


(i)

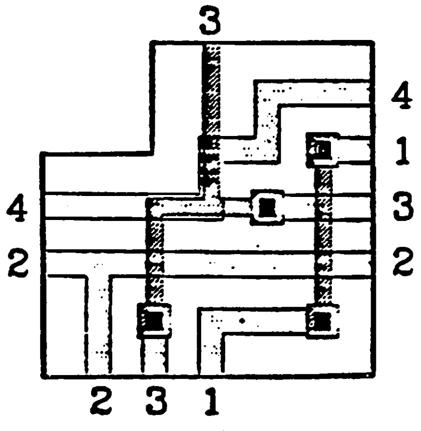


(j)

| net | 4 |
|---|---|
| cost | 108 |



(k)



(l)

**Figure 2.3.** *An example to show the routing order and the schedule, with shortest paths and their costs.*

### 2.3.2.1. Overall algorithm

The precise algorithm is as follows.

**mighty()**

```
        /* Mark obstacles and pre-routed nets */
        pre_processing;

        /* Initially, schedule all nets */
        for( $i$ = 1; $i$ $\leqslant$ num_nets; $i$++ )
        {
                if( net $i$ has more than one component )
                {
                        find a shortest path, path[$i$], connecting
                                any two components of net $i$;
                        if( path[$i$] != $\phi$ )
                        {
                                Schedule net $i$ with path[$i$] in increasing
                                        order of the path length;
                        }
                        else
                        /* There exists a net that can not be connected
                                without changing the routing area */
                        {
                                report failure;
                                exit;
                        }
                }
        }


        /* Main routing loop */
        while( schedule is not empty )
        {
                $i$ = the first net in the schedule;
                if( path[$i$] is feasible )
                {
                        /* implement path[$i$] */
                        confirm_path( $i$ );
                }

                if( net $i$ has more than one component )
                {
                        find a shortest path path[$i$] connecting
                                any two components of net $i$;
                        if( path[$i$] = $\phi$ OR (path[$i$] is a bad path) )
                path[i] = weak_modification( $i$ , path[$i$] );
            if( path[$i$] != $\phi$ )
            {
                Schedule net $i$ with path[$i$] in increasing
                    order of the path length;
            }
            else
            {
                strong_modification( $i$ );
            }
        }
    }

    /* All nets have been connected. Vias and wire-length can be reduced further,
       and metal maximization can be done if necessary */
    post_processing();
```

Now all the important functions used in the above algorithm will be described.

### 2.3.3. Finding a Shortest Path

The routine that finds a minimum-cost path is a modification of Lee's algorithm [55]. The major extension is that our algorithm can find a minimum-cost path connecting any two of the components of each net, i.e. the minimum-cost path that can reduce the number of components by one. Hence it is not necessary to specify two points (or two components) before finding a path, but the path-finder makes use of all the existing pins and connections.

Since the search for the minimum-cost path is started from every component of the net, component number as well as the cost has to be propagated. In the data structure, every grid point $p$ of the routing area can contain two integers, $p$.net and $p$.component. If $p$ is a point of a confirmed path, $p$.net contains the owner net. Otherwise, $p$.net is used to store the negative value of the cost during find_path. All the connected pins and wires of a net belong to the same component.

The algorithm consists of two parts, schedule and search. At the beginning, the grid points belonging to a pin or implemented connections of the net being processed are scheduled with cost value 1. Then, the search starts from the first grid point $p_f$ in the schedule, in which the cost value of the point is marked and all available neighbor grid points $p_n$ are scheduled with cost = cost of $p_f$ + *distance* $(p_f, p_n)$, where *distance* is the incremental cost depending on the direction, the layer, and change of layers. The frontiers of searched grid points are expanded like a wavefront from all the components of the net. It should be pointed out that the path found by this approach is frequently shorter than a minimum-path between two specified points, since the search is triggered from all the confirmed wire-segments as well as pins of the net. When cost = $c$, the algorithm marks all the free grid points reachable at cost $c$ from any component of a net $i$. We set all the cost parameters as even numbers, so that we can get an integer even after dividing the cost by 2 (see the algorithm description). The desired result comes from the following theorem.

**Theorem I.** If there exists a path of length less than Max_cost, then the find_path algorithm finds a minimum-cost path connecting two components of net $i$.

**proof:** When cost is equal to $c$, the algorithm has marked all free grid points reachable from each component of net $i$ with cost $<c$ and part of the grid points reachable with cost $= c$. If two different components (wave fronts) meet at a grid point, then a path is found and the cost of the path connecting the two components is calculated. Let *touchcost* be the half of the cost of the path found. If a path is found at *touchcost* $= c$ for the first time, then there does not exist a path with cost $\leq 2\,(c - 1)$. (Otherwise we had to find the path at *touchcost* $= c - 1$.) Since all the cost parameters are even numbers, the minimum possible cost is $2\,c$, and the path found is an optimal one. If

there does not exist a path of length less than Max_cost, then the algorithm can not find a path until *cost* is increased to Max_cost/2 and the algorithm returns nil ($\phi$).

### 2.3.3.1. Algorithm Find_path

This algorithm finds a minimum-cost path connecting any two components of a net. In the algorithm, the *searchq* contains all the scheduled grid points in increasing order of costs, with their cost values and component numbers.

```
find_path( i )

          /* Initialization and schedule */
          cost = 1;
          schedule all the pins and existing connections of net i in searchq;

          /* Search for a path.
             Max_cost is a user defined parameter */
          while( cost < Max_cost/2 )
          {
                if( searchq has not a point p of cost )
                {
                      cost = cost + 1;
          continue;
                }
                else
                {
                      /* get the first point p from searchq */
                      pop p with component comp from the searchq;

                      if( point p is searched already with less cost
                            by the same component )
          continue;
                      }
                      else if( point p is searched already
                            by a different component )
                      /* A minimum-cost path has been found */
                      {
                            find the path, path[i ], by backtracing;
                            return( path[i ] );
                      }
                      else
                      {
```

```
                    /* Mark the point $p$ as searched */
                    if( $p$.net = $\phi$ )
                         $p$.net = - cost;
                    $p$.component = *comp*;
                    /* schedule its neighbors */
                    for( each neighboring point $s$ of point $p$ )
                    {
                         if( $s$.net = $\phi$ )
                         {
                              /* schedule $s$ */
                              add $s$ in searchq with *comp*
                                   and cost + distance($p$,$s$);
                         }
                         /* if $s$ is searched by another component,
                            then a path is found */
                         else if( $s$.component $\neq$ $p$.component
                              AND $s$.component $\neq \phi$ )
                         {
                              /* A path is found which may or may not
                                 be a shortest path */
                              touchcost = (cost of $p$ + distance($p$,$s$) +
                                   cost of $s$)/2;
                              /* schedule $s$ */
                              add $s$ in searchq with *comp*
                                   and touchcost;
                         }
                    }
               }
          }
     }

     /* Path does not exist */
     return( $\phi$ );
```

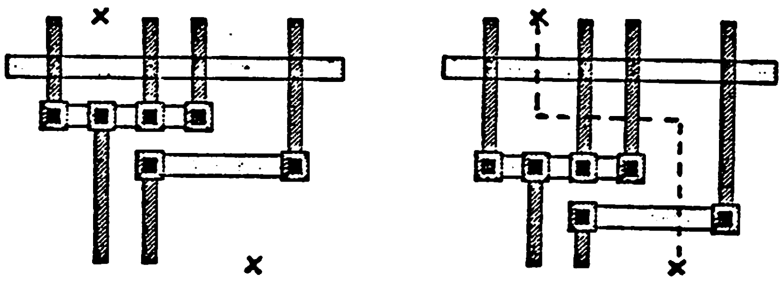
## 2.3.4. Confirmation of a Path

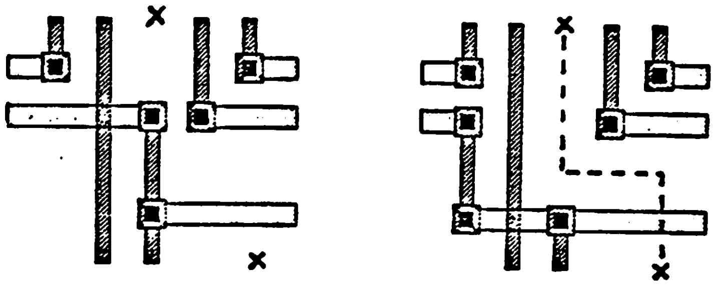
In the basic algorithm, when net $i$ is processed according to the schedule, if all the points of path[$i$] are available (they have not been assigned to another net) *confirm_path* is called. The routine confirm_path( $i$ ) implements the path of net $i$ by assigning the path[$i$] to net $i$, and by merging the two components just connected into a big component.

### 2.3.5. Weak Modification

As discussed above, during weak modifications, part of the existing paths can be relocated while maintaining all the existing connections. After a minimum-cost path is found, the path is evaluated as either a good path or a bad path according to the number of floating segments (see Fig. 2.2). If the path is good, it is scheduled and processed. Otherwise, weak modifications are attempted to find a better solution.

We use three types of weak modifications (*unit_push*, *jump_push* and *point_push* ), as shown in Fig. 2.4. Each of these modifications is in four directions( down, up, left, and right ). For the sake of simplicity, only downward modifications are shown in the figures.


*(a) unitpush( down )*


*(b) jumppush( down )*

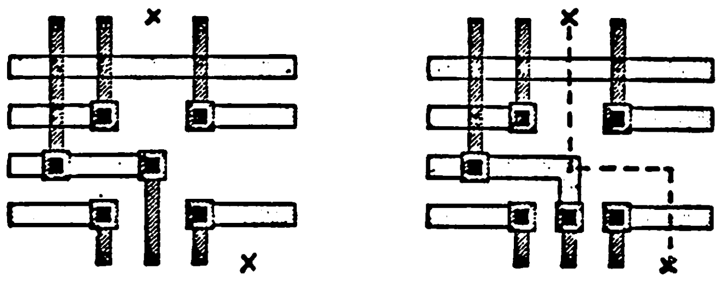

*(c) pointpush( down )*

**Figure 2.4.** *Examples of weak modifications*

In the algorithm, described below, *weak_modification* calls *modify*, and *modify* calls *tryunitdown*, etc. The routine *tryunitdown* finds a blockage, and if the blockage is modifiable, calls *unitdown* which actually tries to push the blockage a unit grid space downwards and returns YES if successful.

The routines *jumppush* and *pointpush* are implemented in a similar way, and their algorithms are not included. *Jumppush* differs from *unitpush* in that it can move segments over the segments of other nets. *Pointpush* is similar to *unitpush*, but is more powerful in that it can change the layer of a segment at the blocked point if it is necessary. *Unitpush* and *pointpush* are recursive so that they can push a stack of nets (for example, two nets are modified in Fig. 2.4(a)).

One of the difficult issues in rerouting or routing-modification is how to prevent oscillations. To prevent oscillation in weak modifications( push/jump up and down or left and right can be repeated forever ), we used history. We ordered the twelve types of weak modifications as shown in *modify*. If one type of modification did not solve the problem, then we make sure that the same modification is not called again unless one of the related nets is rerouted by a strong modification or one more connection is made. The element on the $i$-th row and the $j$-th column of the history matrix contains the type of weak modification to be tried if net $i$ calls weak modification to push net $j$. When one more connection is made in *confirm_path* for net $i$ or when net $i$ is affected during *strong_modification*, all the elements on the $i$-th row or $i$-th column are reset so that all types of weak modification can be tried later.

```
weak_modification( i, path[i] )

        /* i is a net, path[i] is a shortest path of net i */
        /* Save current paths */
        for( j = 1; j ≤ num_nets; j++ )
            savedpath[j] = path[j];
```

```
/* find_cost returns the sum of costs of path[1..num_nets] */
totalcost = find_cost();
while( (type = modify( i )) ≠ φ )
{
      path[i] = find_path( i );
      if( path[i] != φ )
      {
            newcost = find_cost();
            if( newcost < totalcost )
            {
                  /* better solution is found */
                  totalcost = newcost;
                  for( j = 1; j ≤num_nets; j++ )
                        savedpath[j] = path[j];
            }
            /* if path[i] is good then break,
               otherwise try more modifications. */
            if( path[i] is a good path )
break;
      }
}

/* Choose the best path found */
for( j = 1; j ≤num_nets; j++ )
      path[j] = savedpath[j];

/* reschedule affected nets */
for( each modified net k )
      reschedule( k );

return( path[i] );
```

**modify( i )**

```
find the two closest blocked pins p1 and p2 of net i;

if( tryunitdown( p1, p2, i ) )
      return( Unitdown = 1 );
if( tryunitup( p1, p2, i ) )
      return( Unitup = 2 );
if( tryunitleft( p1, p2, i ) )
      return( Unitleft = 3 );
if( tryunitright( p1, p2, i ) )
      return( Unitright = 4 );
if( tryjumpdown( p1, p2, i ) )
      return( Jumpdown = 5 );
if( tryjumpup( p1, p2, i ) )
      return( Jumpup = 6 );
if( tryjumpleft( p1, p2, i ) )
```

```
        return( Jumpleft = 7 );
    if( tryjumpright( p1, p2, i ) )
        return( Jumpright = 8 );
    if( trypointdown( p1, p2, i ) )
        return( Pointdown = 9 );
    if( trypointup( p1, p2, i ) )
        return( Pointup = 10 );
    if( trypointleft( p1, p2, i ) )
        return( Pointleft = 11 );
    if( trypointright( p1, p2, i ) )
        return( Pointright = 12 );
    return( φ);
```

**tryunitdown( p1, p2, *i* )**

```
    /* p1 and p2 are blocked pins of net i.
       If possible, move a blocking segment one grid space
       in the downward direction and update history */

    /* find a blockage */
    scan the vertical layer of the routing region downward from p1
        until a blockage b is found;

    /* Let the location of the blockage be (x, y) */
    if( b is a modifiable net AND history[i, b] ≤ Unitdown )
    {
        if( unitdown( i, b, x, y ) )
            /* The downward push was successful */
            history[i, b] = Unitdown;
        else
            /* Same type of push can not be repeated */
            history[i, b] = Unitdown + 1;
    }
    repeat the above for pin p2;
```

**unitdown( *i*, *b*, x, y )**

```
    /* i is the blocked net,
       b is the blocking net,
       (x, y) is the grid point on which the blockage is found */

    find the interval( from x1 to x2) of net b to be pushed;
    if( there is a pin of net b on the interval )
        return( NO );
```

```
find the blockages that prevent the move of the segments of net b
     on the interval by one grid space downward;
if( there are no blockages )
{
     move the segments of net b on the
          interval by one grid space downward;
     return( YES );
}
else if( there is a single modifiable net n in the new blockage )
{
     /* push the new blockage first */
     if( unitdown( b , n , x, y+1 ) )
     return( YES );
}
else
     return( NO );
```

### 2.3.6. Strong Modification

When weak modification fails to find a path, then strong modification is called. If strong modification fails to find a path within the maximum cost limit, the router reports failure and exits. Since the report includes the locations of failure, more space can be added at the right position to complete the routing.

During a strong modification, we remove part of existing connections so that the blocked net can be connected. First a minimum-cost rip-up path is found. Then all the connections of the nets in the path are removed, and the blocked pins are connected. All the nets disconnected during the rip-up process are rescheduled.

The rip-up cost consists of the number of nets affected, the difficulty values of the nets, the length, and the number of vias in the path. The difficulty values are zero at the beginning. If connections of a net are removed during strong modification, the difficulty value of the net is increased by Delta which is a user defined parameter. This makes the same net unlikely to be removed repeatedly, prevents oscillations, and allows to try a new path when strong modification is called several times to connect the same pair of blocked pins.

To find an optimal rip-up path between the pins of a blocked net, we evaluate the rip-up cost of all the straight, L-shaped and Z-shaped paths, and choose the minimum-cost one.

**strong_modification( *i* )**

```
        /* find a pair of blocked pins of net i */
        find two closest pins p1 and p2 of net i
              which are not in the same component;

        find a minimum-cost rip-up path connecting p1 and p2;
        /* Limit_cost is a user defined parameter */
        if( rip-up cost ⩾ Limit_cost )
        {
              /* rip-up cost is too large */
              report_failure();
              exit();
        }
        else
        {
              /* remove all the connections in rip-up-path.
                 reschedule all the affected nets. */
              for( each net k in the rip-up path )
              {
                    difficulty[k] = difficulty[k] + Delta;
                    remove all the connections of net k;

                    /* reschedule with zero cost and nil path */
                    reschedule( k );
                    clear_history( k );
              }
              while( net i has more than one component )
              {
                    find a shortest path[i] connecting any two
                          components of net i;
                    if( path[i] != φ )
                          confirm_path( i );
                    else
                          strong_modification( i );
              }
              /* reset the i-th row and i-th column of history
                    which is used in weak modification */
              clear_history( i );
        }
```

## 2.4. COMPLEXITY OF THE ALGORITHM

In this section, it is proved that the algorithm given in Section 2.3 terminates and that its run-time is bounded by a polynomial in the size of the input. Let the number of pins be $p$, and let the number of nets be $k$. Let $L$ be the complexity of finding a minimum-cost path within the routing area. If there are $m$ rows and $n$ columns, then clearly $L = O(m\ n)$. Furthermore, without loss of generality, we assume that $m \leqslant n$ so that $O(m + n) = O(n)$.

To find the computing complexity when weak modifications are used, a "history" of the weak modifications is used. History is a $k$ by $k$ matrix. The element on $i$-th row and $j$-th column of the matrix has a number which represents the type of weak modification to be tried if net $i$ calls weak modification to relocate net $j$. The 12 modifications are tried in the order listed in the algorithm in the previous section. If the current weak modification fails to modify, then the next in the order list is tried.

Keeping the information of strong modification is a little simpler. Each net has a difficulty value which is increased if the net is ripped-up during a strong modification. Since this difficulty value monotonically increases whenever a strong modification is called, the algorithm must terminate after a finite number of operations, since Limit_cost is finite.

Theorem II (a) and (b) explore cases of successful routing, and Theorem II (c) holds whether the routing is successful or not.

First we prove four intermediate results, and then state the theorem.

**Lemma I.** If strong modification is not called, $(p - k)$ connections complete the routing.

**proof:** There are $p$ unconnected pins initially, hence the total number of connected components is $p$. After completion of routing, there will be $k$ connected components. Since each connection reduces the number of connected components by one, $(p - k)$ connections complete the routing.

**Lemma II.** If no modification is used, each of the connections requires at most $k$ $L$ operations.

**proof:** All the nets which have more than one connected component are in the queue. Among them, those which are scheduled after the last successful connection have feasible paths because all confirmed paths have been considered when finding the paths, and others may not be feasible. Since there are $k$ nets, all the scheduled paths are feasible at most after $k-1$ failures. Hence, the saved path of the first net in the queue is also feasible and one connection is possible at most after $k$ find-paths.

**Lemma III.** If strong modification is not used, the maximum number of weak modifications is bounded by $O(k\ p\ n)$.

**proof:** First we show that at least one element of the history matrix is increased within $2\ n$ modifications. For any of the weak modifications( unitpush, jumppush, and pointpush ) in the directions of left or right, on each row of the routing area, the same type of modification in the same direction can be successful at most $(n - 1)$ times. Since we start the modification from the closest two pins of different subnets, $2\ (n - 1)$ modifications are possible in the worst case. Similarly for the up or down modifications, $2\ (m - 1)$ modifications are possible.

It is clear that the maximum sum of all the element of the history matrix is $(T + 1)\ k\ (k - 1)$, where T is the number of weak modifications to be tried and is 12 in the current implementation of Mighty. Note that the diagonal elements of history are not used at all, because a net can not block itself. Each element of the matrix grows from 1 to $(T + 1)$. Since clear-history can be called at most $(p - k)$ times by Lemma I and each time $2\ (k - 1)$ elements are reset to 1, the maximum number of weak modifications is given by $2n\ (2(k-1)T\ (p-k) + k\ (k-1)T)$. Since $k < p$, the number of weak modifications is $O(k\ p\ n)$.

**Lemma IV.** The number of operations necessary for one weak modification is bounded by $O(k\ L)$.

**proof:** In the worst case, all $k$ nets can be pushed during one weak modification. If a moved net has more than one connected components, we need to do find-path and reschedule the net.

**Theorem II.**

(a) If the algorithm completes the routing without calling any modification routines, the computing complexity is bounded by $O(k\ p\ L)$.

(b) If the routing is completed by using weak modification but no strong modification, the worst case complexity of the algorithm is $O(k^2\ p\ n\ L)$.

(c) If strong modification is used, then the algorithm terminates after at most $O(k^3\ p\ n\ L)$ operations.

**proof:**

(a): The proof follows from Lemma I and II. Since we need to make $(p - k)$ connections and each connection takes at most $k\ L$ operations, $(p - k)\ k\ L$ operations are enough to complete the routing.

(b): From Lemma III, the maximum number of weak modifications is $O(k\ p\ n)$. From Lemma IV, one modification takes $O(k\ L)$. Hence the total number of operations

required in weak modification in worst case is $O(k^2\ p\ n\ L)$. To complete the routing, the complexity is $O(k^2\ p\ n\ L) + O(k\ p\ L) = O(k^2\ p\ n\ L)$.
(c): Let the limit of rip-up cost be $C$, and let the incremental rip-up cost be $D$. Then the maximum number of calls of the strong modification is $k\ \lceil C / D \rceil$. Since $C$ and $D$ are constants, it follows from (a) and (b) that the number of operations either to complete the routing or to report failure is $O(k^3\ p\ n\ L)$.

The bounds given above are worst case ones. Empirical results show that the computing time is roughly $O(mn)$ to $O((mn)^{1.5})$.

## 2.5. PARTITIONING OF LARGE PROBLEMS

Several other features are added to the routing algorithm based on the incremental routing modifications described in Section 2.3. The most important one is the addition of pseudo-pins to give global information to the maze routing phase in *find_path*. Other features are explained in the next section.

### 2.5.1. Pseudo-Pins

When we applied the algorithm to long channels of irregular shape, we found that the algorithm may spend large amount of time and find solutions that are not satisfactory. For this reason, we developed an additional technique that has made the application of the algorithm to large routing regions with irregular shape as well as to long channels with many nets successful. The idea is to guide the selection of the paths by placing "pseudo-pins" inside the routing region for a number of critical nets.

This scheme adds some global information about the routing region to the maze routing phase.

For long channels such as the Deutsch's difficult example, we place pseudo-pins at the column of maximum density. For wildly irregular channels, pseudo-pins are added across the columns where the variation in width occurs. To guide the search even further, pseudo-pins can be added at regular intervals away from the column of maximum density or where the irregularity takes place. The placement of the pseudo-pins is accomplished by examining the vertical constraint graph (see [110] or [73] for the definitions) of the channel. Pseudo-pins are not real pins in that the net being routed is not forced to go through them, but if it does not go through them the path is penalized.

Note that the selection of where to insert pseudo-pins as well as their placement is totally automatic in Mighty. No user intervention is required.

To describe the algorithm used for pseudo-pin generation, we need the following definitions.

**pseudo-pin:** A pin temporarily added to give some global information to the path finder and to help modification routines.

**compensated lateral distance** The distance in the horizontal direction in which pin density is considered. For example, the compensated distance between columns $c1$ and $c2$ is given by $|c1 - c2| + a\ (b + 2d)$, where $a$ is a constant and $b$ is the number of columns having one pin and $d$ is the number of columns having no pin, between $c1$ and $c2$. This reflects the freedom of routing due to pin density.

**vertical constraint:** A constraint that exists when two nets each have a pin in the same column. The net connected to the top pin must have its horizontal segment above that of the net connected to the bottom pin on the column.

**horizontal constraint:** A constraint that exists when two nets cross the same column. If pseudo-pins have to be added to guide the routing of nets that have an horizontal constraints, pseudo-pins should be added on different rows even though these pins are added on different columns.

Violating vertical or horizontal constraints does not imply that complete routing can not be achieved. However, satisfying these constraints makes the later routing easier.

As shown below, pseudo-pins are generated using a linear assignment algorithm so that the number of VCV (vertical constraint violations) or HCV (horizontal constraint violations) and wire-length are minimized. Setting the cost parameters in linear assignment is very important for satisfactory results. The algorithm we use is described in detail in Section 2.5.1.2. Since the path finder does not use any information about vertical or horizontal constraints, we can give some hints to the path finder by generating pseudo-pins such that each vertical or horizontal constraint violation is penalized. Using the top-most and bottom-most tracks is also penalized. To find the level_from_top and level_from_bottom which are the highest and lowest rows that a net can be assigned without VCV [73], we need to break all the cycles, if any, in the vertical constraint graph. We remove cycles by removing edges which are created by a column farthest from the column on which pseudo-pins are being generated. In other words, we generate the pseudo-pins such that vertical constraints are satisfied near the column and let the main detailed router( path finder, path confirmer, weak and strong modifiers ) choose the exact path of connections to avoid blockages where vertical constraint violations are observed.

### 2.5.1.1. Pseudo-Pin Generation

A linear assignment algorithm generates a matching of crossing nets and available tracks while minimizing the total cost of the matching. (Linear assignment has independently been used for global routing by M. Marek-Sadowska and U. Lauther [51, 64]. )

Let $N = \{n_1, n_2, \dots, n_p\}$ be the set of crossing nets, and let $T = \{t_1, t_2, \dots, t_q\}$ be the set of available tracks on the column. Note that $q$ should be greater than or equal to $p$ to assign pseudo-pins for all the crossing nets. A bipartite graph $B(X, Y, U)$ is constructed by associating nodes in $X$ to the nets in $N$, nodes in $Y$ to the tracks in $T$ and adding an arc $(n_r, t_c) \in U$ with cost $c_{rc}$ for each $1 \leqslant r \leqslant p$ and $1 \leqslant c \leqslant q$. ($c_{rc}$ is the cost of assigning a pseudo-pin of net $n_r$ on track $t_c$.) Then a source node $s$ and a sink node $t$ are added. Finally to make sure that each net is assigned to only one track, arcs $(s, n_r)$ and $(t_c, t)$ are added with capacity 1, for all $1 \leqslant r \leqslant p$ and $1 \leqslant c \leqslant q$. Now we can obtain an assignment by solving a maximum flow and minimum-cost problem on the graph.

The following algorithm shows how pseudo-pins are generated on a selected column.

**pseudo-pin_generation()**

```
select a column;
find available tracks;
find the nets crossing the column;
construct vertical constraint graph on proper interval;
while( there is a cycle )
    break the edges that are caused by constraints on farthest columns;
fill cost matrix;
solve linear assignment problem;
add pseudo-pins;
```

### 2.5.1.2. Setting Parameters in the Cost Matrix

In the algorithm, each row $r$ of the matrix corresponds to a crossing net *net* $(r)$ and each column $c$ corresponds to an available track $tr(c)$. Each element $cost[r, c]$ of the cost matrix is obtained by adding the costs from vertical constraints, horizontal constraints, and other pin locations of the *net* $(r)$.

**fill_cost_matrix()**

```
    /* VCV_cost: parameter penalizing vertical constraint violation.
       HCV_cost: parameter penalizing horizontal constraint violation.
       Hslope (Vslope): parameter penalizing distance to other pins on
       horizontal (vertical) preferred layer. */

    /* Generate pseudo-pins on a column col of the channel */
    for( each row r )
    {
        /* penalize vertical constraint violation */
        for( each column c )
            if( assigning net (r) on track tr(c) cause VCV )
                cost[r, c] = cost[r, c] + VCV_cost;

        /* penalize horizontal constraint violation */
        for( each column c )
            /* HCV may occur due to other nets or obstacles */
            if( assigning r to c causes HCV )
                cost[r, c] = cost[r, c] + HCV_cost;

        /* look around other pins */
        for( each pin p of net (r) )
        {
            /* by using large value of Hslope, all the pseudo
               pins of a net can be generated on a track. */
            /* Let (x, y) be the coordinates of the pin p */
            if( p is on the horizontal preferred layer )
                /* α is a constant
                   dist returns the compensated distance */
                slope = Hslope/(α + dist(x, col));
            else
                slope = Vslope/(α + dist(x, col));
            for( each column c )
                cost[r, c] = cost[r, c] + slope * | tr(c) - y |;
        }
    }
```

## 2.6. OTHER VARIATIONS TO THE BASIC ALGORITHM

Several other features have been added to make the router practical. These beneficial features are equivalent pins, floating pins, prerouted nets, critical nets, and handling of pins on either layer.

### 2.6.1. Pre and Post Processing

Pre-processing sets up the data structure and marks obstacles in the routing region so that *find_path* can avoid them when finding a shortest path.

Post-processing is a "clean-up and beautification" step. During post-processing, all the nets are rerouted from the net with longest net length to the one with shortest net length. One net at a time is completely removed and routed again with different cost parameters. Since all other nets are connected, we can penalize less routing in the vertical (horizontal) direction on horizontal (vertical) layer. Metal maximization can be achieved by assigning less cost to one of the two layers (for example, when the two interconnection layers are poly and metal). Also the total wire-length and the number of vias can be traded off by setting the cost parameters accordingly.

**pre-processing()**

```
move pins on the boundary inside the channel by one grid space;
if( channel is long OR irregular )
      generate pseudo-pins;
if( there exist floating pins )
      assign positions for them;

/* Mark obstacles as blocked */
if( there exist blocked area )
      mark the blocked grids in the routing area;
if( there exist equivalent pins )
      merge the pins to form a component;
```

**post-processing()**

```
/* via and wire-length minimization */
change cost parameters in find_path;
sort the nets by total wire length in decreasing order;
for( each net i in the order given above )
{
      remove all segments of net i
      while( net i has more than one component )
      {
            path[i] = find_path( i );
            confirm_path( i );
      }
}
```

### 2.6.2. Equivalent Pins

Some pins are electrically equivalent since they are interconnected outside the routing region already. The algorithm deals with these cases very naturally, since these pins may be given the same "component name" when the maze router is invoked and the interconnection will reflect the additional optimization that results from taking advantage of this feature.

### 2.6.3. Floating Pins

Floating pins at the boundaries of routing regions are often present. The algorithm decides the positions of floating pins during preprocessing, using the same procedures as in the generation of pseudo-pins. As shown in the pre-processing algorithm, this is done after all the pseudo-pins are generated.

### 2.6.4. Obstacles and Prerouted Nets

If there are obstacles on any of the two layers, then we mark this area and the path-finder is prevented from using it.

If a net is completely prerouted then we can mark the interconnection as an obstacle. (For example, power and ground nets can be prerouted [14, 68, 97, 108]. ) If only part of a net is prerouted, then we can start routing from the existing status. For example, if we attach temporary pins on all the grid points used by the pre-routed net, the pre-routing is used as an initial condition and the path-finder will optimize other necessary connections.

If Mighty is used for automatic cell generation or routing over the cells, we may find an obstacle covering a large portion of one layer. In this case, if there are several pins of different nets on the other layer at the same location as the obstacle, modifications of existing connections are limited. To overcome this problem, a routine is activated if strong modification fails because of an obstacle, which tries to guide the blocked pins to the region where both of the layers are available for routing.

### 2.6.5. Critical Nets

Some nets may be electrically critical. These nets can be routed first using minimum number of vias and wire-length. If necessary, we can use a different set of costs. For example, we can route power nets on the metal layer simply by assigning a very high cost to the poly-silicon layer. We can force the modifiers not to touch these important nets by assigning high cost to the modification of their interconnections.

### 2.6.6. Routing on the Boundaries

Another important factor that decides the routing area of real examples is the ability to handle pins on an arbitrary layer. Most conventional routers [35, 73] assume that pins on vertical boundary segments are on one layer and those on horizontal boundary segments are on the other layer. However, real examples frequently do not satisfy this assumption, and conventional routers need extra tracks and columns to change layer and make more vias than necessary.

## 2.7. EXPERIMENTAL RESULTS

All the results described in this section have been obtained by using the same set of cost values, on a DEC VAX 11/785 running 4.3BSD UNIX.

In the tables, "Mighty" is the name of our router. Table 2.2 shows a comparison with other well-known routers on the Burstein's difficult switchbox. Mighty uses one less column than any other routers. Mighty takes 2.7 seconds to complete the connections in this example, and post processing takes 1.3 seconds. Note that WEAVER took 1,390.0 seconds to route the same example on a VAX 11/780. WEAVER is written in OPS5 and C. Its run-time is long in part because the use of OPS5. A speed-up can certainly be obtained by rewriting the router entirely in C. However, in this case, some of the most interesting features of expert systems would be lost. We believe that the complexity of the underlying approach is also responsible for its long run-time. Table 2.3 shows the results of other switchbox examples.

The detailed router described in this chapter is quite symmetric in the vertical or horizontal directions, as opposed to other algorithms reported in the literature that are sensitive to this factor [62]. We tried to route some of the switchboxes after rotating the channel by 90 degrees, and in all cases obtained basically the same results. As an example, two routers are compared on Burstein's switchbox example in Table 2.4.

| Router Name | #rows | #columns | #vias | total length |
|---|---|---|---|---|
| Hamachi | 15 | 23 | 67 | 564 |
| Luk | 16 | 23 | 58 | 577 |
| Mod Detour | 15 | 23 | 63 | 567 |
| M-Sadowska | 15 | 23 | 58 | 560 |
| WEAVER | 15 | 23 | 41 | 531 |
| Mighty | 15 | 22 | 39 | 541 |

Table 2.2 : Routing of Burstein's difficult switchbox

| Example Name | Router Name | #rows | #cols | #vias | total length |
|---|---|---|---|---|---|
| simple | Lee | 7 | 7 | 11 | 60 |
| | WEAVER | 7 | 7 | 4 | 60 |
| | Mighty | 7 | 7 | 5 | 60 |
| term inten | Luk | 16 | 23 | 68 | 632 |
| | WEAVER | 16 | 23 | 49 | 615 |
| | Mighty | 16 | 23 | 50 | 629 |
| dense | Luk | 18 | 16 | 36 | 527 |
| | Mighty | 18 | 16 | 32 | 530 |
| mod dense | WEAVER | 17 | 16 | 29 | 510 |
| | Mighty | 17 | 16 | 29 | 510 |

Table 2.3 : Routing of switchbox examples

| scan direction | #tracks used | |
|---|---|---|
| | Luk | Mighty |
| orientation1 | 23 x 18 | 22 x 15 |
| orientation2 | 23 x 20 | 22 x 15 |
| orientation3 | 23 x 16 | 22 x 15 |
| orientation4 | 24 x 16 | 22 x 15 |

Table 2.4 : Dependence on the orientation of the routing area.

As pointed out above, our router has been developed to deal with switchboxes and highly irregular shaped channels as well as rectangular channels. Table 2.5 shows the comparisons on the Deutsch's difficult channel example [17]. Table 2.6 shows the results when we applied Mighty to a number of difficult channels as compared with the results of YACR [73] and Chameleon [6]. In all the examples tried, our router shows equal or better results as we expect since Mighty has more degrees of freedom. However, note that in the past the use of switchbox routers in channels has not given as compact results as the ones which can be obtained with channel routers. Hence, the above results show how powerful the techniques used in Mighty really are.

Three examples corresponding to cases when the modifiers were not used, when weak modification was used and when strong modification was employed are shown in Fig. 2.5, 2.6 and 2.8. Fig. 2.5 is the result of our router when applied to a difficult channel presented in a hierarchical routing paper [10]. Note that we have not used an empty column in the middle of the channel. Mighty could complete the routing without using any modifications by using 4 tracks. To route this example with one more empty column, Burstein's router used 5 tracks and YACR used 6 tracks. Fig. 2.6 shows the result of Burstein's switchbox. To complete the routing, two weak modifications and no strong modifications were used. Fig. 2.8 shows the result of

Deutsch's channel example routed by Mighty. During routing Deutsch's channel. twenty-seven weak modifications and sixteen strong modifications were used. The total CPU time was 178 seconds including 46 seconds of post processing time.

Table 2.7 shows the statistics of the channel examples. The number of weak and strong modifications required to complete the routing are shown with the number of vias and total wire-length of the routing results. For some examples such as ex2 and r3, weak modification was effective. For others such as ex1 and ex4, weak modification hardly resolved situation, but strong modification was necessary to make complete connections. Hence both of the weak and strong modifications are important for satisfactory results. These results are obtained using the same set of parameters. By adjusting the parameters for each example, better result can be obtained.

Since the electrical properties are different on different layers, one layer may be preferred to the other. For the examples, we assumed that one layer is polysilicon and the other layer is metal. The usage of metal layer could be maximized by assigning larger cost parameters on the polysilicon layer as shown in Table 2.8.

The router has also been used to generate CMOS cells, interfaced with TOPOGEN which is a synthesis tool that takes a logic description at the gate level and converts it into a symbolic layout of a static CMOS circuit on virtual grids [84]. TOPOGEN generates the sequence of p and n channel transistors on two rows and Mighty generates all the necessary interconnections between the transistors to obtain a desired functional circuit block. Fig. 2.9 shows a sample result routed by Mighty, during the generation of a 16 input CMOS gate. YACR used 4 more tracks to route the same example.

| Router Name | #rows | #vias | total length |
|---|---|---|---|
| Yoshimura.Kuh | 20 | 308 | 5075 |
| Hamachi | 20 | 412 | 5302 |
| Burstein | 19 | 354 | 5023 |
| YACR | 19 | 287 | 5020 |
| Mighty | 19 | 301 | 4812 |

Table 2.5 : Routing of Deutsch's difficult channel

| | channel | | | #tracks used | | |
|---|---|---|---|---|---|---|
| Example Name | #nets | #cols | density | YACR | Chameleon | Mighty |
| 3a | 30 | 45 | 15 | 15 | 15 | 15 |
| 3b | 47 | 62 | 17 | 18 | 18 | 17 |
| 3c | 54 | 103 | 18 | 19 | 19 | 18 |
| chris1 | 158 | 432 | 49 | 50 | 49 | 49 |
| cycle.t | 65 | 134 | 16 | 19 | 17 | 17 |
| ex1 | 235 | 417 | 16 | 17 | 16 | 15 |
| ex2 | 282 | 421 | 15 | 16 | 16 | 15 |
| ex3 | 291 | 421 | 11 | 12 | 12 | 11 |
| ex4 | 270 | 421 | 19 | 22 | 22 | 20 |
| r1 | 77 | 139 | 20 | 22 | 22 | 22 |
| r2 | 77 | 117 | 20 | 21 | 20 | 20 |
| r3 | 78 | 123 | 16 | 18 | 18 | 17 |
| r4 | 74 | 150 | 15 | 17 | 17 | 17 |

Table 2.6 : Routing of channel examples

| Example Name | # weak mod. | # strong mod. | # vias | total length |
|---|---|---|---|---|
| 3a | 10 | 7 | 72 | 991 |
| 3b | 14 | 6 | 107 | 1617 |
| 3c | 14 | 7 | 163 | 2392 |
| chris1 | 52 | 33 | 471 | 19858 |
| cycle.t | 36 | 27 | 231 | 3345 |
| ex1 | 23 | 19 | 326 | 6794 |
| ex2 | 7 | 0 | 341 | 7346 |
| ex3 | 12 | 6 | 308 | 5653 |
| ex4 | 20 | 19 | 303 | 7768 |
| r1 | 32 | 19 | 220 | 4120 |
| r2 | 17 | 12 | 165 | 3232 |
| r3 | 10 | 11 | 189 | 2950 |
| r4 | 20 | 9 | 248 | 3458 |

Table 2.7 : Routing results of channel examples

| Example name | before post processing | | | after post processing | | |
|---|---|---|---|---|---|---|
| | #via | metal length | poly length | #via | metal length | poly length |
| 3a | 79 | 561 | 430 | 72 | 565 | 426 |
| 3b | 118 | 843 | 810 | 107 | 851 | 766 |
| 3c | 170 | 1312 | 1115 | 163 | 1344 | 1048 |
| chris1 | 499 | 13364 | 7010 | 471 | 13801 | 6057 |
| cycle.t | 257 | 1686 | 1687 | 231 | 1746 | 1599 |
| ex1 | 377 | 3284 | 3616 | 326 | 3448 | 3346 |
| ex2 | 390 | 3148 | 4266 | 341 | 3407 | 3939 |
| ex3 | 351 | 2902 | 2770 | 308 | 2978 | 2675 |
| ex4 | 343 | 2828 | 5028 | 303 | 3410 | 4358 |
| r1 | 227 | 2271 | 1935 | 220 | 2370 | 1750 |
| r2 | 172 | 1668 | 1570 | 165 | 1712 | 1520 |
| r3 | 206 | 1405 | 1567 | 189 | 1434 | 1516 |
| r4 | 279 | 1925 | 1606 | 248 | 1951 | 1507 |
| ddr | 319 | 2579 | 2333 | 301 | 2628 | 2184 |

Table 2.8 : Beautification by post-processing

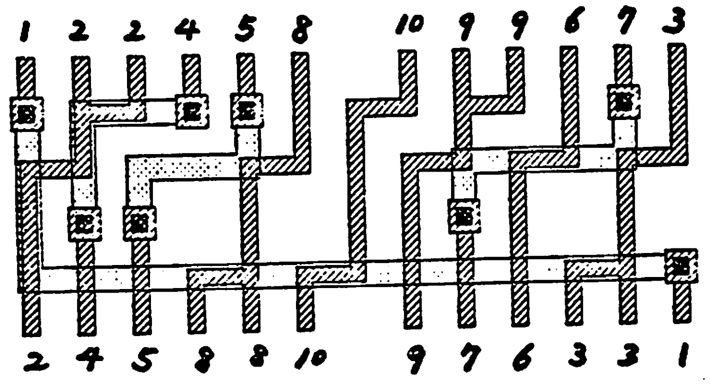


**Figure 2.5.** *Channel example routed by Mighty.*

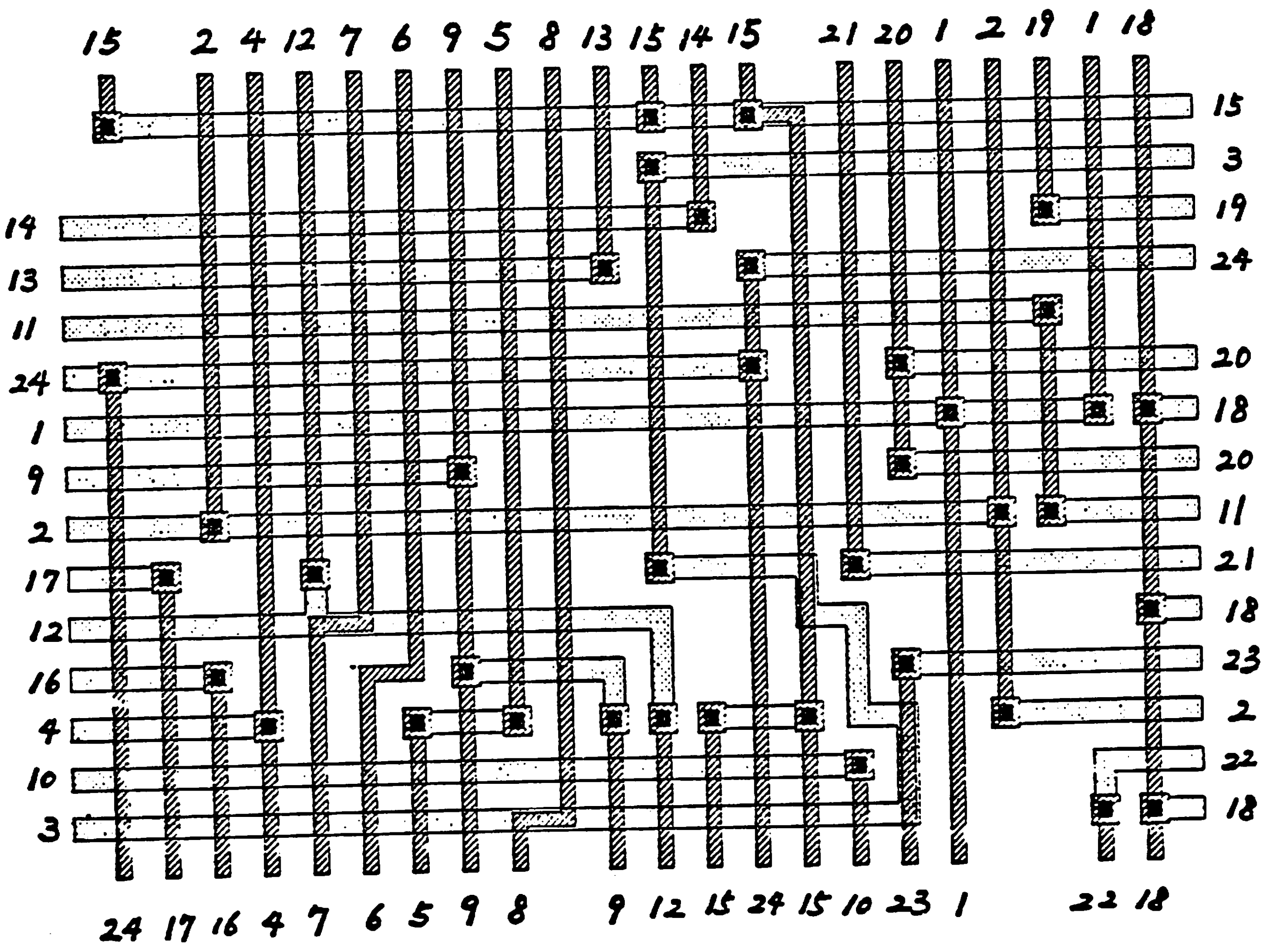
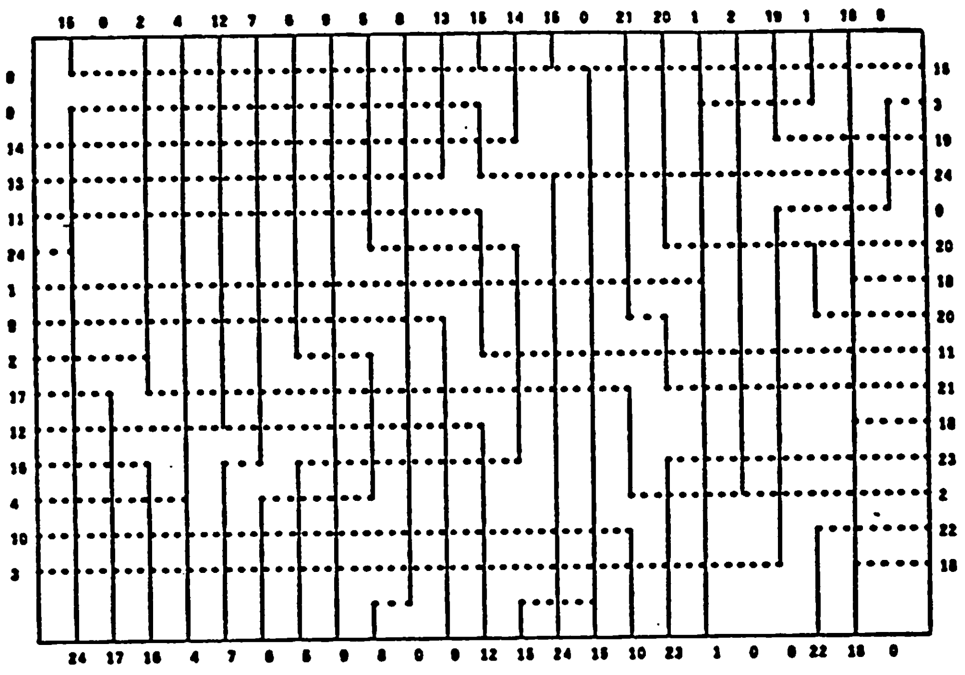
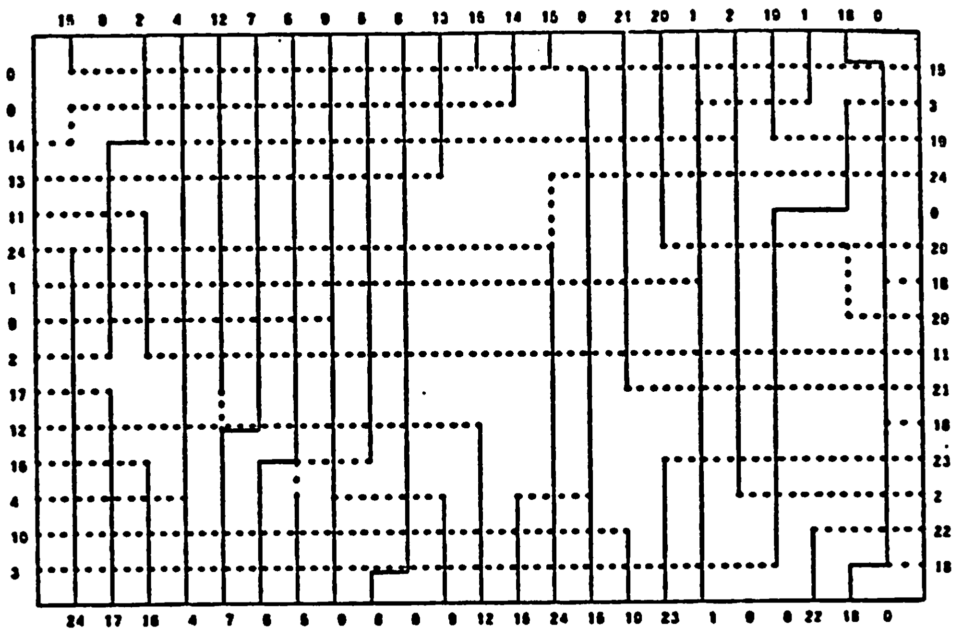


**Figure 2.6.** *Burstein's difficult switchbox routed by Mighty.*



(a) Luk's solution to Burstein's difficult switch-box.



(b) WEAVER's solution to Burstein's difficult switch-box.

**Figure 2.7.** *Burstein's difficult switchbox routed by two other routers. (a) Luk's solution, (b) WEAVER's solution.*

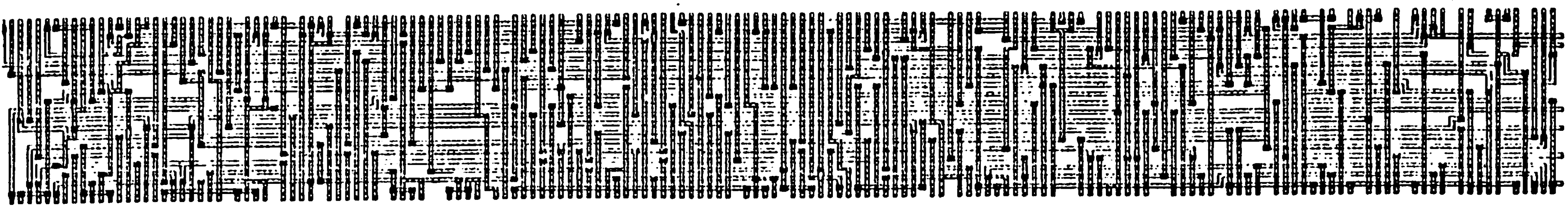


Figure 2.8. *Deutsch's difficult channel routed by Mighty.*

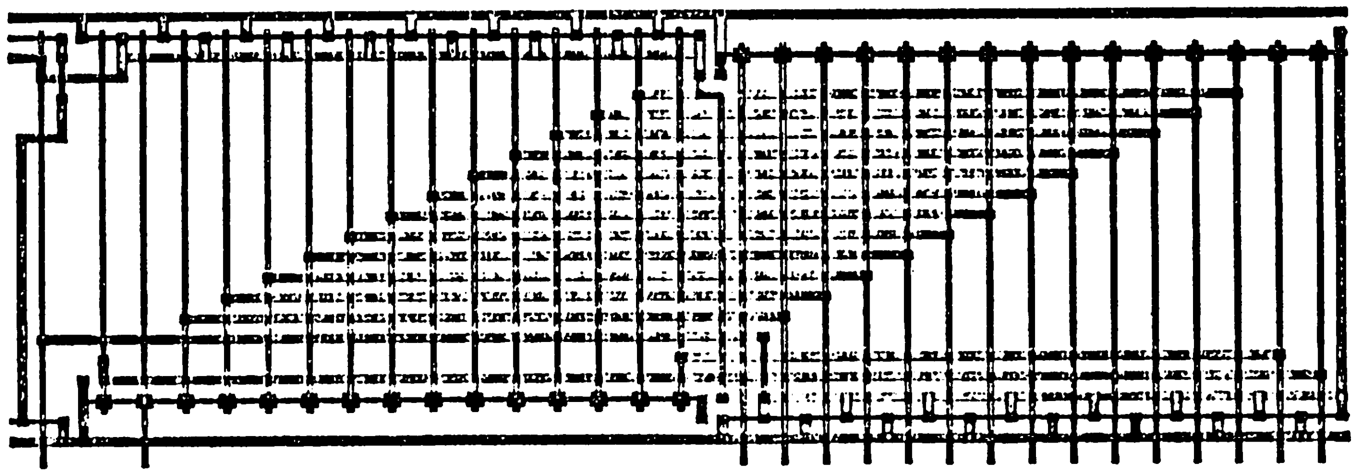


Figure 2.9. *CMOS gate(NAND (NOR X2 X3 ... X16) (NAND X1 X2 ... X15)) routed by Mighty.*

## 2.8. SENSITIVITY OF THE ALGORITHM

The described algorithm has many parameters. For all the examples described in this chapter, the same parameters are used. Since Mighty performed as well as or better than existing algorithms on both channel and switchbox examples, the algorithm seems to be robust with respect to parameters.

The cost parameters in finding a shortest path are the most important ones. Hence, the sensitivity of the algorithm to cost parameters in finding a shortest path is examined. We fixed the cost per wire-length in the preferred direction to 2 and the cost of a via to 30; we varied the cost per non-preferred direction (Wnp) from 20 to 70. When applied to Burstein's switchbox example, Mighty gives the same results for Wnp = 40, 50, 60 (see Fig. 2.10). When applied to Deutsch's channel, Mighty completes the routing with 19 tracks for Wnp = 40, 50, 60, 70; the total wire-length is minimum at Wnp = 50.

| Example | Wnp | 30 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|---|
| Burstein switch-box | via<br>length | fail | 39<br>541 | 39<br>541 | 39<br>541 | fail | fail |
| Deutsch channel | via<br>length | fail | 300<br>4825 | 301<br>4812 | 299<br>4830 | 314<br>4920 | fail |

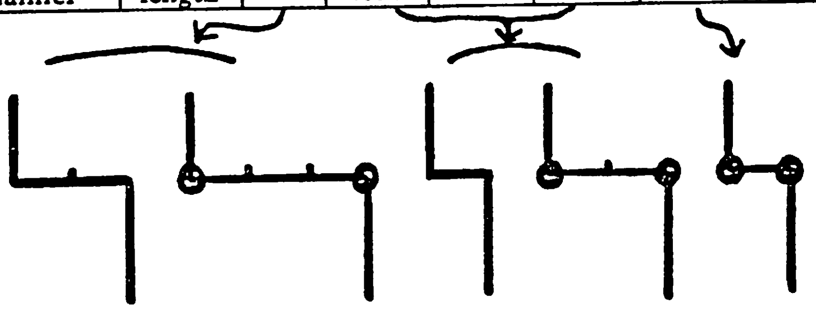


Figure 2.10. *Sensitivity of the algorithm on cost parameters.*

The basic ideas we followed to select the parameters are also shown in Fig. 2.10. If changing layers by introducing two vias is not wanted even if the routing is in non-preferred direction, we may use small values of Wnp (for example, Wnp = 30). On the contrary, if we like to use one layer for horizontal segments only and the other layer for vertical segments only, then we may use large values of Wnp (for example, Wnp = 70). Too much restriction on layer usage increases the number of vias unnecessarily, and too much flexibility can create blockages. Hence, we believe that Wnp = 40 to 60 is a good range to use.

Max_cost in *find_path* and Limit_cost in *strong_modification* are not critical as long as they are large enough. Max_cost should be large enough so that all the available paths can be found. Large values of Limit_cost allow the algorithm to search more rip-up paths; however CPU time increases as shown in Section 2.4. Currently, Max_cost = Limit_cost = 500 is used, and further increase of these values could not reduce the routing area for the examples tried. Note that if routing area is large, pseudo-pins are added and the length of the shortest paths connecting components of nets can be bounded.

## 2.9. SUMMARY

A new general two-dimensional routing algorithm has been presented. For this algorithm, rather than trying to avoid blocking the paths of unrouted nets, a new approach is developed in which the minimum-cost path is found and connected incrementally, and then the existing connections are modified or rerouted whenever the incremental interconnections are not satisfactory. This router is very flexible. For example, it can route on a preferred layer as much as possible or trade off the number of vias and the wire-length. The mechanism used to guide this routing is parameter setting.

This approach can be extended to multi-layer routing for macro-cells and printed circuit board routing. A multi-layer router, Angelar, has been developed using the same algorithm and is described in Section 4.1.1.1.

The program implementing the algorithms described in this chapter consists of about 11,000 lines of C. This router has been developed as a part of an integrated automatic synthesis and layout system for VLSI design [7].

3.

# Chapter 3

# Two-Dimensional Layout Compaction Using 'Zone-Refining'

Layout compaction plays an important role in the automatic or semi-automatic generation of integrated circuits. Layout compaction consists of rearranging the geometries of the design with small (if any) topological modifications to generate a layout that satisfies all the design rules and is as compact as possible.

The availability of good layout compaction programs speeds up the design of custom integrated circuits. The designer can work at the symbolic level, where it is easier to capture the design intent and to find an optimum topology for a particular circuit block. The conversion from the symbolic representation to an actual layout that obeys all technological design rules is a very tedious and time consuming process that is best off-loaded to automated procedures.

In standard cell layout systems good layout compactors permit a quick adjustment of the cells to small changes in the technological specifications.

Even for the development of fully procedural 'silicon compilers', the availability of good geometrical spacing programs is a big win. The routine that generates a cell can then produce a description at the symbolic level, rather than the detailed final layout, and the proper geometrical spacing of all components is handled by a separate tool. The detailed knowledge about the design rules and the algorithms needed to enforce them are thus concentrated in one program and do not have to be dealt with by the individual designer who wants to create a procedural description of a cell or a chip. This approach speeds up the process of creating procedural cell generators, since the procedural specification of all the detailed geometric design rules would be the most time-consuming part of writing such a generator.

The most general form of compaction involves moving the geometries of the design in the $x$ and $y$ coordinates simultaneously. The problem of generating a correct layout of minimum area with this set of moves has been shown to belong to the class of NP-hard problems [79]. For this reason, most of the compactors proposed in the past, e.g. SLIP [20], STICKS [104], CABBAGE [37], SPARCS [9], and MACS [61] decompose this 2-dimensional problem into an alternating sequence of independent one-dimensional compaction steps that ignore the information in the direction orthogonal to the axis of compaction. These one-dimensional compaction steps can be solved efficiently with longest path algorithms [37, 58]. (A slightly different approach is virtual grid compaction [13, 31, 69].)

However, such algorithms cannot compete with the quality of layouts produced by human designers. Situations where two circuit blocks cannot be shifted past one another because they just catch with their corners cannot be handled by one-dimensional algorithms, whereas a human designer would have no problems to make the necessary adjustments in the other coordinate (Fig. 3.1).

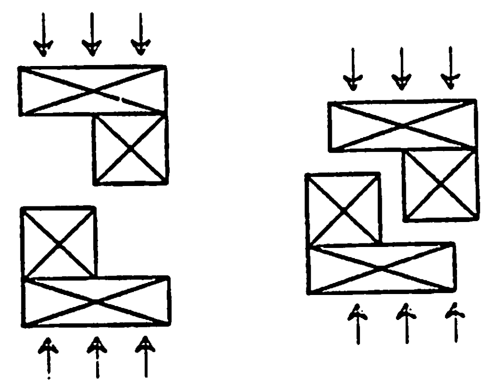


**Figure 3.1.** *Interference that needs to be resolved with a movement perpendicular to the axis of compaction.*

In earlier compactors only lower bound constraints on the position of the geometries were taken into consideration. Several layout problems such as those encountered in the design of analog components require for electrical reasons that certain geometries be not separated by more than a prescribed quantity. This requirement amounts to considering upper bound constraints when compacting the layout. Recently, compaction algorithms that deal with upper bound constraints have been developed [4, 58].

In this chapter, a new algorithm for two-dimensional compaction with upper bounds as well as lower bounds on the positions of the components is presented. After a traditional one-dimensional precompaction step, the size of the layout is further reduced with a technique that bears a strong similarity to the technique of 'zone-refining' used in the purification crystal ingots. Individual circuit components or small clusters of components are peeled off row by row from the precompacted layout, moved across an open zone, and reassembled at the other end of this zone in a denser configuration. In this process both coordinates of the moved components are altered and jogs are introduced in the connecting wires between them to produce the needed flexibility for placing components into optimal positions.

The change in the position of the geometries is not as free as in the general 2-dimensional compaction problem. but the restrictions placed by the peeling procedure have allowed the development of a polynomial algorithm that produces excellent results when compared to traditional compaction algorithms.

This chapter is organized as follows. In Section 3.1. previous work on two-dimensional compactions are summarized. In Section 3.2. the main idea of the algorithm is informally described. In Section 3.3. the technique applied to the problem of compacting unconnected boxes in a bin to demonstrate the operation of the algorithm is described. In Section 3.4. the algorithm for compacting layouts of integrated circuits is presented. In this section. we also describe the jog generation technique used in our prototype. and show how simulated annealing can be used with zone-refining. In Section 3.5. non-trivial design rules are discussed. In Section 3.6. a new hierarchical compaction method is presented. In Section 3.7. variations of the zone-refining technique are discussed. In Section 3.8. we discuss our experimental results. Finally. in Section 3.9. summary is presented.

## 3.1. PREVIOUS WORK ON TWO-DIMENSIONAL COMPACTION

To obtain competitive layouts from automatic compactors. it is necessary to introduce techniques that have the freedom to move components in both coordinate axes when the compacted circuit is built up. Several approaches have been proposed. W. Wolf et al. [105] described a method in which cell pitch is minimized by finding the critical path, shearing the component pairs on this critical path, and performing compaction in the preferred direction. In this method, the compaction itself is still one-dimensional.

True two-dimensional compaction has been attempted with exponential complexity algorithms. One approach [81] is to start with a totally collapsed layout except for topological constraints (base constraints) and then remove the distance violations one by one. For this, a branch and bound search technique is used, keeping only the best layout obtained. G. Kedem and H. Watanabe [45, 102] translated the compaction problem into a special form of a mixed-integer programming problem and used a graph-based optimization to solve the resulting problem. However, formulating a model which is NP-hard does not seem to be very helpful for solving the problem [13]. J. Weinstein suggested a simple heuristic in selecting between horizontal and vertical constraints; the preference is given to the constraint in the direction where the original distance was larger and this algorithm is strongly dependent on the initial topology of layouts [103]. Recently R. Mosteller et al. [67] used a Monte Carlo simulated annealing technique and conjectured that the average complexity of the algorithm is given in polynomial form. However, this algorithm still takes large amount of CPU time and cannot guarantee a globally optimum solution.

With two-dimensional compaction, the user may have difficulty in predicting the outcome due to the increased degree of freedom during compaction process. However, this can not be considered a real problem when the physical design procedure becomes completely automatic.

## 3.2. COMPACTION BY 'ZONE-REFINING'

The natural process of crystallization has proven to be an inspiration to the generation of densely packed layouts. The physical model of a crystal that is cooled slowly and thereby settles in its minimum energy configuration, has stimulated the technique of simulated annealing [48], used for global placement and routing of integrated circuits [83, 101]. This process moves blocks in any order and deals with the circuit as a whole. As a consequence, globally optimal solutions can be achieved but at the cost of very long run times.

We have developed a new technique that concentrates on more localized rearrangements of the circuit components, assuming that a good global ordering of the circuit has already been generated by some other tool. Our technique resembles the process of zone-refining used for the purification of crystal ingots [72].

Fig. 3.2 shows zone-refining process (a) of a crystal ingot and (b) of a layout. Fig. 3.2 (c) shows zoom-in view of an intermediate step of zone-refining. The zone-refining process starts with an already established crystal ingot. This ingot is slowly pulled through a heating spiral that locally heats the crystal to melting temperature. At the exit end of that heat zone the material recrystallizes. In this process the newly formed crystal has a lower concentration of impurities since the impurities are typically built into the lattice at a lower rate than the proper crystal atoms. Thus most impurity atoms are kept in the molten zone and are eventually swept out of the crystal. If higher purity is required, the process can be repeated.

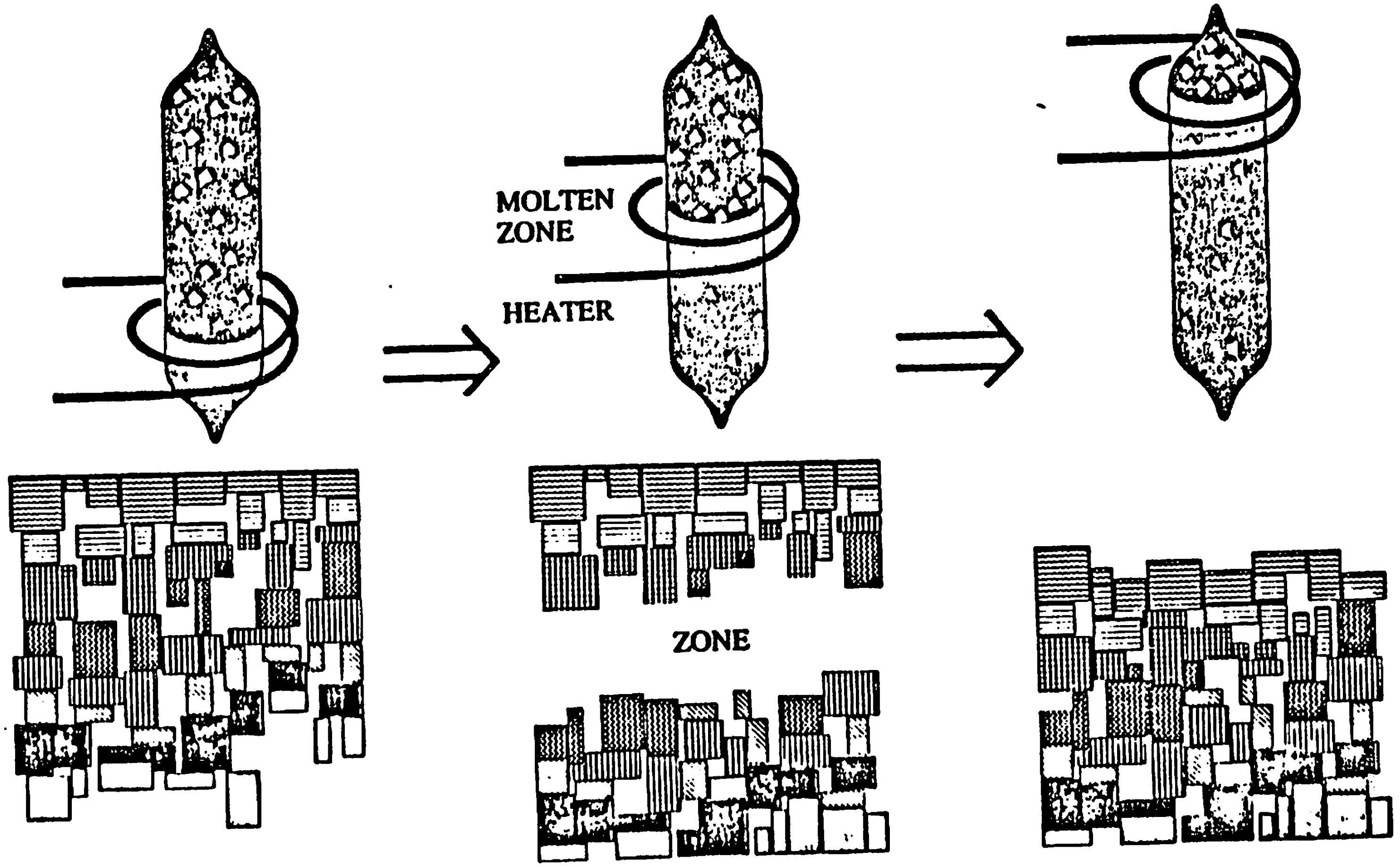


Figure 3.2. *Zone-refining: (a) of crystal ingots and (b) of layouts.*

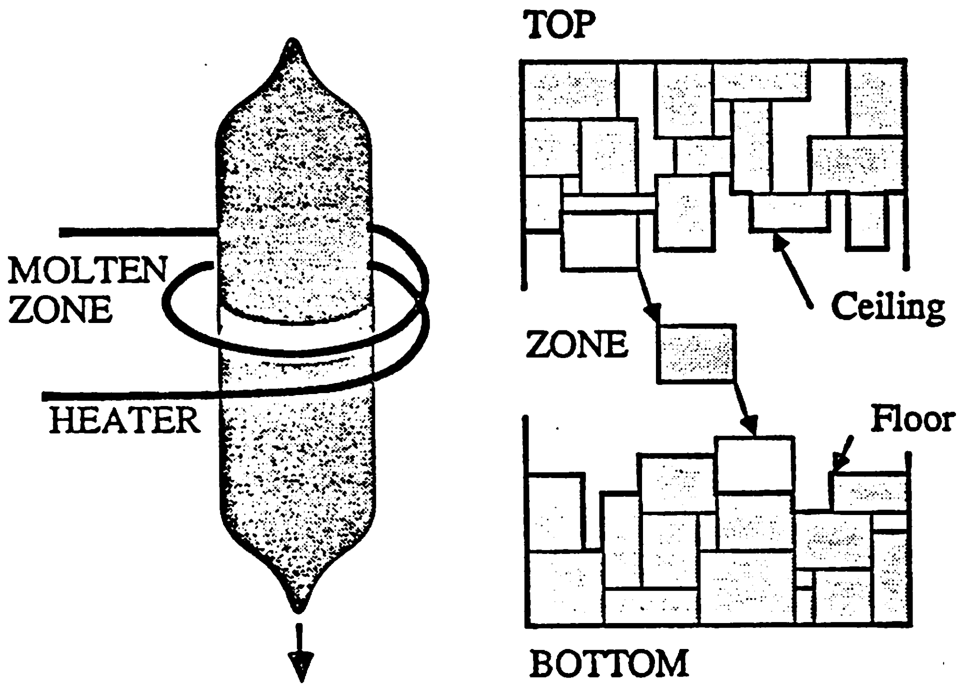


**Figure 3.2.** *Zone-refining: (c) intermediate step*

In close analogy with this process, we start our compaction procedure from a precompacted circuit that corresponds to the original crystal (Fig. 3.2). In our case the 'impurities' are the unnecessary voids between circuit components. Starting from one side, individual circuit components or small clusters of components are peeled off row by row from the precompacted layout and are reassembled after they have been moved across an open zone. In the reassembly step at the other end of the zone, both coordinates of the moved components can be altered and jogs can be introduced in the connecting wires between the circuit components. These additional degrees of freedom permit a higher packing density in the newly formed part of the layout.

The implementation of this concept is straight-forward for the simple case of rectangular blocks that need to be placed in a compact manner into a bin of given width as shown in the next section.

In integrated circuit layouts the interconnections add a considerable complication. They can severely constrain the freedom with which the individual components can be moved. Thus it is necessary to introduce additional jogs in the wires between them to permit less restricted lateral movements of each component. We have considered moving the blocks without their attached wires and subsequently rerouting the connections, a technique similar to the one proposed by Maley [63], but we have not pursued this possibility so far. First, this approach would considerably increase the possible moves that need to be considered and would thus increase the complexity of the algorithm and its run time. Secondly, interconnections play a crucial role in the performance of the circuit, and the given topology of the circuit has often been chosen based on considerations at the micro-architecture level. Thus we did not want the compaction tool to make profound changes to the topology of the circuit; this is the task of a different kind of tool that can take properly into account concerns beyond the adherence to the geometrical design rules.

Thus for the zone-refining process we assume that we start from a good topology given, for instance, in the form of a symbolic sticks diagram. This basic ordering is maintained in the compaction process, distinguishing our approach from the more general problem of block placement and routing.

The advantage of the zone-refining approach is that the number of components that must be considered at any one time is thus dramatically reduced, and the complexity of the algorithm becomes manageable. Just as in the physical zone-refining process, the compaction process can be repeated if the results are not satisfactory after the first pass.

## 3.3. BOX PACKING

To demonstrate the operation of the zone-refining algorithm and the data structures employed, we use a much simpler problem than actual circuit compaction. We assume that an ordered collection of rectangular boxes are to be packed into a bin of given width, while minimizing the original height of the packed bin and keeping the reordering of the topological arrangement of the boxes limited.

The starting configuration, analogous to the original crystal, is placed at the top part of Fig. 3.3(a). Boxes are removed from the lower end of this *top* configuration and reassembled in a new *bottom* configuration growing upwards. The *floor* and *ceiling* are the borders of the active "molten" zone towards the bottom and top constellations respectively. These structures are referred to when blocks are selected and moved across the free zone between top and bottom. They can easily be updated using the information in the adjacency graph. The selected box being transferred across the zone can move freely between floor and ceiling (see Fig. 3.3(c)).

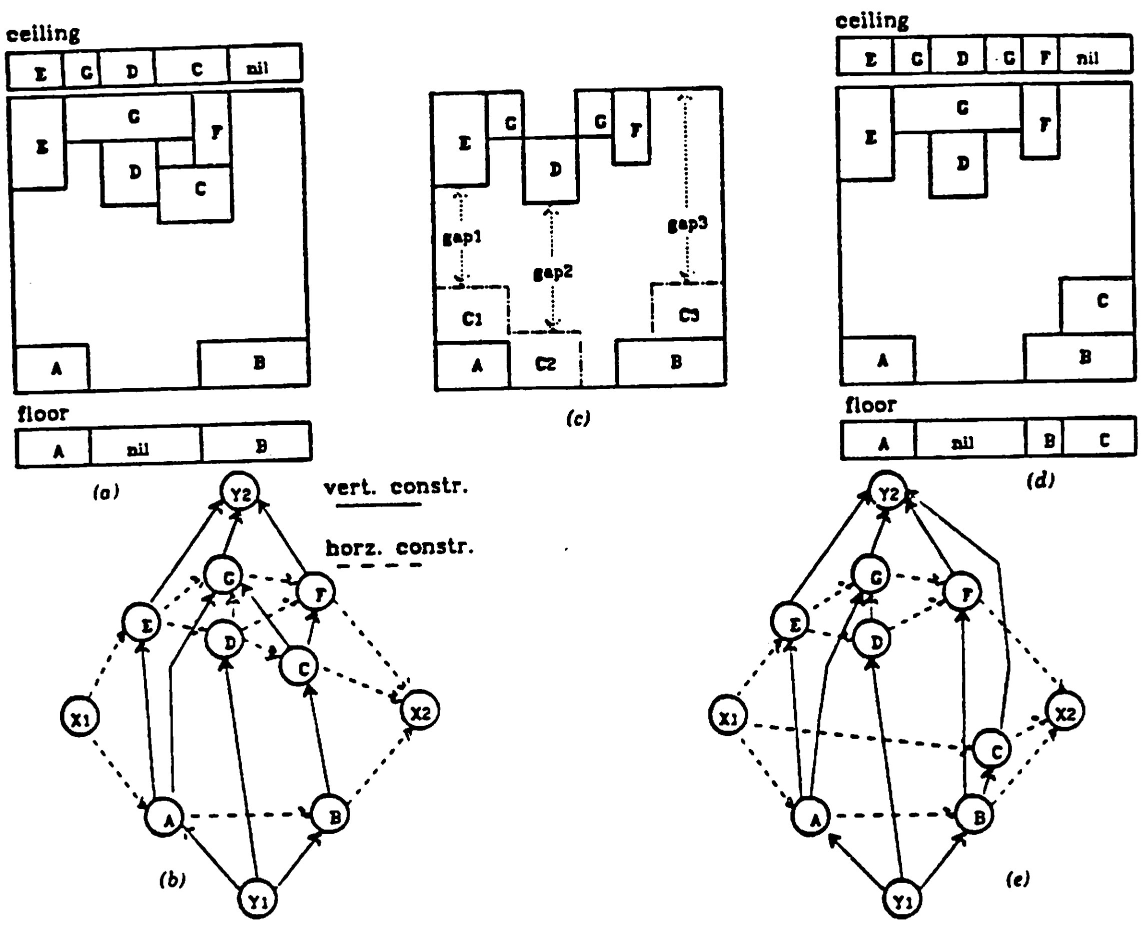


**Figure 3.3.** *Example of box packing in progress.*
*(a) Actual box constellation and floor and ceiling data structures. (b) Corresponding adjacency graph. (c) Box C has been selected to be moved. Three candidate places C1, C2, C3 are evaluated. (d) Box constellation after box C has been placed and new floor and ceiling structures. (e) Updated adjacency graph.*

### 3.3.1. XY Adjacency Graph

The positions of all blocks during the compaction process are represented in the form of two superposed graphs, $G_x$ $(V_x, E_x)$ and $G_y$ $(V_y, E_y)$, where $V_x$ and $V_y$ are sets of nodes and $E_x$ and $E_y$ are sets of arcs. Consider the nodes of the graphs, $V_x = \{ x_1, x_2, V \}$ and $V_y = \{ y_1, y_2, V \}$. $x_1$ and $y_1$ are source nodes corresponding to the left and the bottom edges of the bounding rectangle, respectively. $x_2$ and $y_2$ are sink nodes corresponding to the right and the top edges of the bounding rectangle, respectively. Each node $v \in V$ is in one-to-one correspondence with each box. The arcs in the graphs are added such that $(v_i, v_j) \in E_x$ if the box corresponding to $v_i$ is to the left of the box corresponding to $v_j$. Similarly, $(v_i, v_j) \in E_y$ if the box corresponding to $v_j$ is on top of the box corresponding to $v_i$. These arcs can be labeled with the minimal allowable horizontal or vertical separations between the centers of the block, and this adjacency graph can thus be turned into a constraint graph for properly placing the blocks without overlap. For an actual circuit layout, the constraints attached to these arcs can become more complicated and contain upper as well as lower bounds (more on that in the next section).

### 3.3.2. Zone-Refining Algorithm

The main loop is: while the top is not empty, select individual boxes according to some *selection* function, move them from the top to the bottom, and place them on the floor according to some *placement* function. In the following subsections, we describe the individual modules of the box packing algorithm.

#### 3.3.2.1. Initialization

A good starting configuration can be achieved with a simple one-dimensional compaction from bottom to top. At the start of zone-refining, the floor is thus flat while the ceiling is the profile of the precompacted block constellation.

#### 3.3.2.2. Selection

One of the blocks that currently borders the ceiling is selected for transfer across the zone. Several rules can be used to choose the block. The rule we implemented is to choose one of the blocks that defines the lowest point in the ceiling. If there is a tie, the left-most block is selected (alternatively a random choice could be made) and is removed from the top.

#### 3.3.2.3. Placement

For the selected block an optimal position must now be found on the floor. For different $x$-positions the resulting minimum gap of the free zone is evaluated. The position that maximizes this gap is chosen. At this point, there are many possibilities to choose an appropriate cost function. If the original ordering of the blocks should be modified as little as possible, a hard limit on the maximum lateral displacement can be set. Alternatively, a cost can be associated with any displacements in $x$-direction that is traded off against the gain in zone-gap.

To find the optimal place for box $C$ in Fig. 3.3, for example, it is only necessary to compare the three candidate places shown in Fig. 3.3(c). The left-most positions are chosen as representative candidate places for each span of $x$-positions leading to the same final zone-gap. Since gap 3 is the largest, box $C$ is placed as shown in Fig. 3.3(d). When evaluating different placements of a box, there may be more than one location that yield the maximal improvement. In our approach, the left-most position is then selected.

Once the optimal position has been determined, the block is placed correspondingly and the adjacency graph is updated (Fig. 3.3(e)).

### 3.3.2.4. Updating the XY Adjacency Graph

As the box is being moved across the zone, the adjacency graph is modified to correspond to the new situation. For every move of the box in the $x$-direction, we need to modify the $y$-arcs, and vice versa. For the sake of simplicity, we will explain this updating operation only for movements in the $x$-direction. Updating the graph for movements in the $y$-direction is analogous with the roles of $x$ and $y$ interchanged.

If an instance is moved in the $x$-direction, then all the $y$-arcs incident to the node are removed from the $y$-graph, and new arcs are generated between its former predecessors and successors, if necessary. For example in Fig. 3.3(e), a new arc is generated between box $B$ and box $F$ after the placement of box $C$. Finally arcs are generated from the moved node to the other newly adjacent nodes. . This process is performed as follows. The horizontal edges of the boxes are recorded as intervals and sorted in increasing $x$ and increasing $y$. To find which arcs to add to the graph amounts to finding intersections among the intervals corresponding to the boxes. An arc is added between the moved node and another node in the upper half of the graph, if there exists a straight vertical line from the top edge of the moved box to the bottom edge of a box in the ceiling; and similarly for the bottom half of the graph.

Since this operation is in one of the innermost loops of the algorithm, an efficient search algorithm for this adjacencies is crucial. A new tiling approach is explained in Section 3.4.1.2.

### 3.3.3. Sample Results of Box-packing Algorithms

Fig. 3.4 demonstrates the results that can be achieved with zone-refining for the simple box-packing task.

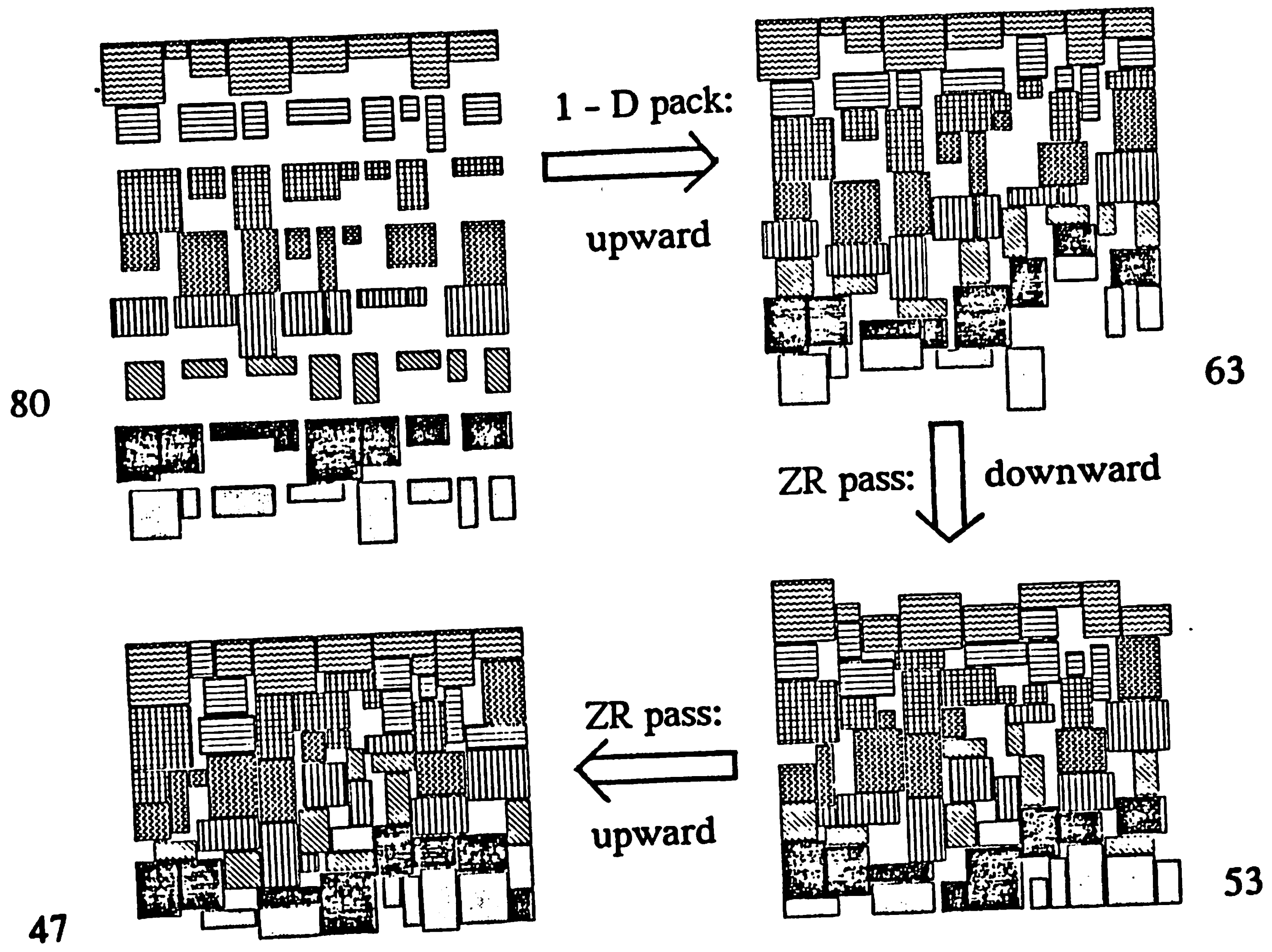


**Figure 3.4.** *Example of box packing : (a) 1-D precompaction left to right with averaging of the lateral positions ( void-space = 47.8 % ). (b) Precompaction is completed with a 1-D upward compaction ( void-space = 33.7 % ). (c) After first zone-refining pass in the downward direction ( void-space = 21.2 % ). (d) After one more upward zone-refining pass - convergence has been reached ( void-space = 11.1 % ).*

We start with an 8 by 8 array of boxes, sized randomly in both dimensions in the range of 3 to 10 units. The starting configuration for zone-refining is reached by first performing a compaction step to the right in which the block positions are averaged within the given slack (Fig. 3.4 (a)); this is followed with a simple compaction step upwards (Fig. 3.4 (b)). Fig. 3.4 (c) shows the results after the first zone-refining pass downwards. The height is reduced to 53 units, that is 84 % of the precompacted constellation (63 units in Fig. 3.4 (b)). In the next upward zone-refining pass, the height of the packed bin settled at 47, as shown in Fig. 3.4(d). The void-space occupies only 11.1 % of the total area. The whole process takes 4 seconds on a DEC VAX 11/785 running 4.3BSD UNIX.

## 3.3.4. Pitch Matching

When designing a component of a standard cell library or a cell in a full custom chip where the height or width of a circuit block is required to have a specific value, it is necessary to compact the layout so that the constraints on the size of the cell are satisfied, i.e. pitch matching has to be accomplished. To be able to satisfy this kind of constraints two-dimensional movements are essential. As a part of compaction by zone-refining, the width of the layout can be changed to a desired value. This is demonstrated in Fig. 3.5. The width of the bounding bin has been fixed to two different values, 76 and 50. In both cases, rather compact layouts are obtained after 3 and 5 zone-refining passes, respectively.

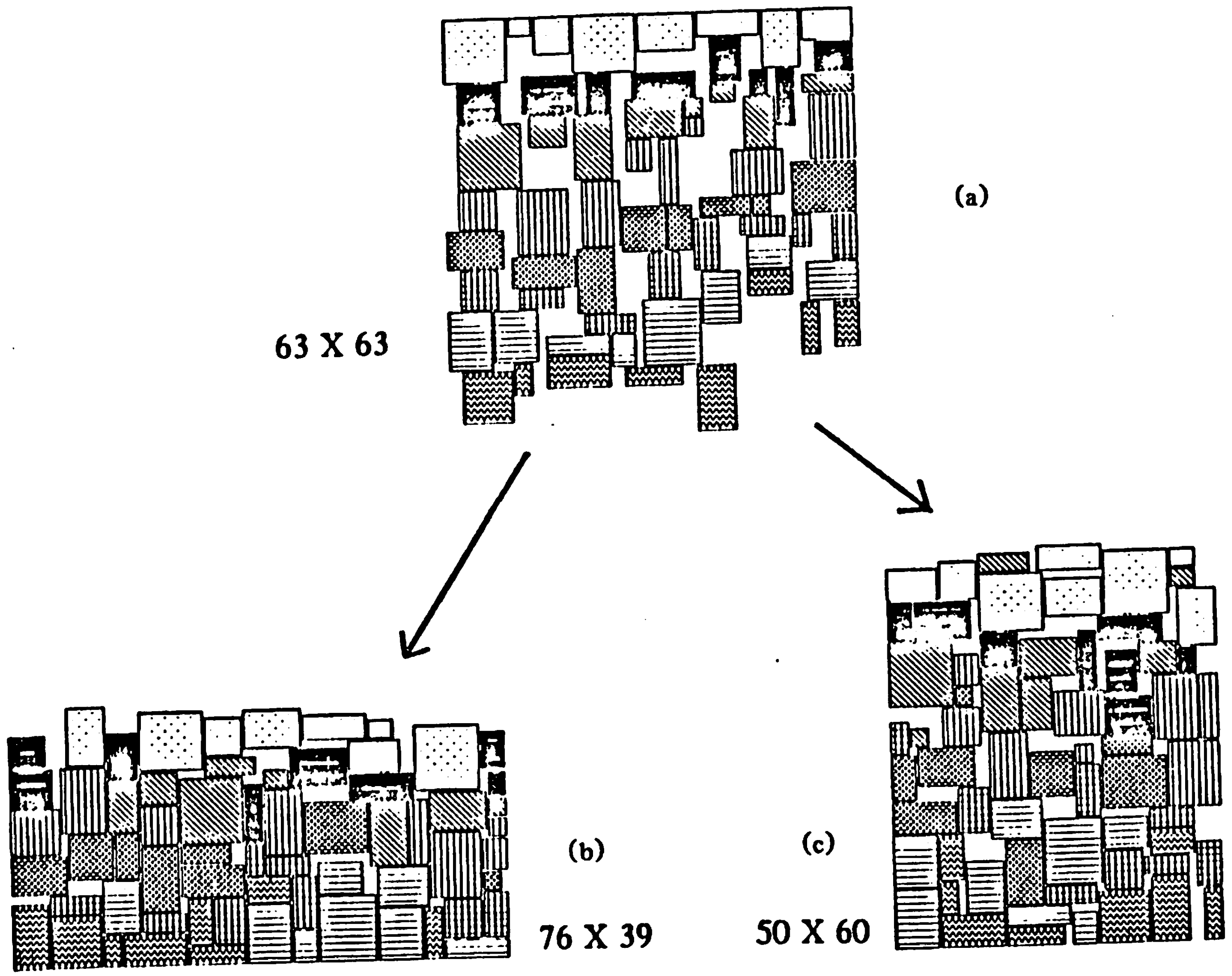


**Figure 3.5.** *Pitch Matching : (a) original layout,*
*(b) width has been increased by 20%, (c) width has been decreased by 20%.*

## 3.4. CIRCUIT LAYOUT COMPACTION

Zone-refining of an actual circuit layout follows the same basic algorithm as box-packing. However, its implementation is much more complicated than in the case of box-packing for several reasons:

1 There is more than one level of interconnect. Thus we need to consider proper spacing in all of these levels, while maintaining the proper geometric relationships between levels for individual components.

2 The presence of complicated components such as transistors and inter-level contacts that span several levels and the existence of sophisticated geometrical layout rules leads to more complicated constraints for the placement of components. For each arc there are now more complicated expressions than a simple minimum separation. There can also be upper bounds as in the case when a connection must touch a circuit block within the extent of a contact area of a certain size.

3 The components in a circuit layout are connected by wires that are neither infinitely thin nor infinitely flexible. This tends to limit the degree to which individual components can be moved and thus the degree of topological rearrangement possible. For a full exploitation of the possible two-dimensional movements, jogs need to be inserted into the wires at appropriate places.

As a consequence of all the points above, components cluster into groups that should be moved as a whole. There are additional difficulties associated with discovering these groups, moving them around, finding an optimal place for them, and dealing with the multitude of wires connected to them.

We will discuss our solutions to these problems as we discuss the individual phases of the zone-refining algorithm for circuit layouts. All circuit elements such as transistors, various contacts, and horizontal wires, in the following called *components*, are handled by the same data structure. Also an *instance* is defined as any component that is not a wire.

### 3.4.1. Data Structures

Because of the increased complexity associated with real circuits, we use data structures for circuit compaction that are slightly different from the ones used in box-packing. For example, since wires can stretch or shrink during compaction, horizontal (vertical) wires are not represented in the horizontal (vertical) constraint graph. Instead, the length of a wire is implicitly given by the distance of the two instances to which it is attached.

#### 3.4.1.1. Floor and Ceiling

A circuit layout consists of several layers. The first data structure used to implement the zone-refining algorithm (Zorro1 in table 3.1) maintained separate linked lists for all the elements forming the floor and the ceiling for each independent set of interconnection layers for an efficient look-up. However, this approach is cumbersome for circuit layouts. Many sets of floor and ceiling lists are required and they all must be updated after every component move. Therefore, in the present version (Zorro2), we do not keep explicit lists for the floor and the ceiling, but we search the constraint graph $G_y(V_y, E_y)$, whenever necessary. The floor and ceiling components are readily found from the information in the constraint graph by searching the arcs below and above the vertices corresponding to the components currently being placed. Since we do not keep the floor and the ceiling lists explicitly but find them from the constraint graphs, there does not exist any limitation on the number of independent layers this algorithm can handle.

### 3.4.1.2. Coarse Grid Structure

The updating of the constraint graphs is one of the most frequent, and overall the most expensive operation in constraint graph-based algorithms. T. Hedges et al. [27] and C. Kingsley [47] reported that "intervening" method is faster than commonly used shadowing method. However, the "intervening" feature loses its efficiency when it is used for incremental graph modification. Recently, a more sophisticated incremental constraint generation has been published [12]. This approach is based on shadowing and corner-stitching [70]. It is faster than sorted every-pair constraint generation, but is more complicated.

To optimize performance of the zone-refining algorithm, we devised new simple data structures that permit fast updating of the adjacency graphs. The basic idea is to sub-divide the layout area into tiles defined by a set of horizontal and vertical grid lines (see Fig. 3.6). All components are linked to all the tiles they touch and are dynamically reattached after every move.

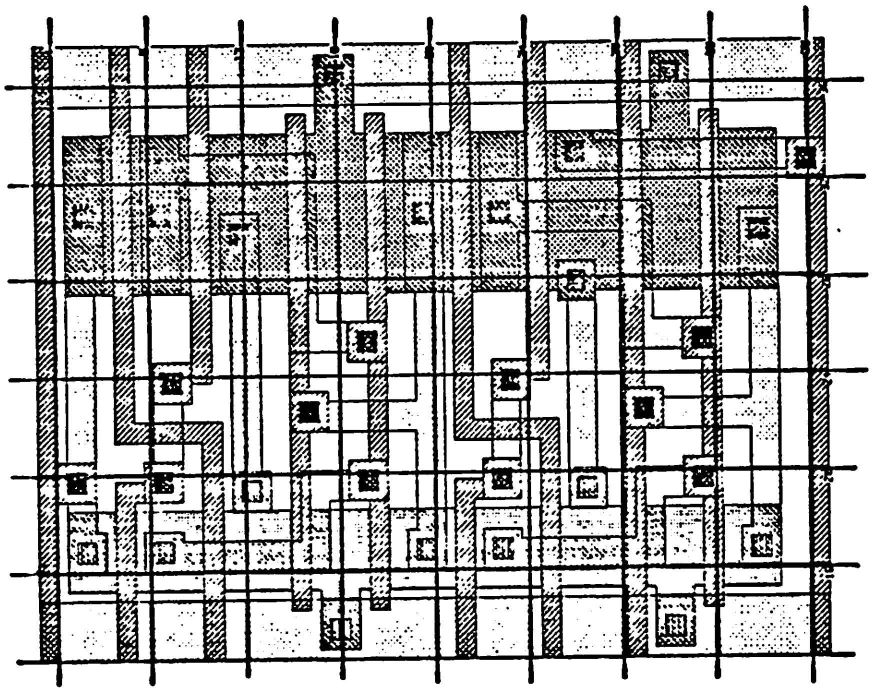


Figure 3.6. *The use of space-partitioning in constraint graph construction.*

When a component is moved, only the tiles neighboring the tiles containing the moved component are checked to update the graphs. The set of tiles that need to be checked depends on the *extent* of the move of the component. We limit this extent to update the graphs efficiently. The extent of a simple move is limited to $\alpha * S_a + \beta * R_m$ where $\alpha$ and $\beta$ are parameters ($\alpha = 4$, $\beta = 2$ are currently used). $S_a$ is the average width of the instances if the move is in the horizontal direction. If the move is in the vertical direction, $S_a$ is the average height of the instances. $R_m$ is the maximum spacing rule between any two components.

By adjusting these parameters, the user can trade-off the performance and CPU time of a pass of zone-refining. Large values of $\alpha$ and $\beta$ allow the components to move longer distance in the free zone and may produce a better solution at the expense of increased run time. The above values are chosen since there is rarely a reduction of the vertical dimension by more than this limit, for a typical precompacted layout. Even if the possible reduction of the height of a layout is larger than this limit, one can achieve the maximum reduction by repeating the Z-R passes until no further reduction is possible. Note that the height can be reduced up to this limit at each pass of zone-refining.

If the size of the tiles is too small, large components intersect with many tiles and hence moving these components requires updating a large number of tiles. If the size of the tiles is too large, a tile may contain many components and this degrades efficiency since all the components touching a tile are stored as a linked list. If we assume that the components are uniformly distributed throughout the layout, it appears reasonable to choose the number of grid lines as a function of the number of instances in the layout. With this choice, the space between grid lines is large when the layout is sparse (like in sticks diagram or on virtual grids), and it gets reduced as the layout becomes denser. The size of tiles is evaluated at the beginning of each Z-R pass and remains fixed during a whole Z-R pass.

The number of horizontal and vertical grid lines are given by the following formula.

$$n_{xgrid} = \lceil \text{sqrt}\,(K_g * n_i * W_c / H_c) \rceil$$

$$n_{ygrid} = \lceil \text{sqrt}\,(K_g * n_i * H_c / W_c) \rceil$$

where $n_{xgrid}$ and $n_{ygrid}$ are the number of grid lines in horizontal and vertical direction respectively, $K_g$ is a user defined parameter, $n_i$ is the number of instances in the layout, $W_c$ is the width of the given layout, and $H_c$ is the height of the given layout. Note that the total number of space tiles is roughly $K_g\ n_i$ and that $n_{xgrid}/n_{ygrid}$ = $W_c/H_c$.

Table 3.1 shows the dependences of CPU time (in seconds on a VAX 785) on the parameter $Kg$ for two example circuits. The dependence of CPU time on $K_g$ is not critical for a moderate range of $K_g$. In the current implementation, $K_g = 2$ has been chosen.

| | $n_i$ | $K_g = 1$ | $K_g = 2$ | $K_g = 3$ | $K_g = 4$ |
|---|---|---|---|---|---|
| ex1 | 89 | 69 | 65 | 71 | 79 |
| ex2 | 317 | 278 | 276 | 279 | 288 |

Table 3.1 : Run-times on different values of $K_g$

### 3.4.1.3. Permanent and Non-permanent Arcs

There are two types of arcs in a constraint graph [4]. Permanent arcs represent electrical connectivities that must be maintained throughout the compaction. For example, a wire connected to a terminal area must always remain attached to the terminal area and a permanent arc is necessary to constrain the spacing between the wire and the terminal. The non-permanent arcs represent spacing constraints due to design rules and are modified when the corresponding components are moved. For example two different contacts may or may not have a spacing constraint depending on their relative position and a non-permanent arc is added between the two contacts whenever necessary.

### 3.4.2. Initialization

As in box-packing, a good starting point is the result of a one dimensional compaction step. If the layout needs to fit into the fixed bin dimension of a standard cell system, then we use a constraint-graph based one-dimensional compaction algorithm and compress the circuit in the $y$-direction to the desired height, keeping all components roughly in the middle of their allowable range of $y$-positions. (If the desired height can not be achieved by 1-D algorithm, we may apply Z-R compaction in $y$-direction first to further reduce the height of the layout.) We then apply a one-dimensional pre-compaction in the $x$-direction to create the starting constellation for zone-refining compaction in the $x$-direction.

For the purpose of the following discussion, consider the cell turned on its side, so that Z-R compaction can proceed from *top* to *bottom* as outlined in the previous sections.

### 3.4.3. Cluster Selection

As pointed out above, in the case of a circuit layout, components have to be moved as *clusters*. Since the movement in each Z-R pass is two-dimensional, there are two clusters, the $x$-cluster and the $y$-cluster; the $y$- cluster is a subset of $x$-cluster. To find the clusters, we first find a *seed* node (component) that is responsible for the lowest $y$-value in the current ceiling profiles. Then, all the nodes that are semi-rigidly coupled to this *seed* node in the graph $G_y(V_y, E_y)$ are added to the $y$-cluster. (A pair of nodes are "semi-rigidly" coupled if there is an arc between the nodes whose upper and lower bounds differ by less than *ybound*, i.e. *upperbound* - *lowerbound* < *ybound*.) All the nodes that are semi-rigidly connected to some node in the $y$-cluster are added to the cluster by recursively continuing the search on the newly added nodes. For this selection, we may do either breadth-first or depth-first search on the $y$-constraint graph. The resulting cluster of nodes is called the $y$-cluster and is moved across the zone as a whole.

The $x$-cluster includes all the nodes in the $y$-cluster and possibly some additional nodes that are "semi-rigidly" connected to the nodes in the cluster in $G_x(V_x, E_x)$, i.e. *upperbound* - *lowerbound* < *xbound*. The purpose of the $x$-cluster is to give more degrees of freedom for the nodes in the $y$-cluster to move in the $x$-direction. If these nodes were not included in the cluster, $x$-movements of the nodes in the $y$-cluster would not be possible without introducing jogs on vertical wires. By changing the value of *xbound* dynamically as a function of zone-gap, we can control the flexibility of the movements in the lateral direction in the molten zone. In the current implementation of our algorithm, we use a larger value of *xbound* if the current $y$-cluster contains a critical component that defines the minimum gap between *floor* and *ceiling*.

During each placement step, the node positions in the $y$-cluster may change their positions in the $y$-direction and the node positions in the $x$-cluster may change their positions in the $x$-direction. The positions of all the other nodes are fixed during the placement of the cluster.

### 3.4.4. Cluster Placement

If the cluster is not considered a completely rigid object but a collection of components that can be moved with respect to one another, then the determination of an optimal place on the given set of floor profiles is by itself a hard problem (probably

NP-hard). Thus some heuristic restrictions have been made.

There may be many options for implementing the placement routine of zone-refining. Here are some of the decisions that can change the performance and complexity of the Z-R algorithm.

1. To what extent in the $x$-direction should movements in the free zone be considered ?

   As explained in Section 3.3, there is a trade-off between the limitation of the movements and run-time.

2. How much flexibility should be put into the wires ?

   Horizontal and vertical wires could be bent by introducing jog-points, and wires could be straightened by removing unnecessary jogs.

3. To what extent can the topology of the circuit be changed ?

   For example, one may allow changing the orientation of components, permit merging of contacts and wire-segments, and reroute part of the connections among circuit components to further optimize the cost function.

The above features have been implemented to various degrees in the present version of Zorro. We now describe the basic heuristics for placing the clusters and then explain more sophisticated extensions in the following subsections.

1. First the $y$-cluster is moved into the "middle" of the zone with a straight $y$-move. The middle of the zone is defined as the $y$-position where the cluster can be moved freely left and right if it were not for the attached wires. Then the minimum zone-gap for the current $x$-positions is determined and saved together with the current $x$-coordinates of all components as initial values for the optimal placement of the cluster.

2. The next step is to determine the range of $x$-positions to be evaluated for the $x$-cluster placement. To avoid an exhaustive search of all positions, the current $x$-cluster components are first compacted to their right-most position (Fig. 3.7(b)) and then to their left-most position (Fig. 3.7(a)) that are possible without generating new jogs in any wires. These positions are constrained by the vertical wires attached to the components as well as by other vertical wires crossing the zone.

3. The cluster is positioned on the profile of the floor in its left-most position and the minimum zone-gap is recorded together with the critical component in the cluster, i.e., the component responsible for the minimum distance from the ceiling profile.

4 The next candidate place for the critical component considered is the nearest rightward place where the ceiling profiles go up or the floor profiles go down on the layers corresponding to the critical component. If no further candidate place is found, go to 7, else go to 5.

5. An attempt is made to move the critical component to the candidate place. To make this possible, we may have to move other components in the $x$-cluster. If this movement is possible go to 6, else go to 7. (If this move is not possible due to constraints in the $x$-direction, it is not necessary to try other candidate places since they are to the right of the current position, and the current critical component can not be moved to the right of the current candidate place.)

6 For the current $x$-positions of all components in the zone, the minimum gap and the critical component are determined. If this gap is larger than the previously stored best value, we update the value and save the current $x$-positions as the best solution found so far. The procedure is continued by going back to step 4.

7 We place the components at the optimal $x$-positions saved (Fig. 3.7(c)).

The algorithm has a loop (4 -> 5 -> 6 -> 4) and it is legitimate to ask whether the algorithm terminates in a finite number of steps. It indeed does so because the width of the layout is finite, the design rules are discrete, and hence the number of candidate places is finite.

A simplified example of a cluster placement is demonstrated in Fig. 3.7. Fig. 3.7 (a) and (b) show the extreme positions of search-range for the candidate places. The final settlement is shown in Fig. 3.7(c). This placement is chosen since the ceiling is highest for the critical block and the minimum zone-gap is maximized. Note that wires can slide within the terminal area and jogs can be generated after placement.

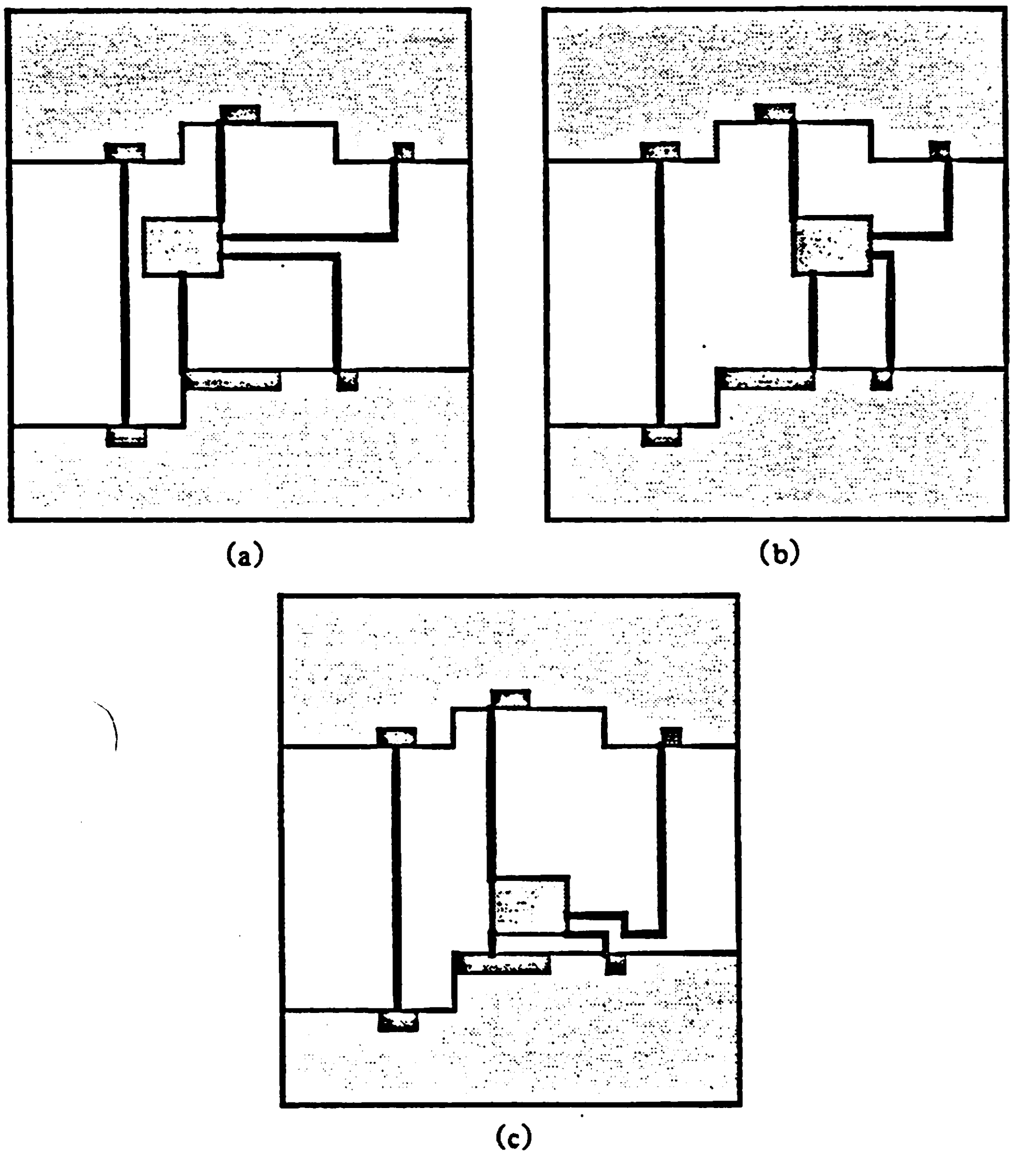


**Figure 3.7.** *An Example of Placement. (a) left-most position for the cluster. (b) right-most position for the cluster. (c) final settlement.*

The algorithm finds a locally optimal placement if each component is placed individually. However, for the case of component clusters there are situations where the algorithm produces sub-optimal local solutions. One such case is shown in Fig. 3.8. Block $Y$ is the critical component, since gap 2 is less than gap 1. We may increase the gap either by shifting $X$ to the right of $x1$ or by shifting $Y$ to the right of $x2$. Our current implementation can deal only with the latter move and hence it does not find the locally optimal solution that corresponds to the former move. Note that the former move could be added in steps 4 and 5, or that a better placement can be achieved by introducing a jog in the wire between the components X and Y (see Section 3.4.6.2).

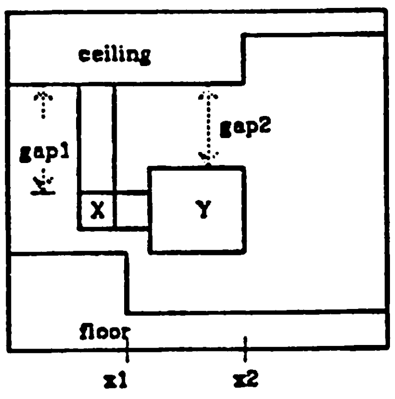


Figure 3.8. *An Example of Sub-Optimal Placement.*

In step 6, a more complicated cost function could be used to take into account additional factors such as a maximal allowed offset in the $x$-direction or limits on the maximum length of polysilicon wires.

### 3.4.5. Updating the Constraint Graph

The basic idea for updating the adjacency graphs is essentially the same as for box-packing. One of the most notable differences is the fact that we are using a coarse grid structure for efficient generation of the constraint graph. Another major difference is the possibility of merging terminals belonging to the same net (same electrical node in the lumped circuit diagram). Merging can be implemented by not generating any constraint between components that can be merged. However, this makes the generation of the constraint graph substantially more complicated. When merging is not an option, we can generate the constraint graph easily by checking neighboring components only, using a scan-line approach, for example (see Fig. 3.9 (a)). However, if merging components of the same net is considered, then the constraints depend on net-list information and the search may have to be extended beyond the nearest neighboring components. For example, in Fig. 3.9(b), component C is not a neighbor of X, but the arc between X and C is still needed. This is because X and B can overlap in Fig. 3.9(b), while B and C can overlap in Fig. 3.9(c). In the figure, net numbers are shown in parentheses.

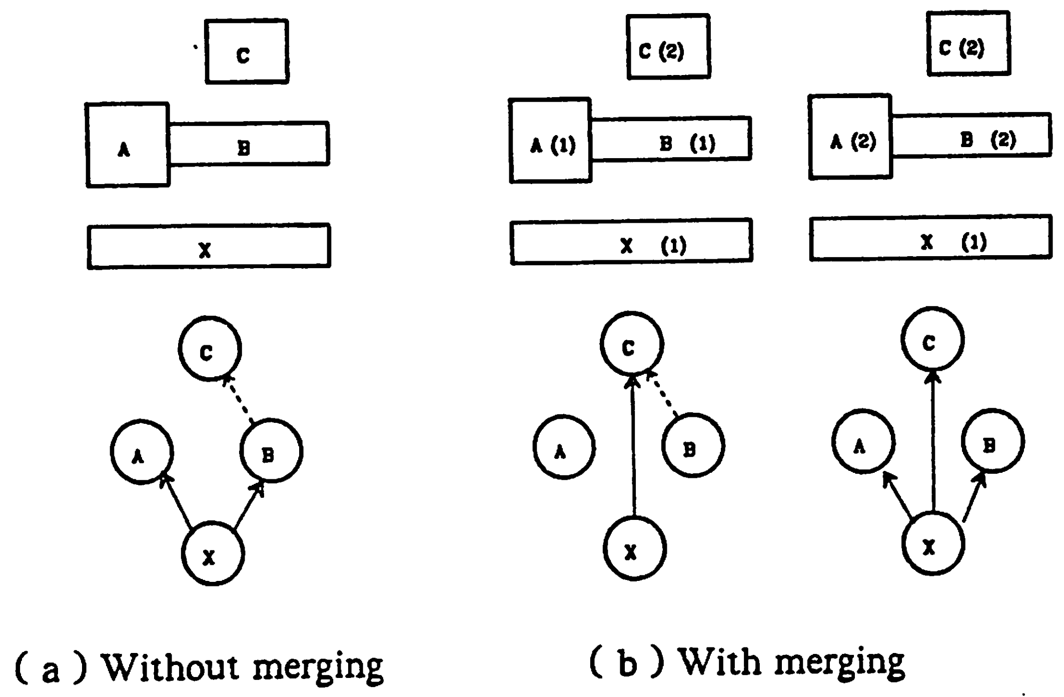


Figure 3.9. *Constraint graphs with and without merging.*

In our constraint graph, we include arcs between all component pairs that lie closer than *exten* and which have some spacing constraints from design rules. With such a graph we can easily evaluate the legality of a particular component placement by simply looking at the constraint graph. This approach may generate redundant arcs in the constraint graph. (Redundant arcs are the arcs which can be deleted without affecting the correct spacing.) However, we observed experimentally that finding a set of arcs that is minimal with respect to correct spacing requires more CPU time in generating the constraint graphs when used in incremental modifications. Generating all arcs between two components within some given distance has at least three advantages:

1 The complexity of constraint graph generation is independent of net-list information. Constraining arcs are generated whenever two components lie within a given distance and have a spacing constraint.

2 When a cluster of components is moved, we only need to update arcs connected to the nodes corresponding to the moved components. Otherwise we would have to search the data structure for places where new arcs might originate. For example, Fig. 3.3 has a minimum set of arcs, and a new arc from B to F has to be created after C has been moved in Fig. 3.3 (e).

3 We can find the exact profiles of the floor and ceiling by scanning the arcs connected to the components of the current cluster. A minimum set of arcs would give only the most critical part of floor and ceiling.

The price for using a non-minimal set of arcs is that graph-solving (finding legal positions for all nodes) will take more time. However, the graph-solving task usually takes less than 3 % of the overall CPU time, since only a small number of components are moved at once and solving a graph for a small set of nodes is very fast.

The two implementations of zone-refining are compared in Table 3.2 for a latch driver circuit [87]. In the table, profile of run times on a DEC MV8650 under the Ultrix operating system is shown in seconds. Zorro1 is an older version which uses data structures very similar to the one described for box-packing and maintains explicit lists of floors and ceilings. Zorro2 is a new version with the coarse grid data structures described above. Note that constraint graph generation dominates execution time, and that most of the reduction in CPU time has been obtained from this part. Other routines also have been optimized in Zorro2, and overall the new version is 5 to 10 times faster.

| five passes of Z-R with jog-generation | | | | |
|---|---|---|---|---|
| function | Zorro1 | ( % ) | Zorro2 | ( % ) |
| graph construction | 1187.5 | ( 60.0 ) | 121.4 | ( 58.2 ) |
| edge/grid updating | 35.7 | ( 1.8 ) | 7.0 | ( 3.3 ) |
| graph solving | 14.5 | ( 0.7 ) | 5.8 | ( 2.8 ) |
| mcount | 286.9 | ( 14.5 ) | 49.6 | ( 23.8 ) |
| others | 459.4 | ( 23.2 ) | 24.7 | ( 11.8 ) |
| total | 1984.0 | ( 100.0 ) | 208.4 | ( 100.0 ) |

Table 3.2 : Run-time comparisons between Zorro1 and Zorro2

Now we describe the algorithm used to modify the constraint graph. Since the modification of $G_y(V_y, E_y)$ is similar to the modification of $G_x(V_x, E_x)$, only the routine *modify_xgraph* is described. It is composed of two subroutines: In the first one, all the arcs that might be affected due to movements of nodes in the $y$-cluster are deleted from the $x$ constraint graph $G_x$. Note that $G_x$ has to be updated after moving vertices in the $y$-cluster in the $y$-direction. In the second subroutine *new_xarcs*, all the necessary new arcs are generated. The detailed algorithm of *new_xarcs* is given below. In the algorithm, the terms *predecessor* and *successor* refer to the "from" and "to" nodes of an arc, respectively.

```
new_xarcs( exten )
/*
 * create new arcs to maintain a legal layout.
 */
{
    for( each vertex vx corresponding to a vertical wire connected to a component
        whose vy is in the ycluster ) {
            /* add constraints to successors of vx */
            find maxrule which is a maximum possible spacing
               from the component vx->comp;
            find a list ilist of components which touch the space
               tiles with distance < exten + maxrule,
               in the positive x-direction;
            for( each component comp of ilist ) {
                  if( there is a constraint between vx and comp->vx ) {
                        if( an arc has not been added )
                              add an arc from vx to comp->vx;
                  }
            }

            /* add constraints to predecessors of vx */
            find a list ilist of components which touch the space
               tiles with distance < exten + maxrule,
               in the negative x-direction;
            for( each component comp of ilist ) {
                  if( there is a constraint between vx and comp->vx ) {
                        if( an arc has not been added )
                              add an arc from comp->vx to vx;
                  }
            }
      }
      for( each vertex vy in ycluster ) {
            vx = vy->comp->vx;
            if( vx != $\phi$ ) {
               add constraints to successors of vx as above;
               add constraints to predecessors of vx as above;
            }
      }
}
```

### 3.4.6. Optimizing the Connections

As pointed out above, the presence of wires limits the extent to which components can move. Collecting more of the nodes that are restricting lateral motion into the $x$-cluster increases the number of possible moves at the cost of complicating the algorithm. To give more degrees of freedom, we have implemented sliding connections, overlapping contacts, extra jogs, and wire-length reduction, as detailed in the following subsections.

### 3.4.6.1. Sliding Connections

To obtain more flexibility for 2-D movements, wires connecting to contacts or ports on other circuit components are allowed to slide freely within the extent of these connection regions. Thus a wire of width $w$ attached to a port of extent $p$ can move over a range $(p-w)$, as can be seen in Fig. 3.10.

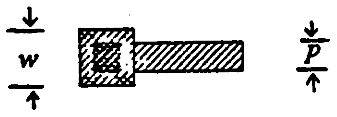


Figure 3.10. *Sliding connection.*

### 3.4.6.2. Automatic Jog Introduction

Another mechanism that provides even more flexibility for components to be moved, is the generation of new jog-points in the constraining wires. Hsueh [37] proposed the use of a 'torque' to introduce a jog point. Wolf [104, 106] analyzed the effect of jog introduction to compaction and found that jogs introduced to improve density along one axis in general increase the dimension in the other direction. F. M. Maley [63] proposed another interesting approach of jog introduction: all wires are converted into constraints on the positions of the active devices. However, for two-dimensional incremental modifications, it is necessary to reroute the wires after each movement. Therefore, this method may be too expensive for compaction by two-dimensional movements.

General jog insertion is a hard problem and has not yet been implemented. We have only incorporated two situations in which new jogs are automatically introduced. Both of them concern horizontal wires (see Fig. 3.11).

After a cluster has been placed as close to the floor as possible, automatic jog insertion is performed on horizontal wires attached to it. Down-ward compaction of one of the cluster components may be impeded by the existence of such wires as shown symbolically in Fig. 3.11(a). If the wire in this figure is jogged, then the down-ward movement of the component may be continued. Thus we check whether underneath any of the cluster components there is extra space into which the component could fit. If this is the case, we try to generate jogs that permit the component to slide into this space. These additional movements have two effects, one is to increase the minimum zone-gap if the component under consideration is a critical one (Fig. 3.11(a)), the second is to make more room for the next row of components to be placed on the floor.

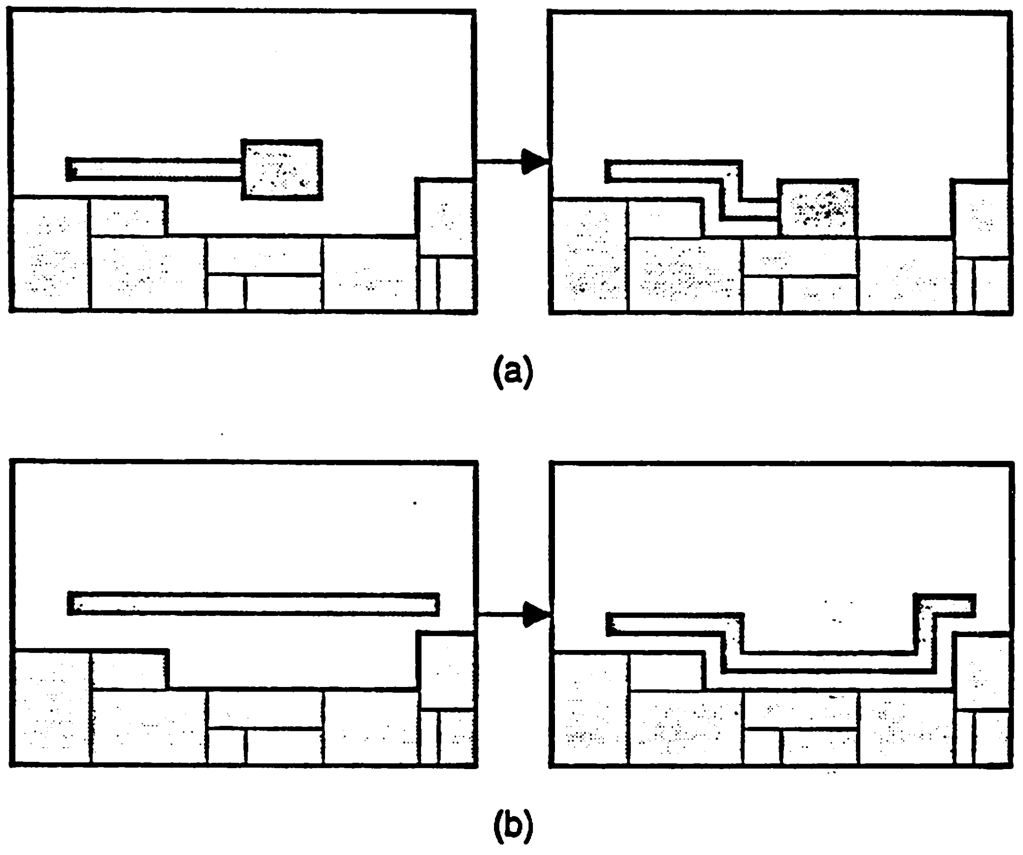


**Figure 3.11.** *Two cases of automatic jog introduction.*

A second situation is that of long horizontal wires stretching above an empty "well" (Fig. 3.11(b)). In this situation the wire is pushed down into the well with the simultaneous introduction of two jogs in the hope that the extra notch in the floor will be large enough to accommodate a component from a cluster to be placed subsequently.

If there is more than one horizontal wire attached to a component, we begin by moving the lowest wire first. We jog wire after wire to fit the floor profiles. This approach overcomes the hard problem of generating "parallel jogs" in wire bundles [13] and can produce patterns that are similar to the ones generated by a river router (see Fig. 3.12 which is a portion of Fig 3.19 (c)).

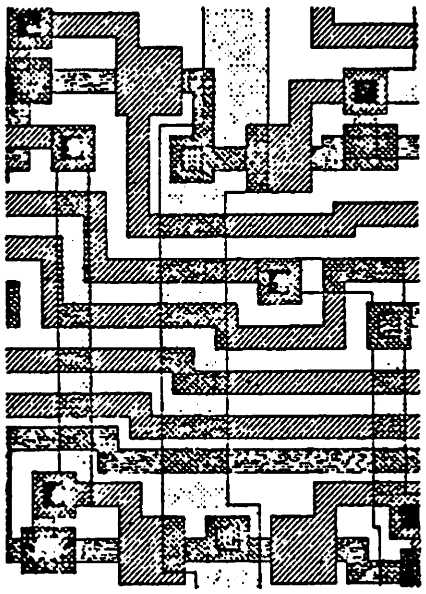


**Figure 3.12.** *Jogged wire bundles after zone-refining with automatic jog introduction.*

Note that jogs for vertical wires can be generated by applying zone-refining from *left* to *right* (or from *right* to *left*), in a similar approach as above.

### 3.4.6.3. Merging Connections of the Same Net

Reducing the number of actual components is a powerful mechanism to reduce the area required. Contacts that belong to a same net are allowed to merge into a single contact. The "mergeability" information for each type of contacts is read from a technology file. In our implementation, merging is a two-step process. These steps are illustrated using a simple example in Fig. 3.13. In the starting configuration (Fig 3.13. (a)), there are two contacts of the same net that can be merged. First the contacts and wires of the same net are allowed to move closer than the minimum separation rule would allow and can even overlap partially (Fig. 3.13 (b)). In the second phase, the algorithm checks if full overlap can be achieved between the two contacts. If complete overlap is possible, then it removes one contact and the unnecessary wire-segments, and reconnects the wires originally attached to the removed contact (Fig. 3.13 (c)). Now further compaction can be performed to reduce the area (Fig. 3.13 (d)).

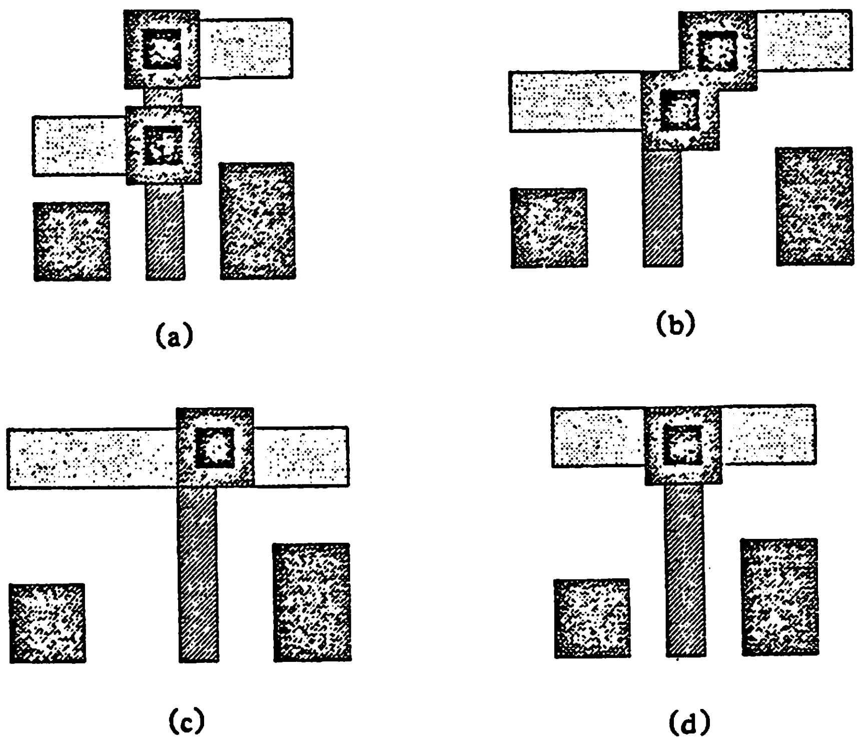


Figure 3.13. *Merging contacts. (a) initial layout with spacing violation. (b) partially merged contacts. (c) one contact is removed. (d) further compaction is applied.*

### 3.4.6.4. Reducing Wire Length

Long wires use more area and degrade the electrical properties of the circuit. In Zorro, we do not have a general wire-length reduction algorithm. Instead, several different simple strategies are used to keep wires short.

Before we start the zone-refining process, we minimize the length of horizontal wires by two special "compaction" steps in the $x$-direction. First we compact left-ward all nodes that have no electrical connections with right-side neighbors, then we compact right-ward the nodes that have no electrical connections with left-side neighbors.

During zone-refinement, we minimize horizontal wire length with every move of a cluster of components across the zone. All moves involving the nodes in the current $x$-cluster are biased left-ward during placement, and after the optimal cluster position has been found, the nodes with no left-side connection are compacted right-ward. Similar steps can be employed in the $y$-direction between or after Z-R passes.

Another important routine is the clean-up of unnecessary jogs and wire-segments. After each pass of zone-refining, each wire is checked whether its length can be reduced to zero by moving a set of components connected to the wire. If this is possible, we remove the wire whose length is zero and reconnect affected wires and components. This routine removes all unnecessary U-shape connections since they contain at least one wire whose length can be reduced to zero (see Fig. 3.14).

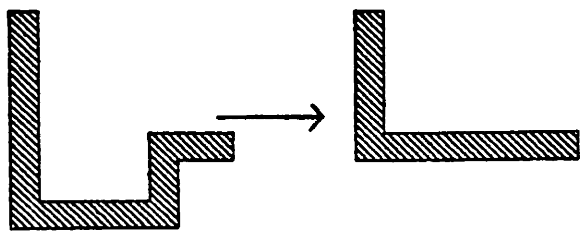


Figure 3.14. *Clean-up removes an U-shape connection.*

More work would be necessary to keep the wires as flexible as possible during Z-R. Efficient wire-length minimization is another area which needs further study [13, 80].

## 3.4.7. Use of Simulated Annealing Technique

By implementing powerful placement and wire-optimization algorithms, one may get better result at the cost of longer CPU time per Z-R pass. However, any greedy algorithm which reduces the area monotonically has a possibility that the solution may be stuck at a local optimum. One way to overcome this problem is to use simulated annealing technique during zone-refining. The cost of this approach is long run-time and loss of monotonicity in area reduction.

The simulated annealing technique can easily be accommodated in zone-refining by using a new "accept" function when candidate places are evaluated in the free zone. One good feature of our approach is that we can always keep the layout design-rule-correct without allowing any illegal overlaps. Due to this advantage, the user has complete freedom to choose the number of Z-R passes with simulated annealing. Typically the compaction by Z-R takes about 5 iterations to converge. Combined with simulated annealing, the compaction by Z-R frequently reduces the layout area by 2 - 3 %, after 30 to 50 passes of zone-refining.

However, adding simulated annealing techniques results in the loss of two beneficial properties. The algorithm no longer has polynomial time complexity unless the user limits the maximum number of Z-R passes. Furthermore, the area (or cost) reduction may not be monotone; however, the user may store the best layout found after each pass of Z-R as a backup.

## 3.5. NON-TRIVIAL DESIGN RULES

Some of the design rules may not be easily represented by lower and upper spacing constraints. For example, MOSIS denser contact rule set 'B' involves considerably more constraints than the simpler rule set 'A'. In addition to this, spacing requirement may depend on the width of the wires involved or on their connectivity. Furthermore, there may be rules that require more information than mask name and net-id: Minimum spacing of an active contact from a poly-Si gate can be less than the spacing from a poly-Si wire [88].

All these rules can be accommodated with extra tests and constraints and the introduction of new pseudo-layers. However many more arcs need to be generated between the nodes in the adjacency graph and the constraint generation takes correspondingly longer. While the introduction of these extra rules was somewhat tedious, none of them interfered in any way with the zone-refining algorithm, and we would expect that any constraint graph based compactor would deal with them in a similar manner.

The basic compaction algorithm is independent of technology, and so most of the routines need not be changed when zone-refining is applied to a new technology. Relevant technology information is described in a separate file and thus can be conveniently updated by the user. In particular it specifies:

1 Minimum spacing between any two layers; this provides the lower bounds in the constraint graphs.

2 Resistivity for each layer; this will eventually be used for electrical optimizations of the interconnections.

3 Contacts connecting two different layers; this is used to propagate connectivity information.

## 3.6. HIERARCHICAL COMPACTION

A hierarchical approach is a necessity for VLSI circuit design. This requires corresponding hierarchical computer-aided design and verification tools. Layout compaction is no exception. First, hierarchical compaction can considerably speed up the compaction process. Second, it is normally desirable to maintain the regularity expressed in the iteration of a single cell type throughout a one- or two-dimensional array; similar cells may be maintained as one shape during compaction and finalized after compaction.

The difficulty with hierarchical layout compaction is to coordinate the compaction of adjacent cells, so that they can abut tightly and even share certain features such as common signal lines, contacts and source/drain regions of transistors.

First, we review two approaches for pitch-matching and then describe a new combined method.

## 3.6.1. Hierarchical Compaction by Routing

This method divides the hierarchical compaction problem into two parts, leaf cell compaction and higher level cell compaction, and then solves the two compaction steps separately. Each leaf cell is compacted and represented as an instance in a higher level of hierarchy. The higher level instance can either be abstracted into a polygon for each mask or contain a more detailed description of the leaf cell. After all leaf cells are compacted and represented as instances, the terminals of instances may not match since each leaf cell has been compacted independently. Therefore a routing phase is necessary to make the required connections between instances.

The advantage of this approach is that its algorithm is simple since compaction and interconnection are solved as two independent problems. The disadvantages are that too much area may be used for routing and that the regularity of the chip may get lost. For example, when an array of a subcell is compacted, corresponding interconnections between different instances may have different shapes.

### 3.6.2. Hierarchical Compaction by Abutment

In this approach, the interconnections between subcells are considered during compaction and pitch-matching between terminals is made. This can be done by compacting all the cells into smallest possible area and by expanding compacted cells to match terminal locations. Since the combined problem of compaction and pitch-matching is solved at once, this approach is complicated and may need several compaction passes to obtain interconnection by abutment. In the virtual grid approach, pitch-matching can be done by expanding the grid line spacing of some of the instances [21].

### 3.6.3. Combined Hierarchical Compaction Strategy

As a combination of the above two methods, we have developed an approach in which one may pitch-match terminals by using partly flexible subcells first and then route the remaining connections.

This approach has similarity to the "improved cell model" approach [74] in that a compacted leafcell is partially rigid and partially flexible. One major difference is that we are using routing in addition to making the leafcell partly flexible for pitch-matching and contact/wire sharing.

The zone-refining technique basically offers the flexibility to pitch match cells to a desired dimension. With suitable additional constraints, it may even be possible to migrate shared terminals on the boundary between two cells to positions that are suitable for both partners. However, the implementation of this technique requires to solve some tricky conceptual problems, such as how to represent shared terminals on different leafcells in the database. Thus, for the moment, we have taken a less direct approach.

For the special case of an iterative array of identical cells, we compact the defining cell with suitable cyclic end-around constraints. For example, if a cell is repeated several times in horizontal direction, one can match the terminal positions in vertical direction by adding fixed constraints between terminals of the same height when compacting the leafcell: This will guarantee that the compacted cells can be abutted tightly.

The more general case where two cells of different type abut is handled in several steps.

(A) First we compact the core of each leaf cell, maintaining the terminal frame in unchanged form as given in the symbolic representation that specifies the topology of the layout. A core is a set of components that are not placed on the boundary of an input layout; a core does not contain any shared terminals. The compacted core remains attached to this shared terminal frame by suitable wires (Fig. 3.15).

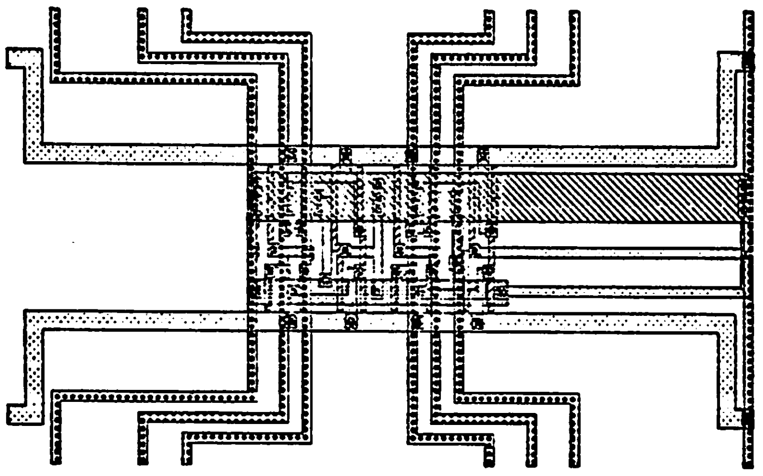


**Figure 3.15.** *A single leafcell with compacted core.*

(B) These core cells with their surrounding frames are then assembled into the desired configuration. A clean-up routine straightens out the wires as much as possible and removes unnecessary wire segments and terminal points (Fig. 3.16).

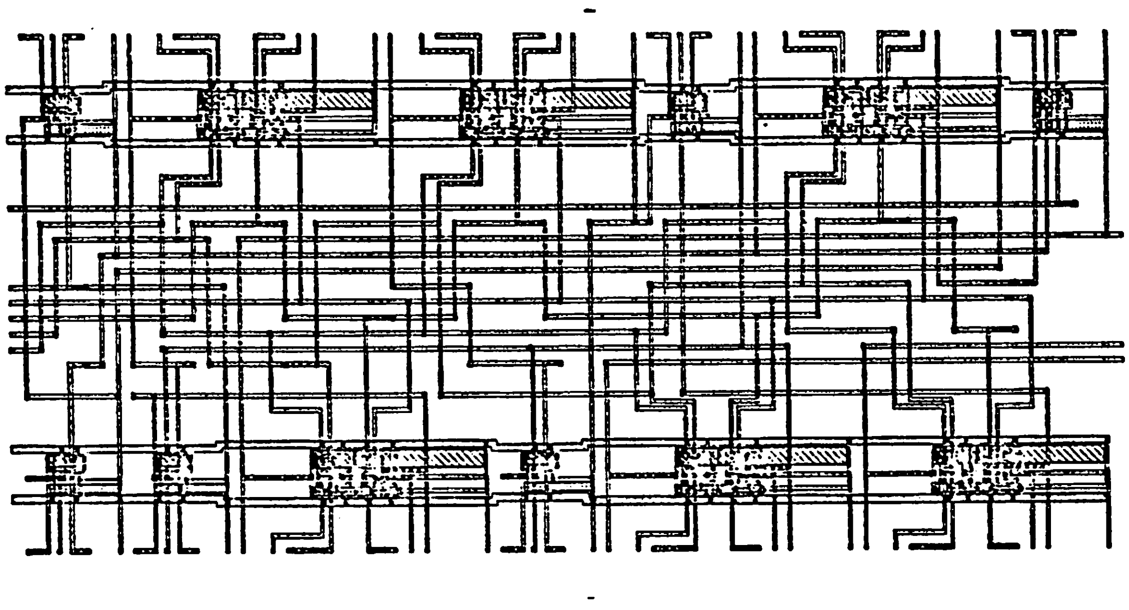


**Figure 3.16.** *Assembly of compacted leafcells.*

(C) Finally, we apply a second compaction step at the next higher level of the hierarchy, in which the cores of the cells are considered fixed clusters. This compacts the terminal frames, the attached wires, and possible routing channels for all participating cells at this level of the hierarchy (Fig. 3.17).

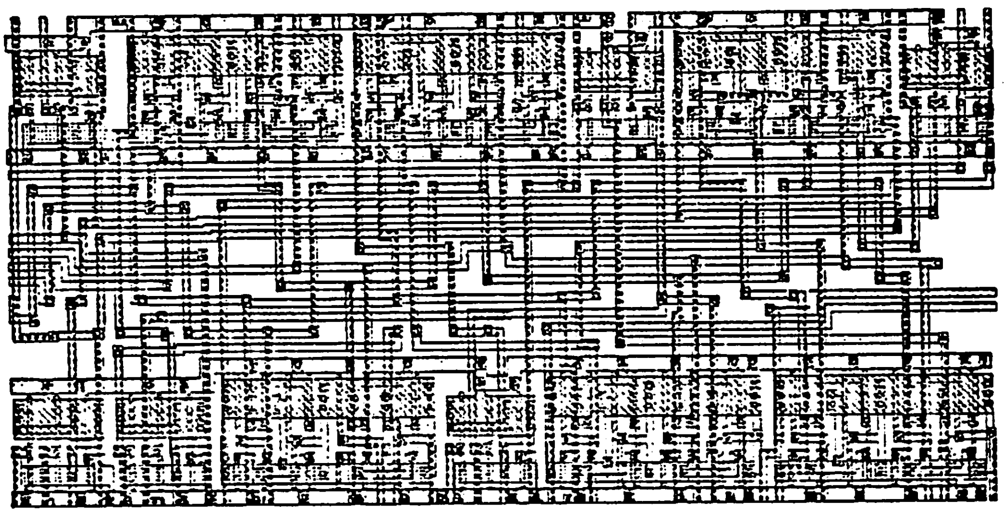


Figure 3.17. *Completed hierarchical compaction of cell assembly.*

This basic strategy, while not as elegant as direct pitch matching of all terminals between cells during the compaction process, allows us to handle all cases in the benchmarks. The strategic control of the necessary sequence of compaction steps is currently under operator control. For the case of a two-dimensional array of identical cells with some one-dimensional arrays of peripheral cells, we first compact the cell of the two-dimensional array. Then the peripheral cells are compacted to the respective width or height of the compacted core of this array cell. After execution of the clean-up routine (B), there is a good probability that the remaining cell cores abut gracefully.

The example used in Figures 3.15, 3.16, and 3.17 is circuit 'ccells132_2' of the benchmarks used at an international workshop on symbolic layout and compaction

[88, 89].

## 3.7. VARIATIONS

Since compaction by Z-R offers various flexibilities in its implementation, there are many possibilities to extend our present implementation, for example, by changing the strategies of selecting clusters of components to be moved, of limiting the range of search, and of placing the components.

### 3.7.1. Back-Tracking

The approach for cluster placement described in Section 3.4.4 is "greedy" in the sense that once an "optimal" position has been found for a cluster, the components of this cluster remain fixed unless they become part of an $x$-cluster.

Better results could be obtained if the positions of the components in the floor could still be changed to make room for cluster parts that are moved across the zone subsequently. Of course, such back-tracking involves a much longer running time since in the limit, all the components can be moved at all times.

### 3.7.2. Inter-pass Processing

Additional freedom in the use of zone-refining results from the possibility to do some processing between subsequent Z-R passes. One option is the possibility to exploit knowledge about the critical path (see next subsection). The critical path can be determined after each Z-R pass and can then be fragmented using special routines. Currently Zorro has an optional routine that preferentially moves all components in the direction away from the critical path; this opens up more space and adds flexibility in the most "congested" area.

Another possibility for inter-pass processing is to use dynamic strategy for jog generation. Since jog generation is very expensive in CPU time, we want to limit its application to cases when we can get good results. A good heuristic on this problem is to use Z-R compaction without jog-generation until the area réduction per pass falls below a given limit, i.e., a few percent of the layout area, and use Z-R compaction with automatic jog-generation later on.

## 3.7.3. Zone-Refining with Critical-Path Shearing

A critical path is a bottleneck which defines the width or height of a layout. W. Wolf of Bell Laboratories suggested a zone-refining method similar to the critical path shearing approach in which each critical path is found and broken if possible to reduce the area of the layout. This speeds up a zone-refining pass since the sophisticated placement routine is applied only to the components in the critical path and non-critical components are placed using a simple straight-downward move. However, the number of passes to obtain a converged solution is larger since fewer critical paths are broken at each Z-R pass.

Table 3.3 shows the compaction result of five benchmark examples [88] using (a) plain zone-refining approach (the method described in Section 3.4) and (b) zone-refining for components on the critical path (the method described in this section). The same set of design rules [88] has been used for all the examples and the results are obtained running Zorro on a DEC MV 8650 under the Ultrix V2.0 operating system.

Both approaches are implemented using the same modular routines. After optimization, more speed-up is expected in approach (b) since most of the routines are written for plain zone-refining, i.e. approach (a). Automatic jog-generation is used; for horizontal wires from the second Z-R pass and for vertical wires from the fourth pass.

The performance of the two approaches is quite similar. Among the five examples, approach (a) resulted in smaller layouts in two examples, approach (b) resulted in smaller layouts in other two examples, and there was a tie in one case. Approach (a) converges faster if the number of zone-refining passes is used as a measure of the speed of the convergence. When critical paths really restrict the compaction, approach (b) wins since it concentrates on the components on the critical path. Otherwise, if several non-critical components should be placed in a compact way to make room for a critical component, then approach (a) wins, since it tries to pack every component as densely as possible.

| | plain zone-refining | | | Z-R on critical path | | |
|---|---|---|---|---|---|---|
| Example | #Z-R pass | area | CPU | #Z-R pass | area | CPU |
| afa | 6 | 10092 | 613 | 9 | 10092 | 744 |
| afakr | 5 | 8181 | 521 | 8 | 8383 | 754 |
| fal3 | 10 | 12084 | 1293 | 11 | 11648 | 1199 |
| aahal | 5 | 8700 | 153 | 6 | 7900 | 182 |
| n28 | 6 | 8750 | 482 | 15 | 8804 | 943 |

(CPU time in seconds and area in lambda square)

Table 3.3 : Comparison of two zone-refining approaches

## 3.8. APPLICATION AND RESULTS

First, we show results of circuit layout compaction, using several different options of zone-refining. Then we present results of the benchmark examples in the compaction session of the international conference on computer design [5].

All examples given in this section have been run on a DEC MV8650 under the Ultrix operating system.

### 3.8.1. Layout Compaction by Zone-Refining

Fig. 3.18 presents the process of zone-refining for an actual circuit. Fig. 3.18(a) shows the configuration after one-dimensional compaction steps in the $x$ and $y$ direction. The bottom contour of this pre-compacted layout becomes the initial ceiling. Fig. 3.18(b) shows an intermediate result after compacting 30 clusters in the down-ward direction; the first Z-R pass finishes after the movement of all 54 clusters. After 5 passes of Z-R, we obtain a converged result shown in Fig. 3.18(c). The area reductions and run-times for this example are summarized in Table 3.4(a); in the table, the width of the layout is reduced by 1-D compaction in the $x$-direction between Z-R passes in the $y$-direction. By using simulated annealing techniques during the cluster placement, we obtain a slightly better solution at the expense of significantly longer CPU times as shown in Fig. 3.18(d) and Table 3.4(b).

For layouts of library cells for a standard-cell environment, it is important to compact the cells to a specified height. Fig. 3.19 demonstrates the pitch-matching capabilities of zone-refining. In Fig. 3.19 (b) and (c), the width of the layout is constrained to a width of 85 units. The precompacted layout (Fig. 3.19 (a)) has been zone-refined into a frame of this width. Monotone greedy Z-R generated the result shown in Fig. 3.19 (b), and Z-R with simulated annealing produced the layout in Fig. 3.19 (c). The compaction results and run-times are listed in Table 3.4 (c). Because of the hill climbing feature of simulated annealing, more changes in relative positions of components can be made with simulated annealing than with a greedy approach.

In Table 3.4, "1-D" indicates compaction in the $x$ and then $y$ direction by a one-dimensional compaction algorithm. "Y+" ("Y-") indicates compaction in the positive (negative) $y$- direction by zone-refining. "Yj" indicates compaction by Z-R in the $y$-direction with automatic jog-introduction.

| (a) Result with 5 passes of Z-R | | | | | |
|---|---|---|---|---|---|
| compaction sequence | width | height | area | (normalized) | CPU time (sec) |
| 1-D (precompaction) | 74 | 147 | 10678 | (100.0 %) | 8 |
| 1-D,Y- | 74 | 140 | 10360 | ( 95.2 %) | 15 |
| 1-D,Y-,Yj+ | 74 | 131 | 9694 | ( 89.1 %) | 28 |
| 1-D,Y-,Yj+,Yj- | 74 | 130 | 9620 | ( 88.4 %) | 46 |
| 1-D,Y-,Yj+,Yj-,Yj+ | 73 | 128 | 9344 | ( 85.9 %) | 82 |
| 1-D,Y-,Yj+,Yj-,Yj+,Yj- | 70 | 127 | 8890 | ( 81.7 %) | 130 |

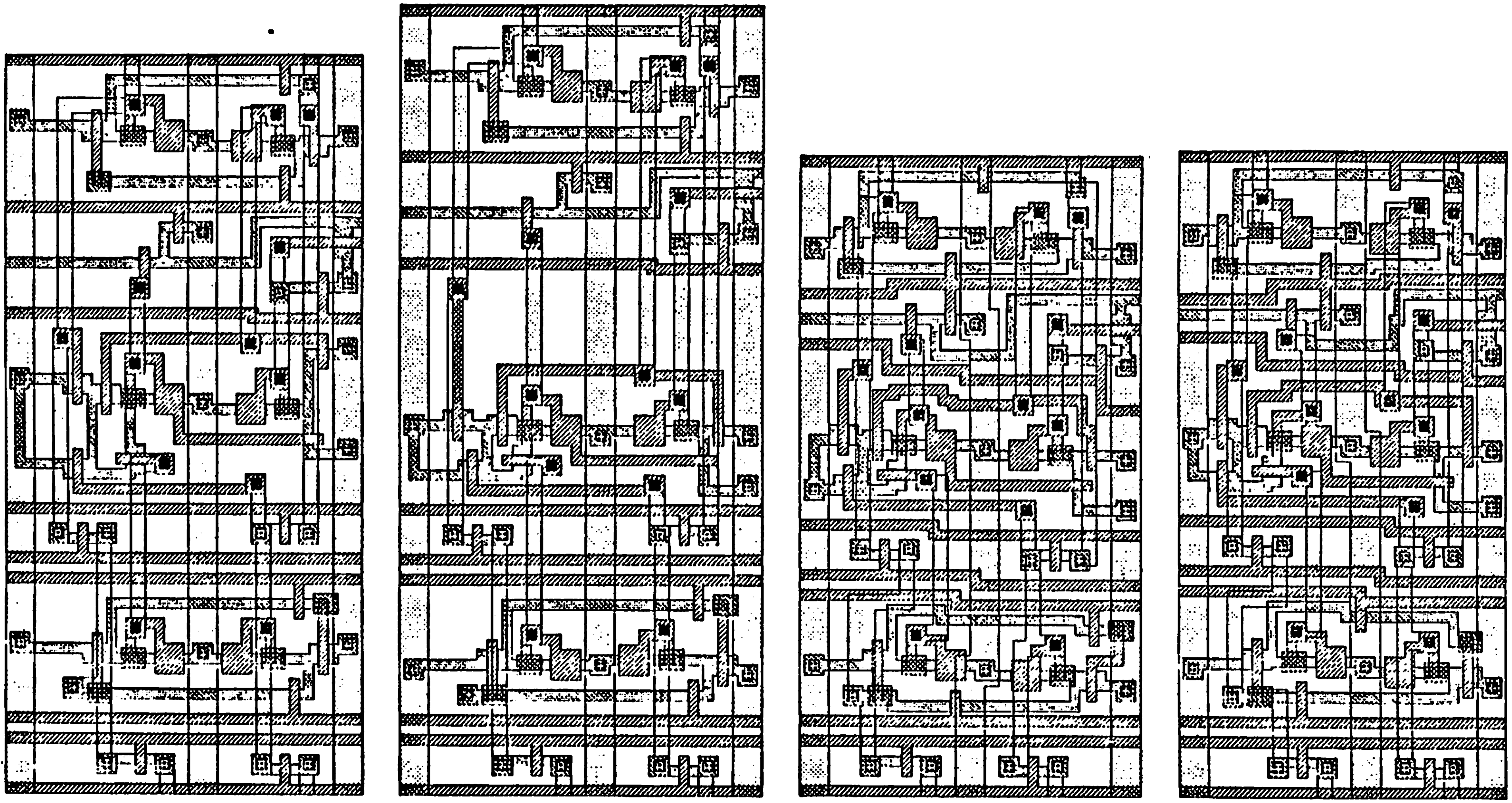
| (b) Result of Z-R with simulated annealing | | | | | |
|---|---|---|---|---|---|
| compaction sequence | width | height | area | (normalized) | CPU time (sec) |
| 30 Z-R passes with SA* (Y-, X+, Y+, X-, ...) | 67 | 128 | 8576 | ( 78.8 %) | 868 |

| (c) Result of Z-R with fixed width constraint | | | | | |
|---|---|---|---|---|---|
| compaction sequence | width | height | area | (normalized) | CPU time (sec) |
| 1-D (precompaction) | 85 | 147 | 12495 | (100.0 %) | 7 |
| 10 Z-R passes | 85 | 119 | 10115 | ( 81.0 %) | 424 |
| 40 Z-R passes with SA* | 85 | 117 | 9945 | ( 79.6 %) | 2046 |

(SA* : simulated annealing)

Table 3.4 : Results of the layout compaction

For the multiple Z-R passes reported in Table 3.4 (c), all passes were executed in alternating (Y-, Y+) directions.



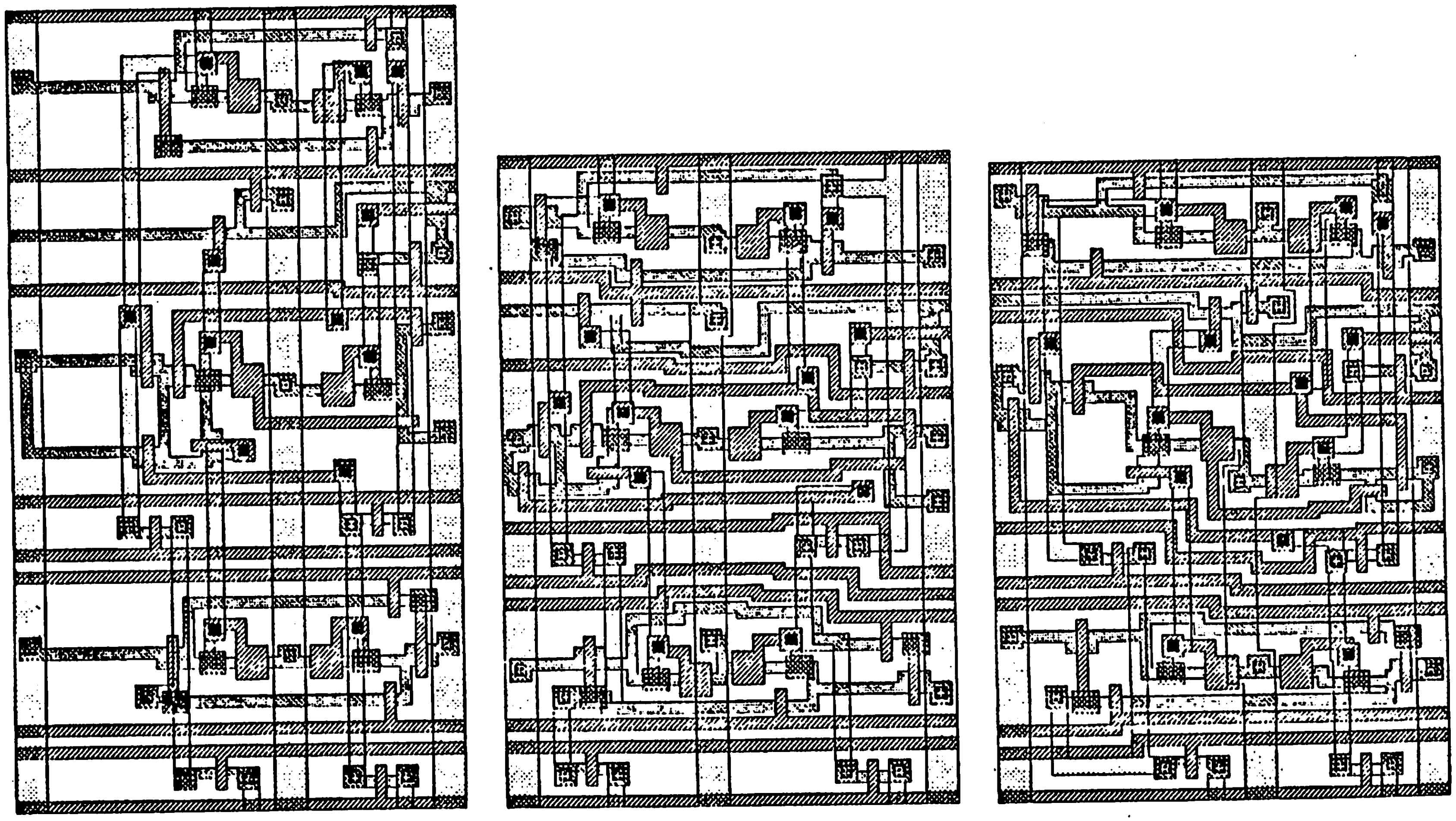
*(a) Precompacted layout using 1-D algorithm in x and then y direction.*

*(b) Intermediate result. 30 clusters have been placed among 54 clusters in the first pass.*

*(c) Result after 5 passes of Z-R. Jogs are generated from the 2nd pass.*

*(d) Result after 30 passes of Z-R with simulated annealing.*

**Figure 3.18.** *Results of layout compaction using zone-refining.*



(a) Precompacted layout with fixed width (width = 85).

(b) Result after 10 passes of Z-R.

(c) Result after 40 passes of Z-R with simulated annealing.

**Figure 3.19.** *Results of zone-refining with a fixed-width constraint.*

## 3.8.2. Results on Benchmark Examples

In this section, results of the benchmark examples used in the 1987 International Conference on Computer Design (ICCD) [5], are presented and they are compared with those of other well-known compactors presented at the conference. The benchmark session is an out-growth of the International Workshop on Symbolic Layout and Compaction (IWSLC). Some of the benchmarks were collected from the workshop and several compactors were invited to the ICCD. Four compactors participated in the session: the MACS [15] constraint graph-based compactor with jog insertion and wire-length minimization, the graph-based compactor SPARCS [8] with analog design support, the Symbolics virtual grid compactor [99] with contact offsetting, and Zorro [88]. All the other compactors except Zorro decomposed the compaction problem into $x$ and $y$ one-dimensional problems.

We compacted all the leaf cells with the same basic strategy: Three or five passes of zone-refining are applied after a one-dimensional precompaction step depending on the number of components of the leafcell. Jog-generation is performed in all but the first pass of zone-refining.

Table 3.5 gives a summary of the results. The CPU times quoted refer to total run-time in seconds including reading the input file and the technology file, writing the output file, connectivity (net list) generation, and all compaction steps. For CMOS cells the area quoted (in $\lambda^2$) includes the PWELL area even though the WELL area is not shown in the layout. For hierarchical examples, the number of zone-refining passes performed at the leafcell level and at the higher level are shown in that order.

| Example | # Z-R pass | area | width | height | CPU | memory |
|---|---|---|---|---|---|---|
| N28 | 5 | 9072 | 72 | 126 | 378 | 704k |
| c132 | 3,0 | 93525 | 435 | 215 | 301 | 1413k |
| c132 | 3,2 | 91258 | 432.5 | 211 | 4821 | 2478k |
| afa | 5 | 10716 | 94 | 114 | 430 | 647k |
| afakr | 5 | 8989 | 89 | 101 | 524 | 598k |
| mul2x2 | 5,0 | 35448 | 211 | 168 | 838 | 614k |
| mul2x2 | 5,2 | 34815 | 211 | 165 | 1062 | 614k |
| mul4x4 | 5,0 | 146304 | 381 | 384 | 1904 | 741k |
| mul4x4 | 5,2 | 142875 | 381 | 375 | 7364 | 741k |
| mul8x8 | 5,0 | 607020 | 755 | 804 | 11738 | 7754k |
| mul2x2.flat | 5 | 35690 | 215 | 166 | 748 | 1470k |
| mul4x4.flat | 3 | 145125 | 387 | 375 | 3388 | 10240k |
| c132.flat | 3 | 90497 | 433 | 209 | 2378 | 5472k |

Table 3.5 : Compaction of benchmark examples

In Table 3.5, we obtained smaller compacted cells (mul2x2 and mul4x4) by using our hierarchical approach than by compacting a hierarchically flat description. We believe that the difference in aspect ratio of the resulting layouts is probably responsible for this. In general, we would expect that a powerful hierarchical compactor would have advantages over flattened-view compaction in CPU time and memory usage.

Fig 3.20 shows the compacted results of mul4x4. Fig 3.20(a) shows the hierarchically compacted result; five zone-refining passes have been applied to compact the leafcells (afa, aahal, fal3), and two zone-refining passes have been applied for higher level compaction. Fig. 3.20(b) shows the result of flattened-view compaction.

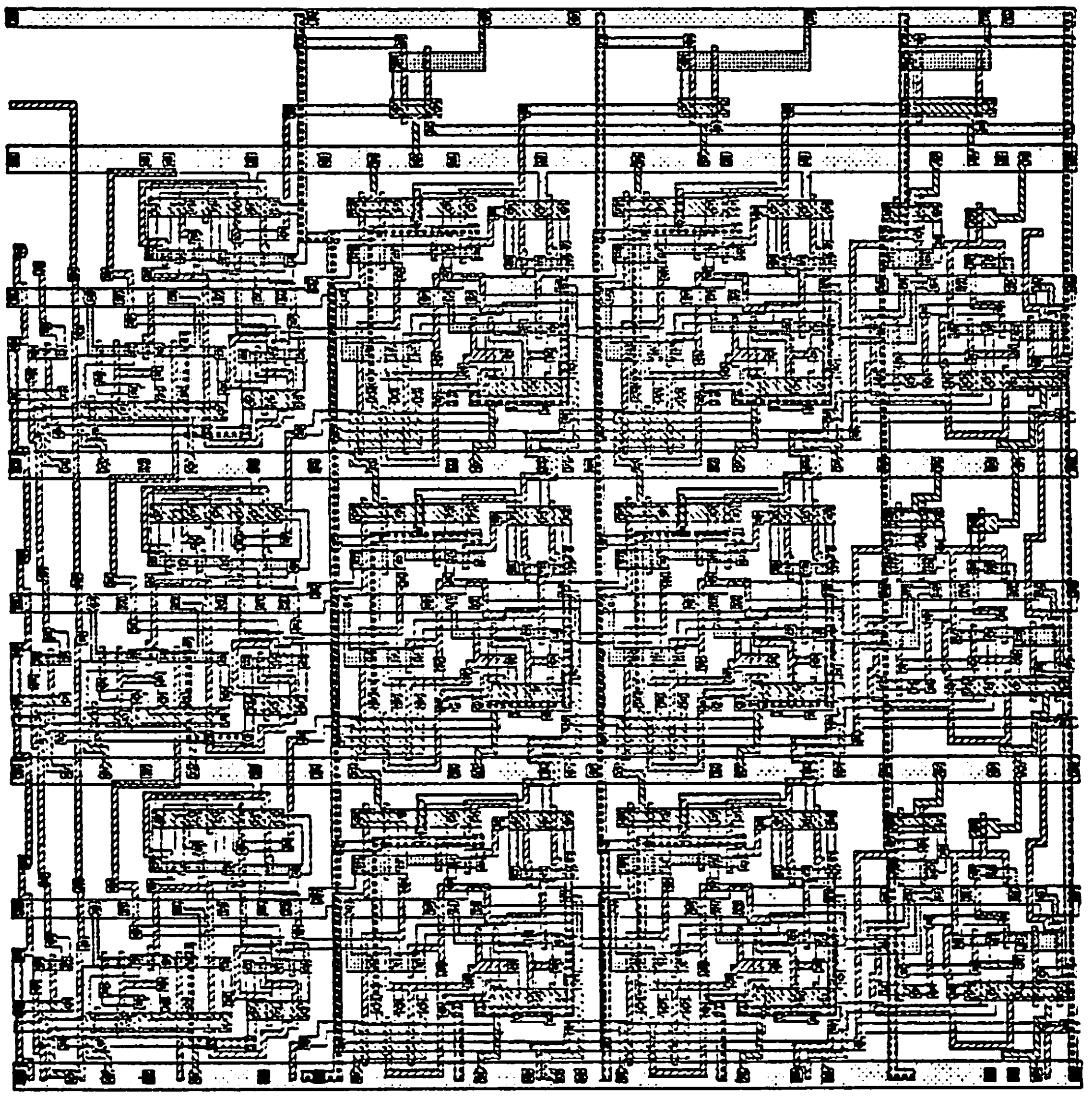


**Figure 3.20.** *Compacted result of mul4x4.*
*(a) hierarchically compacted result.*

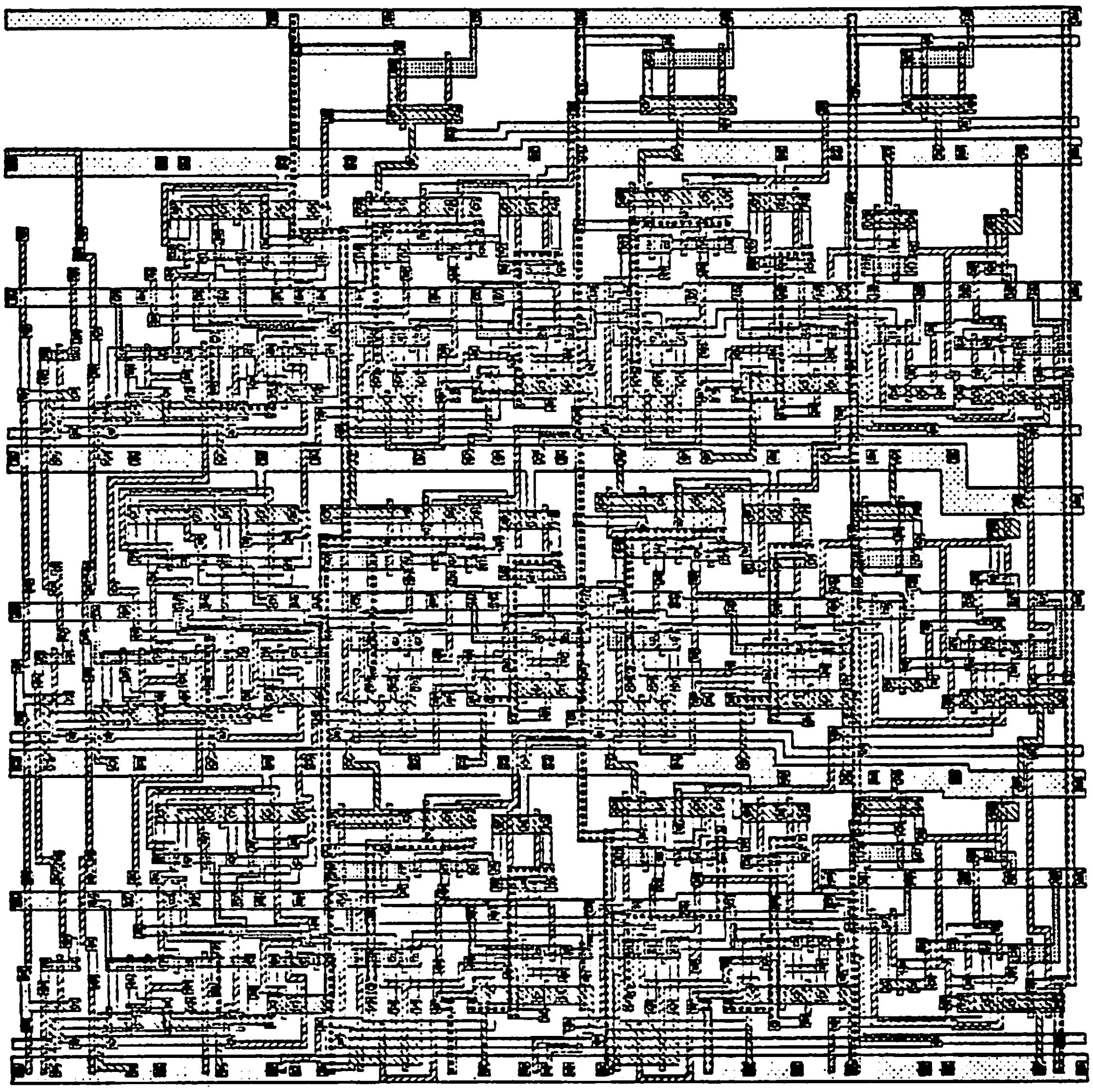


Figure 3.20. *Compacted result of mul4x4.*
*(b) flattened compaction result.*

Table 3.6 shows comparisons of Zorro with other well-known compactors on the benchmark examples. Zorro could generate the best results on most of the examples. No compactor was able to run all 8 examples because of either excessive run-times or other reasons. Zorro produced the smallest area on 5 examples among the 7 examples it participated. On the other two examples, MACS [15] gave the smallest area; however, for these examples, the area differences between results of Zorro and those of MACS are very small (1 - 2 %). The reasons that MACS could gave slightly better results on the two examples are partly because Zorro did not care about pwell area during compaction and added the well area after compaction and partly because MACS has a wire-length minimization routine. On the other examples, Zorro generated up to 17% better results than the next best ones. In memory space requirement, Zorro is comparable with other compactors. Zorro used the largest memory space in two examples among 7, while MACS used largest memory space in all four examples it participated. For hierarchical examples, Zorro kept all the detailed geometries of their subcells even during higher level compaction; mul8x8 has 64 subcells and all the details of them are maintained. By changing the data structure in Zorro to store only abstracted geometries for compacted leafcells, the required memory space and CPU time of Zorro will be substantially reduced.

| Example | Area | CPU | Memory |
|---|---|---|---|
| compactor | (microns) | (sec) | |
| afa | | | |
| MACS | 143 x 166 = 23738 | 9 | 1450k |
| SPARCS | 157 x 180 = 28260 | 11 | 356k |
| Symbolics | 160 x 189 = 30240 | 5 | 164k |
| Zorro | 140.5 x 171 = 24025.5 | 430 | 647k |
| afakr | | | |
| MACS | 142 x 145 = 20590 | 5 | 1390k |
| SPARCS | 157 x 151 = 23707 | 8 | 372k |
| Symbolics | 154 x 154 = 23716 | 5 | 160k |
| Zorro | 128.5 x 151 = 19403.5 | 524 | 598k |
| n28 | | | |
| SPARCS | 123 x 198 = 24354 | 8 | 356k |
| Zorro | 108 x 187 = 20196 | 378 | 704k |
| c132 | | | |
| MACS | 627 x 354 = 221958 | 41 | 3000k |
| SPARCS | 685 x 339 = 232215 | 51 | 184k |
| Symbolics | 675 x 330 = 222750 | 57 | 1040k |
| Zorro | 660 x 322 = 212520 | 301 | 1413k |
| mul2x2 | | | |
| MACS | 309 x 252 = 77868 | 16 | 2050k |
| SPARCS | 343 x 255 = 87465 | 47 | 215k |
| Symbolics | 370 x 270 = 99900 | 10 | 512k |
| Zorro | 312 x 252 = 78624 | 838 | 614k |
| mul4x4 | | | |
| SPARCS | 649 x 601 = 390049 | 66 | 754k |
| Symbolics | 654 x 638 = 417252 | 54 | 840k |
| Zorro | 577 x 577.5 = 333217.5 | 1904 | 741k |
| mul8x8 | | | |
| SPARCS | 1285 x 1285 = 1651225 | 89 | 2066k |
| Symbolics | 1276 x 1352 = 1725152 | 245 | 3200k |
| Zorro | 1138 x 1207.5 = 1374135 | 11738 | 7754k |
| mul16x16 | | | |
| Symbolics | 2524 x 2780 = 7016720 | 1073 | 14400k |

Notes

1 The Symbolics compactor runs on a Symbolics work station. The run times were divided by 4 to scale them to a DEC VAX 8650.

2 Zorro did not generate the pwell. Proper well area has been added.

3 Netlists of SPARCS result of mul8x8 did not match with others.

4 No two netlists for the c132 example were the same.

5 The area, design rules, and netlists are checked by D. Boyer [5]

Table 3.6 : Comparisons on benchmark examples

In general, the speed of convergence and the final result depend on the sequence of the compaction steps. Compaction by zone-refining is robust in that it is less dependent on the initial topology of a layout or compaction sequences when compared with one-dimensional algorithms. The optimal sequence for each case depends on the current and desired aspect ratio and the distribution of the components in the layout.

## 3.9. SUMMARY

Two-dimensional compaction by zone-refining is an effective option in the wide range between simple one-dimensional compaction by constraint-graph based methods and simulated annealing techniques for general placement. This intermediate option is expected to produce denser layouts than one-dimensional compactors because of the additional degrees of freedom in moving individual components, but to run considerably faster than placement by simulated annealing. The greedy, monotone version of zone-refining uses a much more limited way to search for a final solution than simulated annealing techniques. It can thus not be expected to find a solution guaranteed to be close to the global optimum in density if achieving the latter would require dramatic changes in topology. In more sophisticated non-monotone versions, additional CPU time can be traded for slightly improved solutions.

Zone-refining is a novel compaction method, that, while having a preferred direction of compaction at every stage, still moves components in the direction of both coordinate axes to obtain densest local packing. Our first implementation is a greedy algorithm that makes use of any local optimization encountered. The decrease in area is thus strictly monotonic. This method promises to be a good compromise to achieve layouts of a density good enough for practical applications at a reasonable amount of

CPU time.

The compaction results depend on the topology of the circuit since the allowed rearrangement of the components is severely restricted by the original layout. This restriction can be reduced by using more elaborate search algorithms for the placement of circuit clusters at the lower end of the zone or simulated annealing techniques.

Further improvements are expected in Zorro, especially in running time, since the detailed algorithm and the routines have not been completely optimized yet. Our main emphasis was to explore the general concept of compacting a layout by repeated local refinement, to study the different operational modules required for such a compactor, and to discover good data structures for its implementation.

The zone-refining approach presents a general framework for compaction. A wide range of trade-offs between the density of the final solution and the required CPU time can be achieved depending on the sophistication of the algorithms that select, move, and place components in the free zone. One limiting case when all lateral movements are suppressed is a simple one-dimensional compactor. In the other extreme when extensive searching with the use of Monte Carlo hill climbing is employed, the method approaches placement by simulated annealing. This variety could be made available to the user through the choice of a few parameters.

4.

# Chapter 4

# Conclusions and Future Work

In this chapter, areas for future work are described. Also conclusions of this dissertation are presented.

## 4.1. EXTENSIONS AND FUTURE WORK

The physical design problem has been divided into several steps such as floor-planning, placement, global and detailed routing, and compaction, because of the complexity of the overall problem. It is easier to deal with these sub-problems independently. However, these steps are closely coupled and can affect one another. Therefore, to produce a good result that satisfies various goals (which are frequently contradicting), each step should be solved while considering the effects of this step on other routines. An alternative is to use feedback loops and recursive refinements. For example, the placement can be modified according to the result of routing and the routing may be modified to help the compaction step. To guarantee a smooth flow of data between different design steps, all the layout synthesis tools should be able to communicate either by using a centralized database [7] or by defining data formats between all necessary pairs of design tools. Further developments are expected in integrating the CAD tools.

### 4.1.1. Extensions to Mighty

Several improvements can be made to the detailed routing algorithm described in Chapter 2.

#### 4.1.1.1. Multi-layer Routing

Recent advances in process technology allow more than two layers of routing. Hence, to take full advantage of this advances, multi-layer routing becomes an important area of research. The speed of many chips today is limited more by the delay along the connections than by the speed of the basic devices, such as the transistor. This is especially so for LSI (large scale integration) layouts, where up to 80 % of the layout area is used for communication between the modules [92]. Using additional interconnection layers, wires can be shorter and the size of the chip can be reduced [57].

The basic idea of weak and strong modification can also be used for multi-layer routing. However the following generalizations of the two-layer routing scheme have to be made.

1 The pitch, i.e. the center-to-center spacing between two neighboring wire-segments may have different values on different layers. To accommodate this feature in a grid based approach, one may need a rather fine grid which implies longer CPU-times. Therefore, a gridless approach has advantages when the pitches and/or wire-widths vary.

2 When the routing area is large, giving global information to the maze path-finder by generating pseudo-pins is helpful or even necessary. For two-layer routing, VCV (vertical constraint violation) and HCV (horizontal constraint violation) could easily be checked. However, VCV and HCV are not clearly defined when

there are many layers. This problem can be alleviated by partitioning all the nets and assigning set of layers to each subgroup of nets [6]. But this partitioning imposes extra constraints on the use of the various layers and may affect the routing results. We expect that weak and strong modification resolves part of the difficulties.

A prototype of a multi-layer router, ANGELAR (A N-layer GEneral LAyout Router) is under development. So far, it can handle small polygonal routing problems with N-layers of interconnection, where N is a user defined variable. There may be obstacles of any size and shape in the routing region. The basic algorithm of Angelar is similar to that of Mighty, and it seems that strategies used in Mighty will readily be applicable to N-layer routing problems. However, in order to handle various wire-widths and arbitrary pitch values on different layers, we have switched to a gridless data structure similar to the one described by Suzuki, et al. [96]. Fig. 4.1 shows a three-layer routing example routed by Angelar. This is the same example used in Fig. 2.5 and three tracks are used to complete the routing; Mighty used four tracks using two layers.

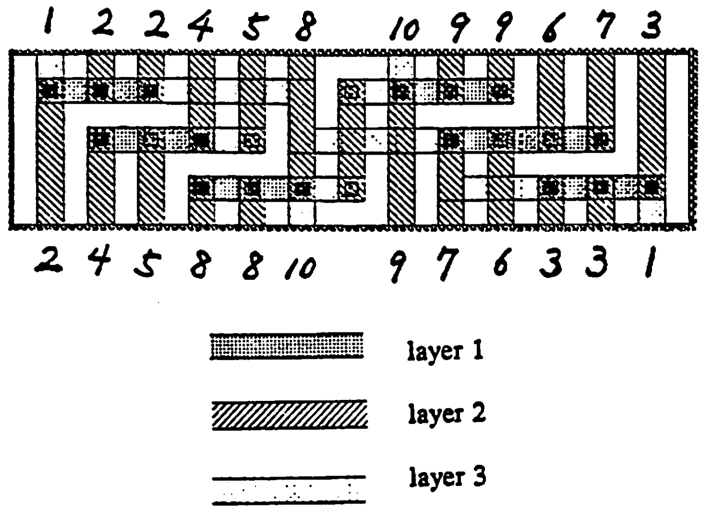


Figure 4.1. *An example of three layer routing by Angelar.*

### 4.1.1.2. Other Extensions to Mighty

When routing analog circuits, there may be some nets which are very sensitive to noise. In this case, the router can protect sensitive nets by routing them first and by penalizing other nets from passing close to the sensitive nets by assigning large cost parameters near the paths of the sensitive nets.

Due to processing technology, we may want to avoid making vias on some particular area in the routing region even though wiring on the area is allowed on both layers. For example, wires on *metal* 1 and *metal* 2 can pass over a transistor active area while we may not want to make a via on top of the active area. Hence, adding the feature to prohibit making vias on some area will allow the router to handle more general cases.

When adjustment of placement is allowed and when complete routing is not possible within the given routing area, the detailed router should be able to suggest the location where it needs more space to complete all the necessary connections. Testing routability is also important [92]. However, I am not aware of any powerful "exact" routability checker [22, 44, 71].

## 4.1.2. Extensions to Zorro

Most of the desired extensions concern the hierarchical part of compaction. First, the data structure to represent a compacted subcell should be reconsidered. In the current prototype implementation of the hierarchical part of Zorro, each subcell contains all the detailed geometries of its components. This increases the required memory space and CPU time. Developing a data structure to maintain abstracted but still partly flexible cells can substantially reduce the memory space and CPU time.

Furthermore, the compactor can be generalized to compact with non-rectangular bounding polygon. With this feature, when one subcell has been compacted, neighboring cells can be compacted to fit on their boundaries to the ones already compacted.

From our experience, we feel that the total compaction time of a chip can be reduced by using a spectrum of different specialized methods to compact layouts of different structure. For example, an array of a single leafcell can be compacted a lot more efficiently than a layout with many different leafcells. For routing area compaction, specialized compactors have been developed already [49, 109]. Recently, some developments are reported [49] on whole-chip compaction methods. Still we need to improve various strategies to handle different special cases effectively.

Wire-length optimization, including wire-length minimization [59, 80, 100], resistance minimization, slack space allocation, is also an important area where efficient algorithm is needed. Wire-length optimization improves electrical properties of the designed circuit and helps compaction by reducing the area occupied by wires.

Another beneficial feature is the ability to modify the topology of the given layout. A good floor planner or topology generator is important to obtain satisfactory results. However, a human designer can often make a few changes to the compacted layout to get a better result. To build a layout synthesis system of the next generation, compactors should be able to make these changes automatically. An alternative approach is to solve several steps of the problem at once since each of the steps is highly dependent on other steps. For example, C.P. Hsu, et al. [34] developed a system which considers the problem of defining the layout hierarchy, area estimation, and aspect ratio assignment simultaneously. Further developments in this direction are expected.

The basic algorithm of zone-refining can also be used for non-Manhattan design style. In this case, the algorithm will decide the slope of wires in the zone so that the minimum zone-gap is maximized, i.e. each connection should be re-implemented by a combination of horizontal, vertical, and 45 degree wires.

## 4.2. CONCLUSIONS

A set of new efficient algorithms for layout compaction and routing have been developed. Since finding the globally optimum solution in either a routing or a compaction problem is NP-hard, exhaustive search can not usually be applied. Instead, new heuristic methods that can iteratively improve the solution have been developed.

The experimental results show that these incremental techniques produce as good as or better results when compared with other pulished methods.

# Bibliography

## References

1. S. B. Akers . J. M. Geyer, and D. L. Roberts, "IC Mask Layout with a Single Conducting Layer," *Proceedings, the 7th Design Automation Workshop,* , pp. 7 - 16 (June 1970).

2. P. Agrawal and M. A. Breuer, "Some Theoretical Aspects of Algorithmic Routing," *Proceedings, 14th Design Automation Conference*, (1977).

3. B. Baker and R. Pinter, "An Algorithm for the Optimal Placement and Routing of a Circuit within a Ring of Pads," *24th Annual Symposium on Foundations of Computer Science*, pp. 360 - 370 (1983).

4. M. W. Bales, "Layout Rule Spacing of Symbolic Integrated Circuit Artwork," *U.C.Berkeley, UCB/ERL Report M82/72*, (1982).

5. D. Boyer, "Symbolic Layout Compaction Benchmarks," *International Conference on Computer Design*, (1987).

6. D. Braun, J. Burns, S. Devadas, H. Ma, K. Mayaram, F. Romeo, and A. Sangiovanni-Vincentelli, "Chameleon: A New Multi-Layer Channel Router," *Proceedings, 23rd Design Automation Conference*, (June 1986).

7. J. Burns, A. Casotto, M. Igusa, F. Marron, F. Romeo, A. Sangiovanni-Vincentelli, C. Sechen, H. Shin, G. Srinath, and H. Yaghutiel, "Mosaico: An Integrated Macro-Cell Layout System," *to be published, VLSI conference*, (August 1987).

8. J. Burns and A. Newton, "Efficient Constraint Generation for Hierarchical Compaction," *International Conference on Computer Design*, (1987).

9. J. Burns and R. Newton, "SPARCS: A New Constraint-Based IC Symbolic Layout Spacer," *Proceedings, IEEE Custom Integrated Circuit Conference*, (1986).

10. M. Burstein and R. Pelavin, "Hierarchical Channel Router," *Proceedings, 20th Design Automation Conference*, pp. 591 - 597 (June 1983).

11. M. Burstein and R. Pelavin, "Hierarchical wire routing," *IEEE Transactions on Computer-Aided Design, Vol. CAD-2, No. 4*, pp. 223 - 234 (1983).

12. C. Carpenter and M. Horowitz, "Generating Incremental VLSI Compaction Spacing Constraints," *Proceedings, 24th Design Automation Conference*, pp. 291 - 297 (1987).

13. Y. E. Cho, "A Subjective Review of Compaction," *Proceedings, 22nd Design Automation Conference*, pp. 396 - 404 (June 1985).

14. S. Chowdhury and M. Breuer, "An O(n) Algorithm for Width Determination of power/ground Routes for VLSI Circuits," *Integration Letter*, pp. 345 - 355 (1986).

15. W. Crocker, R. Varadarajan, and C. Lo, "MACS: a Module Assembly and Compaction System," *International Conference on Computer Design*, (1987).

16. W. A. Dees and P. G. Karger, "Automated Rip-Up and Reroute Techniques," *Proceedings, 19th Design Automation Conference*, pp. 432 - 439 (1982).

17. D. Deutsch, "A Dogleg Channel Router," *Proceedings, 13th Design Automation Conference*, pp. 425 - 433 (June 1976).

18. J. Do and W. Dawson, "SpacerII: A Well-Behaved IC Layout Compactor," *Proceedings, VLSI Conference*, (1985).

19. A. E. Dunlop, "SLIM-The Translation of Symbolic Layouts into Mask Data," *Journal of Digital Systems, Vol V, No 4,*, pp. 429 - 451 (1981).

20. A. E. Dunlop, "SLIP: Symbolic Layout of Integrated Circuits with Compaction," *Computer Aided Design*, pp. 387 - 391 (November 1978).

21. G. Entenman and S. Daniel, "A Fully Automatic Hierarchical Compactor," *Proceedings, 22nd Design Automation Conference*, pp. 69 - 75 (June 1985).

22. A. Frank, P. Levai, J. Mozes, P. Scsaurszki, and E. Tardos, "Sufficient Conditions for Solvability of Channel Routing Problems," *ISCAS*, (1984).

23. S. Goto, T. Matsuda, K. Takamizawa, T. Fujita, H. Mizumura, H. Nakamura, and F. Kitajima, "LAMBDA: An Integrated Master-Slice LSI CAD System," *Integration, the VLSI journal*, pp. 53 - 69 (1983).

24. G. Hamachi, "An Obstacle-Avoiding Router for Custom VLSI," *Ph.D. Thesis*, University of California, Berkeley, CA., (April 1986).

25. G. Hamachi and J. Ousterhout, "A Switch-box Router with Obstacle Avoidance," *Proceedings, 21nd Design Automation Conference*, pp. 173 - 179 (June 1984).

26. N. Hasan and C. Liu, "A Force Directed Global Router," *Proceedings, the 1987 Stanford Conference*, pp. 135 - 150 (1987).

27. T. Hedges, W. Dawson, and Y. E. Cho, "Bitmap Graph Build Algorithm for Compaction," *Proceedings, IEEE International Conference on CAD*, pp. 340 - 342 (November 1985).

28. W. R. Heller, W. F. Mikhail, and W. E. Donath, "Prediction of Wiring Space Requirements for LSI," *Proceedings, 14th Design Automation Conference*, pp. 32 - 42 (1977).

29. D. Hightower, "The Lee Router Revisited," *Proceedings, IEEE International Conference on Computer Design*, pp. 136 - 139 (1983).

30. D. Hightower, "A Solution to Line Routing Problems on the Continuous Plane," *Proceedings, 6th Design Automation Workshop*, (1969).

31. D. D. Hill, J. P. Fishburn, and M. P. Leland, "Effective Use of Virtual Grid Compaction in Macro-Module Generators," *Proceedings, 22nd Design Automation Conference*, pp. 777 - 780 (1985).

32. D. Hodges and H. Jackson, *Analysis and Design of Digital Integrated Circuits*, McGraw Hill (1983).

33. Y. Hsieh and C. Chang, "A Modified Detour Router," *Proceedings, IEEE International Conference on CAD*, pp. 301 - 303 (November 1985).

34. C. Hsu, S. Evans, J. Tang, K. Chow, R. Perry, and J. Liu, "An Efficient Hierarchical Approach to High Complexity Circuit Layout," *Proceedings, IEEE Custom Integrated Circuit Conference*, pp. 614 - 617 (1987).

35. C. Hsu, "A New Two-Dimensional Routing Algorithm," *Proceedings, 19th Design Automation Conference*, pp. 46 - 50 (June 1982).

36. C. Hsu, "Theory and Algorithms for Signal Routing in Integrated Circuit Layout," *Ph.D. Thesis*, University of California, Berkeley, CA., (1983).

37. M. Y. Hsueh, "Symbolic Layout and Compaction," *U.C.Berkeley, UCB/ERL Report M79/80*, (1979).

38. M. Hung and W. O. Rom, "Solving the Assignment Problem by Relaxation," *Operations Research* **28**, No. 4 pp. 969 - 982 (August 1980).

39. M. Ishikawa, T. Matsuda, and S. Goto, "Compaction Based Custom LSI Layout Design Method," *Proceedings, IEEE International Conference on CAD*, pp. 343 - 345 (November 1985).

40. R. Joobbani, "An Application of Knowledge-Based Expert System to Detailed Routing of VLSI Circuits," *Ph.D. Thesis*, Carnegie-Mellon University, (1985).

41. R. Joobbani and D. Siewiorek, "WEAVER: A Knowledge- Based Routing Expert," *Proceedings, 22nd Design Automation Conference*, pp. 266 - 272 (June 1985).

42. R. L. Joseph, "An Expert Systems Approach to Completing Partially Routed Printed Circuit Boards," *Proceedings, 22nd Design Automation Conference*, pp. 523 - 528 (1985).

43. Y. Kajitani, "Order of Channels for Safe Routing and Optimal Compaction of Routing Area," *IEEE Transactions on Computer Aided Design* 2 pp. 293 - 300 (October, 1983).

44. R. Karp, F. Leighton, R. Rivest, C. Thompson, U. Vazirani, and V. Vazirani, "Global Wire Routing in Two-Dimensional Arrays," *24th Annual Symposium on Foundations of Computer Science*, pp. 453 -459 (1983).

45. G. Kedem and H. Watanabe, "Graph-Optimization Techniques for IC Layout and Compaction," *IEEE Transactions on CAD of ICAS Vol. 3, No. 1,* (January 1984).

46. K. H. Keller, "An Electronic Circuit CAD Framework," *U.C.Berkeley, UCB/ERL Report M84/54,* (1984).

47. C. Kingsley, "A Hierarchical, Error-Tolerant Compactor," *Proceedings, 21nd Design Automation Conference*, pp. 126 - 132 (June 1984).

48. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science, Vol. 220*, pp. 671 - 680 (May 1983).

49. R. Kossey, "KCOMP: A Full Chip Compaction Strategy," *Proceedings, IEEE Custom Integrated Circuit Conference*, pp. 610 - 613 (1987).

50. W. Kraft and W. Hein, "A Router for Channels of Nonuniform Width Containing Preplaced Wiring and Obstacles," *Integration, the VLSI journal*, pp. 223 - 244 (1985).

51. U. Lauther, *private communications*, (1986).

52. U. Lauther, "Channel Routing in a General Cell Environment," *Proceedings, VLSI Conference*, pp. 389 - 399 (1985).

53. U. Lauther, "A Data Structure for Gridless Routing," *Proceedings, 17th Design Automation Conference*, pp. 603 -609 (1980).

54. E. Lawler, "Combinatorial Optimization : Networks and Matroids," in *Holt, Rinehart and Winston*, (1976).

55. C. Lee, "An Algorithm for Path Connections and its Applications," *IRE Transactions on Electronic Computers, Vol EC-10*, pp. 346 - 365 (September 1961).

56. J. Lee, "Compaction of VLSI Layouts with General Design Rules," *International Workshop on Symbolic Layout and Compaction, Chapel Hill, North Carolina*, (November, 1986).

57. F. T. Leighton and A. L. Rosenberg, "Automatic Generation of Three-Dimensional Circuit Layouts," *ICCD*, pp. 633 - 636 (1983).

58. Y. Liao and C. K. Wong, "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints," *IEEE Transactions on CAD of ICAS, Vol. 2, No. 2*, pp. 62 - 69 (April 1983).

59. S. Lin and J. Allen, "Minplex - A Compactor That Minimizes the Bounding Rectangle and Individual Rectangles in a Layout," *Proceedings, 23rd Design Automation Conference*, pp. 123 - 130 (1986).

60. R. Linsker, "An Iterative-Improvement Penalty-Function-Driven Wire Routing System," *IBM J. Res. Develop. Vol. 28 No. 5*, pp. 613 - 624 (September 1984).

61. C. Lo, R. Varadarajan, and W. Crocker, "Compaction with Performance Optimization," *IEEE International Symposium on Circuits and Systems*, pp. 514 - 517 (1987).

62. W. Luk, "A Greedy Switch-box Router," *INTEGRATION, the VLSI journal 3*, pp. 129 - 149 (1985).

63. F. M. Maley, "Compaction with Automatic Jog Introduction," *Proceedings, Chapel Hill Conference on VLSI*, pp. 261 - 283 (1985).

64. M. Marek-Sadowska, "Route Planner for Custom Chip Design," *Proceedings, IEEE International Conference on CAD*, pp. 246 - 249 (November 1986).

65. M. Marek-Sadowska, "Two-dimensional Router for Double Layer Layout," *Proceedings, 22nd Design Automation Conference*, pp. 117 - 123 (June 1985).

66. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison Wesley (1980).

67. R. C. Mosteller, A. Frey, and R. Suaya, "2-D Compaction - A Monte Carlo Method," *Proceedings, the 1987 Stanford Conference*, pp. 173 - 197 (1987).

68. A. Moulton, "Laying the Power and Ground Wires on a VLSI Chip," *Proceedings, 20th Design Automation Conference*, pp. 754 - 755 (1983).

69. L. S. Nyland, "Improving Virtual-Grid Compaction Through Grouping," *Proceedings, 24th Design Automation Conference*, pp. 305 - 310 (1987).

70. J. Ousterhout, "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools," *IEEE Transactions on CAD, Vol.CAD-3, No.1*, (January, 1984).

71. A. Patel, C. Yeh, and L. Cote, "Theoretical and Experimental Wirability Analysis System (TEWAS)," *Proceedings, IEEE International Conference on Computer Aided Design*, pp. 69 - 71 (1984).

72. W.H. Pfann, "Principles of Zone-Refining," *Trans. AIME 194, 747*, (1952).

73. J. Reed, A. Sangiovanni-Vincentelli, and M. Santomauro, "A New Symbolic Channel Router : YACR2," *IEEE Transactions on Computer-Aided Design, Vol. CAD-4, No. 3*, pp. 208 - 219. (July 1985).

74. M. Reichelt and W. Wolf, "An Improved Cell Model for Hierarchical Constraint-Graph Compaction," *Proceedings, IEEE International Conference on CAD*, pp. 482 - 485 (1986).

75. R. L. Rivest and Fiduccia, "A Greedy Channel Router," *Proceedings, 19th Design Automation Conference*, pp. 418 - 424 (June 1982).

76. C. Rogers, J. Rosenberg, and S. Daniel, "MCNC's Vertically Integrated Symbolic Design System," *Proceedings, 22nd Design Automation Conference*, pp. 62 - 68 (June 1985).

77. J. B. Rosenberg, "Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries," *IEEE Transactions on CAD* **vol. 4 no. 1** pp. 53 - 67 (January 1985).

78. F. Rubin, "An Iterative Technique for Printed Wire Routing," *Proceedings, 11th Design Automation Workshop*, pp. 308 - 313 (1974).

79. S. Sastry and A. Parker, "The Complexity of Two-Dimensional Compaction of VLSI Layouts," *Proceedings, IEEE International Conference on Circuits and Computers*, pp. 402 -406 (1982).

80. W. L. Schiele, "Improved Compaction by Minimized Length of Wires," *Proceedings, 20nd Design Automation Conference*, pp. 121 - 127 (June 1983).

81. M. Schlag, Y. Z. Liao, and C. K. Wong, "An Algorithm for Optimal Two-Dimensional Compaction of VLSI Layouts," *Integration, the VLSI journal*, pp. 179 - 209 (1983).

82. W. Scott, "Compaction and Circuit Extraction in the MAGIC IC Layout System," *Ph.D. Thesis*, University of California, Berkeley, CA., (1985).

83. C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *Proc. 1984 Custom Integrated Circuit Conf., Rochester, NY*, (May 1984).

84. C. Sequin, "Tools for Macro Module Construction," *International Workshop on Symbolic Layout and Compaction*, (November 1986).

85. H. Shin and A. Sangiovanni-Vincentelli, "A Detailed Router Based on Routing Modifications: Mighty," *To be published, IEEE Transactions for CAD of IC&S.*, (November 1987).

86. H. Shin and A. Sangiovanni-Vincentelli, "MIGHTY: A 'Rip-up and Reroute' Detailed Router," *Proceedings, IEEE International Conference on CAD*, pp. 2 - 5. (November 1986).

87. H. Shin, A. Sangiovanni-Vincentelli, and C. Sequin, "Two-dimensional Compaction by Zone-Refining," *Proceedings, 23nd Design Automation Conference*, pp. 115 - 122 (June 1986).

88. H. Shin, A. Sangiovanni-Vincentelli, and C. Sequin, "Two-dimensional Module Compactor Based on Zone-Refining," *International Conference on Computer Design*, (1987).

89. H. Shin, A. Sangiovanni-Vincentelli, and C. Sequin, "Zorro: Two-Dimensional Compaction By 'Zone Refining'," *Benchmark Session, International Workshop on Symbolic Layout and Compaction, Chapel Hill, North Carolina*, (November 1986).

90. H. Shin, A. Sangiovanni-Vincentelli, and C. Sequin, "'Zone-Refining' Techniques for two-dimensional Layout Compaction," *In preparation*, (1987.).

91. I. Shirakawa and S. Futagami, "A Re-routing Scheme for Single-Layer Printed Wiring Boards," *IEEE Transactions on Computer-Aided Design, Vol. CAD-2, No. 4*, pp. 267 - 271 (October 1983).

92. Soukup, "Circuit Layout," *Proceedings of the IEEE* **vol. 69, no. 10** pp. 1281 - 1304 (October 1981).

93. J. Starzyk, "Decomposition Approach to a VLSI Symbolic Layout with Mixed Constraints," *ISCAS*, pp. 457 - 460 (1984).

94. J. Stenstrom and R. Mattheyses. "Switch-box Routing The Greedy Way." *Proceedings, IEEE International Conference on CAD*. pp. 307 - 309 (1985).

95. K. Suzuki, Y. Matsunaga, M. Tachibana, and T. Ohtsuki. "A Hardware Maze Router with Application to Interactive Rip-up and Reroute Support." *IEEE Transactions on Computer-Aided Design, Vol. CAD-5, No. 4*, pp. 466 - 476. (1986).

96. K. Suzuki, T. Ohtsuki, and M. Sato. "A Gridless Router: Software and Hardware Implementations." *Proceedings, International Conference on VLSI*, (1987).

97. Z. Syed and A. Gamal. "Single Layer Routing of Power and Ground Networks in Integrated Circuits." *Journal of Digital Systems* 6 pp. 53 - 63 (1982).

98. T. G. Szymanski. "Dogleg Channel Routing is NP-Complete." *IEEE Transactions on CAD* vol. 4, no. 1 pp. 31 - 40 (January 1985).

99. D. Tan and N. Weste. "Virtual Grid Symbolic Layout 1987." *International Conference on Computer Design*, (1987).

100.

R. Varadarajan and G. Lakhani. "A Wire-Length Minimization Algorithm for Circuit Layout Compaction." *International Workshop on Symbolic Layout and Compaction, Chapel Hill, North Carolina*, (November, 1986).

101.

M. Vecchi and S. Kirkpatrick. "Global Wiring by Simulated Annealing." *IEEE Transactions on Computer Aided Design Vol. 2, No. 4*, pp. 215-222 (October 1983).

102.

H. Watanabe. "IC Layout Generation and Compaction Using Mathematical Optimization." *Ph.D. Thesis, Dept. of Computer Science, The University of Rochester*, (1984).

103.

J. Weinstein, "A New Two-Dimensional Compaction Algorithm for Symbolic Layout," *Proceedings, IEEE Custom Integrated Circuit Conference,* pp. 605 - 609 (1987).

104.

J. D. Williams, "STICKS - Graphics Editor for High-Level LSI Design," *Proceedings, National Computer Conference,* pp. 289 - 295 (1978).

105.

W. Wolf, "An Experimental Comparison of 1-D Compaction Algorithms," *Proceedings, Chapel Hill Conference on VLSI,* pp. 165 - 179 (1985).

106.

W. Wolf, R. Mathews, J. Newkirk, and R. Dutton, "Two-Dimensional Compaction Strategies," *Proceedings, IEEE International Conference on CAD,* pp. 90 - 91 (1983).

107.

W. Wolf, "Two-Dimensional Compaction Strategies," *Ph.D thesis, Department of Electrical Engineering, Stanford University,* (1984).

108.

X. Xiong and E. Kuh, "The Scan Line Approach to Power and Ground Routing," *Proceedings, IEEE International Conference on CAD,* pp. 6 - 9 (1986).

109.

X. Xiong and E. S. Kuh, "Nutcracker: An Efficient and Intelligent Channel Spacer," *Proceedings, 24th Design Automation Conference,* pp. 298 - 304 (1987).

110.

T. Yoshimura and E. Kuh, "Efficient Algorithms for Channel Routing," *IEEE Transactions on Computer-Aided Design, Vol. CAD-1, No. 1,* pp. 25 - 35 (1982).

NAME
mighty - A Detailed Router Based on Routing Modifications

SYNOPSIS
mighty [options] [file1 [file2]]

DESCRIPTION
Mighty is a two-layer symbolic detailed router for any rectagonal routing area. The pins to be connected can be on the boundaries or inside the rectagon, and the boundaries are not used for connection. Floating pins can be on either leftmost or rightmost edge of the routing area.

File1 is the input, file2 is the output. If file2 is omitted, the output goes to standard output. If no files are specified, input is read from standard input, and output goes to standard output. The formats of input and output files are as follows:

The input file contains a number of key words and data in integers. The key words are

1) number_of_nets : Number of nets to be routed.

2) rectagoncorners : Number of corner points of the rectagonal routing area, followed by coordinates of the corners. The coordinates of rectagon corners are listed in counterclockwise sequence. MIGHTY regards the boundary of the routing area as a directed curve.

3) number_of_pins : Total number of pins to be connected, followed by a list of pins. Each pin is described by net-number, x and y coordinates and a layer-number.

4) left_list : number of nets going out through left edge of the routing area, if any, followed by the list of such nets.

5) right_list : number of nets going out through right edge of the routing area, if any, followed by the list of such nets.

6) obstacles : number of horizontal or vertical lines of obstacles, if any, followed by a list of obstacles. Each obstacle is a line segment with beginning and ending point coordinates, and its layer-number.

7) sensitive_nets : number of critical nets, if any, followed by a list of critical nets. These nets are

routed before other nets with priority.

The origin of the coordinate system of the routing area can be chosen arbitrarily. For the current version of MIGHTY the layer numbers are 1 and 2. Layer 1 is mainly used for vertical connections and layer 2 is mainly used for horizontal connections.

The output file begins with a key word 'channelwiring'. Then the number of vias used in the routing is given, followed by the list of the vias with their net-number, x and y coordinates, and two layer numbers they are interconnecting. After that, the key word 'wires' is followed by the number of wire segments and a list of wires. Each wire is described by a net-number, two pairs of coordinates for the endpoints and a layer number.

The command line options described below can be specified in any order, but must come before the input and output file names.

-s Do not generate output of the routing; only print statistics such as total wire-length, number of vias, etc.

-m Do not perform metal maximization and clean-up.

-e Give the estimate of incremental channel width to complete the routing.

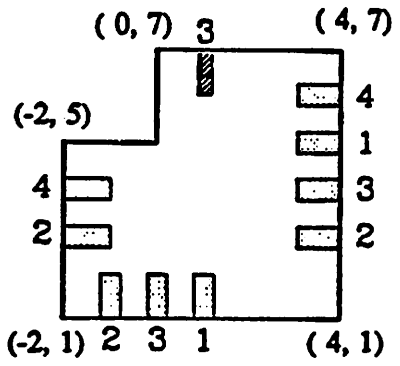
## AUTHOR

Hyunchul Shin

## REFERENCE

H. Shin and A. Sangiovanni-Vincentelli, "MIGHTY: A 'Rip-up and Reroute' Detailed Router", _Proc. IEEE International Conference on Computer Aided Design_, pp 2 - 5, Santa Clara, CA., Nov. 1986.

```
Example of an input file:
number_of_nets 4
rectagoncorners 6
-2 1
4 1
4 7
0 7
0 5
-2 5
number_of_pins 10
1 1 1 2
1 4 5 2
2 -2 3 2
2 4 3 2
2 -1 1 2
3 0 1 2
3 4 4 2
3 1 7 1
4 -2 4 2
4 4 6 2
```



▨ Layer 1

░ Layer 2

```
Example of an output file
channelwiring
vias 4
3 0 2 1 2
1 3 2 1 2
3 2 4 1 2
1 3 5 1 2
wires 24
1 3 5 4 5 2
1 1 1 1 2 2
1 3 2 3 5 1
1 1 2 3 2 2
2 -1 1 -1 2 2
2 3 3 4 3 2
2 -2 3 -1 3 2
2 -1 3 3 3 2
2 -1 2 -1 3 2
3 1 6 1 7 1
3 3 4 4 4 2
3 0 1 0 2 2
3 0 2 0 4 1
3 0 4 2 4 1
3 2 4 3 4 2
3 2 4 2 6 1
3 1 6 2 6 1
4 3 6 4 6 2
4 -2 4 -1 4 2
4 1 4 1 5 2
4 -1 4 1 4 2
4 1 5 2 5 2
4 2 5 2 6 2
4 2 6 3 6 2
```

## NAME

zorro - A Two-Dimensional Compactor Based on Zone-Refining

## SYNOPSIS

zorro [options] [-I file1 [-O file2]]

## DESCRIPTION

<u>Zorro</u>

is a 2-dimensional layout compactor for integrated circuits. After a traditional one-dimensional precompaction step, the size of the layout is further reduced with a technique that bears a strong similarity to the technique of 'zone-refining' used in the purification of crystal ingots. Individual circuit components or small clusters of components are peeled off row by row from the precompacted layout, moved across an open zone, and reassembled at the other end of this zone in a denser configuration. In this process both coordinates of the moved components are altered and jogs are introduced in the connecting wires between them to produce the needed flexibility for placing components into optimal positions. The constraint graphs in both the x- and y-direction are used and updated concurrently. Simulated annealing techniques can also be employed within the zone-refining process.

The formats of input and output files are as follows:

The input file contains a number of key words and data in integers. The key words are

1) I : An instance such as a transistor, a contact, a rigid macro-cell.

2) H : A horizontal wire.

3) V : A vertical wire.

4) P : A formal terminal.

5) BB : Coordinates of the bounding rectangle.

Key word BB is followed by four integers (left, lower, right, top) representing the bounding box of the layout. All the other key words except BB are followed by two integers representing the reference point of the component, the components name (on the same line), and a list of rectangular protection frames on various layers. Each protection frame has four integers showing the bounding box, layer name, and optional net-id. There does not exist any hard limit on the number of protection frames a component can have.

The output file will be in the same format as input if file2 is specified. If file2 is omitted, the output goes to OCT file "SPACED" with physical view.

The command line options described below can be specified in any order.

-I file1
: Set input file name to file1.

-O file2
: Set output file name to file2.

-r file3
: Set rule (technology) file name to file3. By default, the rule file name is ".zorrorule".

-e file4
: Change error file name to file4. By default, error file name is "stderr".

-n # Do compaction by zone-refining # times.

-w # Change desired width of the layout from 0 to #.

-a # Do zone-refining with simulated annealing (hill climbing). Decrease temperature to # % of the previous temperature after each pass of zone-refining.

-s Rotate the layout 90 degrees after each compaction pass. By default, the layout is rotated by 180 degrees.

-y Compact in vertical direction first. By default, the first pre-compaction is in horizontal direction.

-l Do leafcell compaction.

-b Do compaction at higher level in hierarchy.

-h # Do automatic jog-generation for horizontal wires from #-th pass of zone-refining.

-v # Do automatic jog-generation for vertical wires from #-th pass of zone-refining.

-c # Try to push away components from the critical path which defines the maximum width or height of the layout, from #-th pass.

-fh Do not move pins in horizontal direction during compaction.

-fv Do not move pins in vertical direction during compaction.

AUTHOR

Hyunchul Shin

REFERENCE

H. Shin, A. Sangiovanni-Vincentelli and C. Sequin, "Two - Dimensional Compaction By 'Zone Refining'," Proceedings ACM/IEEE 23rd Design Automation Conference, pp 115 - 122, Las Vegas, June 1986.