Relatório Técnico

Núcleo de Computação Eletrônica

A Survey and Taxonomy of Layout Compaction Algorithms

Anido, M. L.
Oliveira, C. E. T.

NCE - 16/00

Universidade Federal do Rio de Janeiro

A Survey and Taxonomy of Layout Compaction Algorithms

Anido, M.L. and Oliveira, C.E.T.
Núcleo de Computação Eletrônica - Universidade Federal do Rio deJaneiro
Email: {mlois,carlo}@nce.ufrj.br

Abstract

This paper presents a survey and a taxonomy of layout compaction algorithms, which are an essential part of modern symbolic layout tools employed in VLSI circuit design. Layout compaction techniques are also used in the low-end stages of silicon compilation tools and module generators. The paper addresses the main algorithms used in compaction, focusing on their implementation characteristics, performance, advantages and drawbacks. Compaction is a highly important operation to optimize the use of silicon area, achieve higher speed through wire length minimization, support technology retargeting and also allow the use of legacy layouts. Optimized cells that were developed for a fabrication process with a set of design rules have to be retargeted for a new and more compact process with a different set of design rules.

Keywords: Compaction Algorithms, Symbolic Layout, EDA tools, CAD tools

1. Introduction

The rapid design of VLSI systems plays an important role in the progress of science and technology. In general, several important problems can be described in VLSI CAD systems. They are typically design specification, functional design, logical design, circuit design, physical design, and fabrication (technology design). To link these problems, we must perform functional and logical simulation, circuit analysis, extraction and verification. Physical design automation of VLSI systems consists of six major problems. They are partitioning, placement, assignment and floorplanning, routing, symbolic layout and compaction, and verification.

Compaction is the translation of a layout designed from a generic technology (such as CMOS) to a particular fabrication technology's design rules. It translates symbolic layout (described in graphical or textual form) or translates the output of the detailed-routing phase into mask data and has to convert the circuit elements into the appropriate mask elements and minimize the chip area. The goal is to minimize the area of the layout, while preserving the design rules and not altering the function of the circuit. This goal gives this process the name compaction. Compaction changes the geometry of the topological design to produce as small a layout as possible while enforcing the design rules.

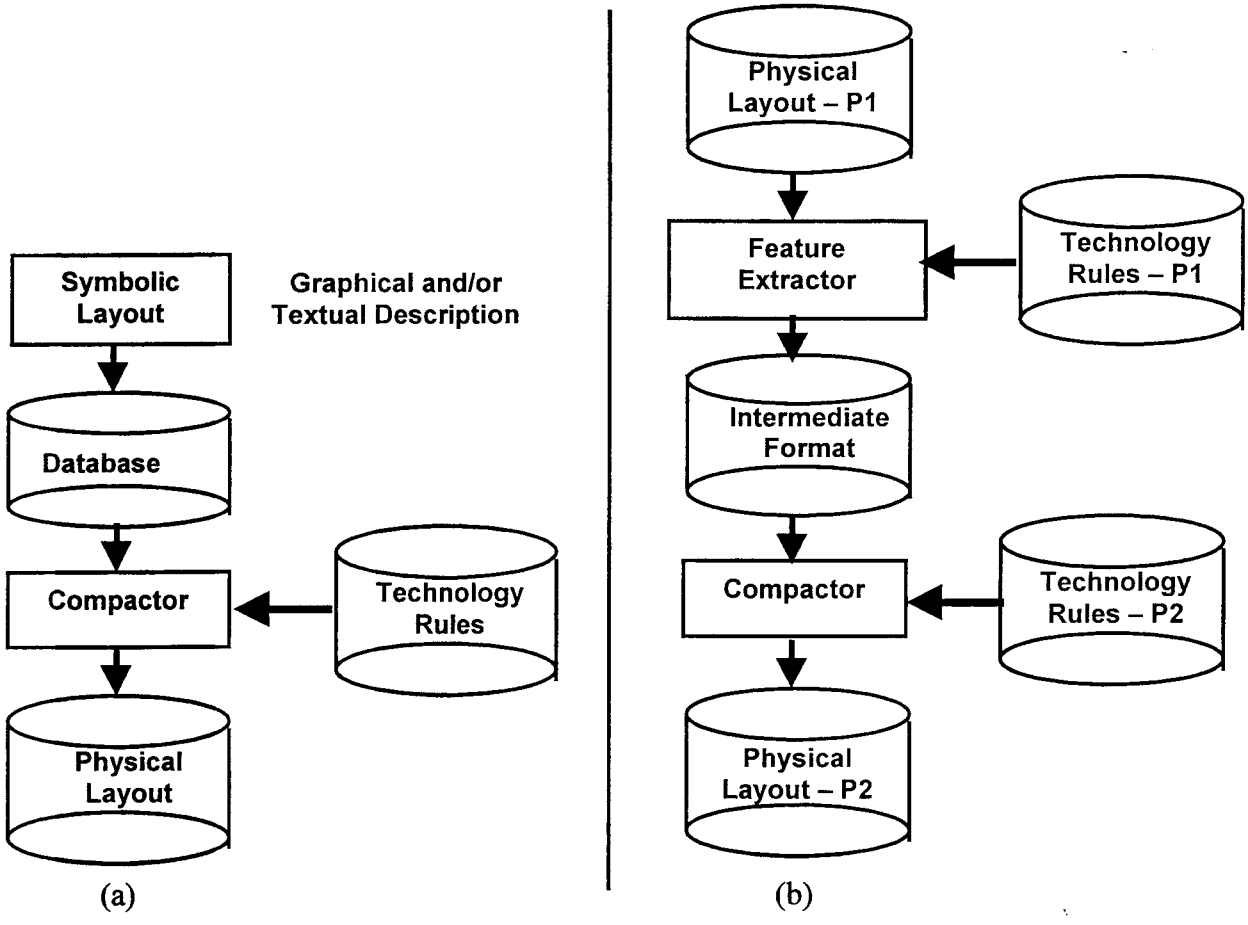


Figure 1. Using compaction to translate symbolic layout into physical layout (a) and using compaction to retarget a layout from a fabrication process P1 to a fabrication process P2 (b)

2. Introduction to Layout Compaction

The layout designer is driven by two conflicting demands. The first is to find the layout with an area as small as possible, in order to minimize manufacturing cost and maximize circuit performance. The second is to create the layout quickly to minimize design time and cost. Compaction may also be required when one wants to convert a layout from an older fabrication process to a new one (using legacy layouts), as illustrated by figure 1b.

The designer usually develops the layout during two major layout phases [1,2,3].

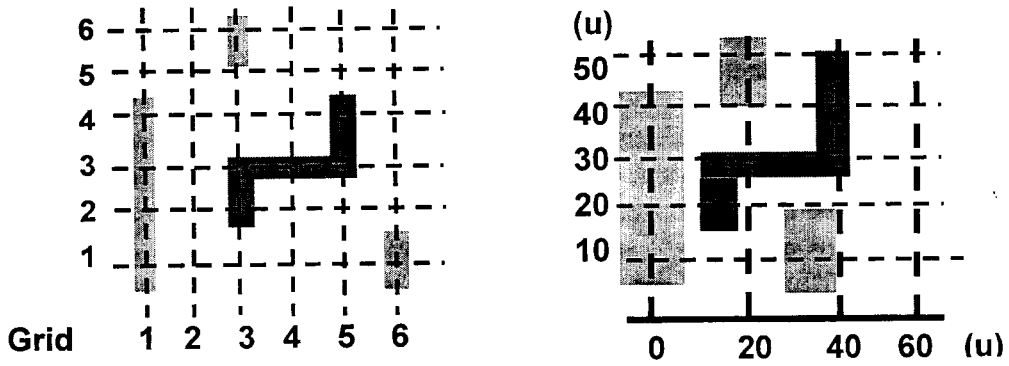
1. *Topological design*, after which the relative placement of components and wires are defined.
2. *Geometrical design*, after which the geometrical (physical) positions of all the layout elements are defined.

The designer uses different design methodologies to solve the problems of these two phases. *Symbolic Layout* and *compaction* are two closely related design methodologies that encourage the separation of topological design and geometrical design and help to automate geometrical design. Symbolic layout allows the topological designer to work with transistors, wires, and cells as primitives, rather than manipulating the individual polygons used in fabrication. A symbolic layout can be drawn as a *stick diagram*, which uses line segments and components as symbols, or as a layout display with wires and devices drawn as rectangles similar or identical to those used in the layout. The stick diagram may clarify the cell topology, while the layout display gives the designer a feel for the relative sizes of layout elements. The compactor usually moves subcells only in the plane, preserving the designer's topology for the cell.

Figure 2 (a) illustrates a symbolic layout using a virtual grid and figure 2 (b) illustrates its corresponding physical layout. Figures 2(c) and 2 (d) illustrate a more realistic example.

Compaction performs a translation from the graph domain into mask geometry. Compaction is more than just an *optimization problem.* Compaction is quite a difficult problem not only from a combinatorial, but also from a systems point of view. Advanced compactors aim at separating combinatorial and technological issues as much as possible. Compaction algorithms use the information from the design-rule database to compute the mask features. The structure of information that the compaction algorithm has to extract from the design-rule database can be quite complicated. It involves not only simple numerical parameters, such as minimum width and minimum distances between features of different masks, but also connectivity information and electrical information. The positioning of tubs around transistors and substrate contacts are additional problems to be dealt with. Most of the steps in the compaction process are made much simpler by working with a symbolic description of the layout.

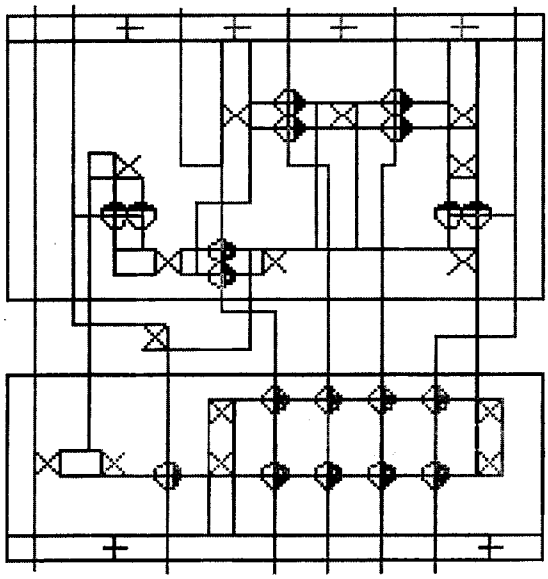
Combining compaction with symbolic layout creates advantages for the *computer-aided design* developer as well. Several companies provide symbolic layout systems [30,31,32].



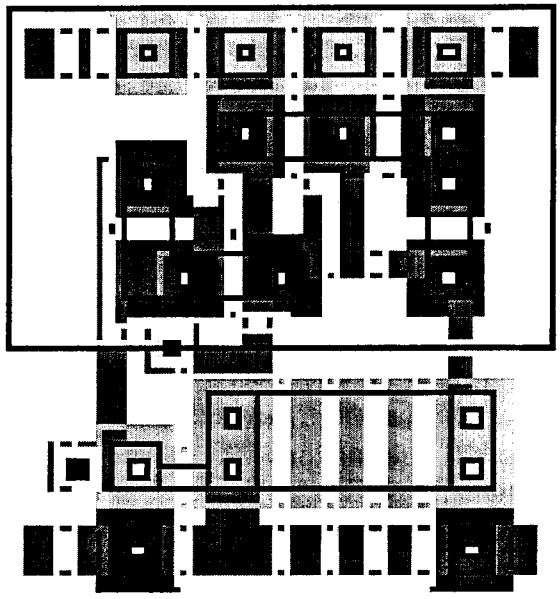
(a) Using a Virtual Grid (b) Physical Layout Associated to (a)

Figures 2a and 2b - Symbolic Layout

The most common method used to implement symbolic layout on systems which do not support the concept of true symbolic layout (and there seems to be only one that does – the US$ 100,000.00+ Compass ROSE system) is the "unit cell approach". In this method, many small cells such as poly contact, via, N- and P-transistor, etc. are constructed. If the user wishes to have several different sizes or shapes of a particular part, he either constructs more different kinds of unit cells or else he places many small unit cells close together so that they will overlap and create the larger area needed. The difficulties caused by this method may be readily seen: First, the unit cells are not truly symbolic; that is, they are icons which "represent" functional elements, they "are" the functional elements. As such, they must be individually redefined and rebuilt for each technology with which they are used. They do not simplify the layout, but, in fact, complicate it by adding all kinds of extra detail, especially if arrays of unit cells are used to construct larger elements.



(c) Virtual Layout – a Realistic Example



(d) Physical Layout associated to (c)

Figures 2c and 2d - Symbolic Layout – a Realistic Example

A much better way to use the "unit cell" approach (as used in the PILGREM system [32]) is to draw a figure, or icon, such as a circle, diamond, square, or cross, on a special layer and define it as a cell. Thus, a green circle might be used to represent an N-Transistor, while a blue circle could mean a P-transistor. Similarly, a green cross might stand for a contact between N-diffusion and metal and a blue one for contact to P-diffusion. If the sizes of the symbolic shapes are chosen judiciously, the symbol-to-symbol spacings can be simplified down to two or three cases, thus making the system simple to use. With this method, it is necessary to use a computer program to analyze the physical layout data, searching out the cell placements and converting them either to true symbolic placement entities, as in the PILGREM system [32], or replacing them with the desired polygons on the correct layers to construct the actual layout database.

A second method for the implementation of symbolic layout on non-symbolic systems is the "symbolic layer" approach. With this method, which is the basis of the previously mentioned ROSE system [31], a set of special layers is defined, one for each type of symbolic entity. Points, lines, and rectangles placed on these layers are used to define functional elements of varying sizes and shapes. This "stretchable" method is very powerful because it decreases the amount of effort required to define long or wide transistors, interconnect paths, various sizes of capacitors, resitors, and so forth. The number of different symbolic entities is kept to a minimum, because a single definition serves for all different sizes of that particular element. Finally, the amount of data required for a design is very small.

The only disadvantage of the "symbolic layer" system is its lack of identifying icons, but this can be alleviated by combining the "unit cell" and "symbolic layer" methods in the same design. Unit cells are used for minimum size elements and to mark the corners or ends of stretched elements, which are implemented with the "symbolic layer" scheme. Interconnect is usually shown as "symbolic layer" lines or rectangles without icons so that the overall appearance of the layout is one of icons wired together with lines. More details of this can be found at http://www.ic-dc.com.

3. Taxonomy of Compaction Algorithms

The most common classification scheme is based on the *direction of movement* of the components (features): *one-dimensional* (1-D) and *two-dimensional* (2-D) and is presented below[1]. A second approach to classify the compaction algorithms can be based on the technique for computing the minimum distance between features (*constraint-graph based compaction* and *virtual grid compaction*). In addition, compaction algorithms can also be classified on the basis of the hierarchy of the circuit. If compaction is applied to different levels of the layout, it is called *hierarchical compaction.* Any of the methods to be described can be extended to hierarchical compaction.

3.1. One-Dimensional Compaction (1-D)

3.1.1. Virtual Grid Algorithm

A. Basic Virtual Grid Algorithm.

Virtual grid layout approach to symbolic design uses a virtual grid to establish the relative placement of circuit elements and does not correspond to a physical grid [4,5,6]. In the virtual grid approach the compactor gives locations to the virtual grid lines, not to the circuit elements themselves. This imposes an arbitrary constraint on a virtual grid design; all elements that fall on a virtual grid are given the same location. This is illustrated by figure 3.

Virtual grid compaction begins by first examining spacings in one direction only (e.g. use the X direction as the first compaction pass). Each X grid line is compared with parallel neighboring X grid lines for spacing requirements. A spacing is required with a neighboring X grid if elements on two neighboring X grids have the same Y coordinate value. The final spacing on an X grid will be the greatest spacing it has encountered with respect to its parallel neighbors. In order to accommodate spacings that exert their influence over a number of symbolic grid lines, *backtracking* must be done. The grid line to be placed is compared with previously placed columns until the distance between the current column and the prior column exceeds some worst-case process value.

During Y compaction (the second compaction pass) corner spacings are accomodated. An element being placed during Y compaction is spaced against diagonal neighbors within the worst-case process window as well as against its adjacent neighbors. All diagonal spacing requirements are handled during the second compaction pass.

The virtual grid algorithm finds the subcell positions by considering the layout to be drawn on a grid; it moves all components on a grid line together so that adjacent virtual grid lines are as close as possible while satisfying all required separations between the symbols on the grid. Improvements of the algorithm, such as the *split-grid* method [7,8] allow for the movement of groups of components, achieving good compaction results.

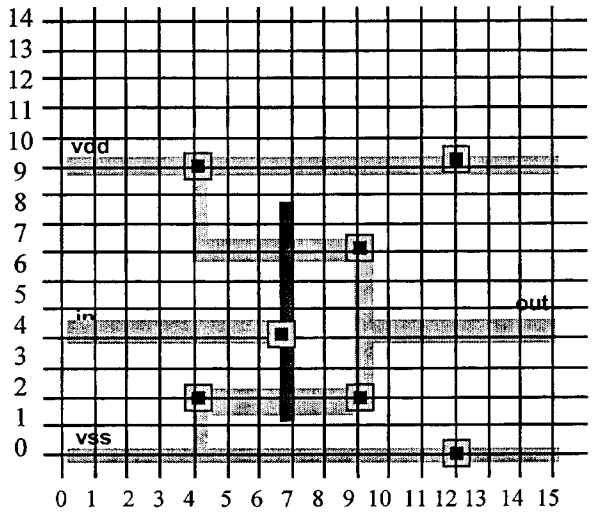


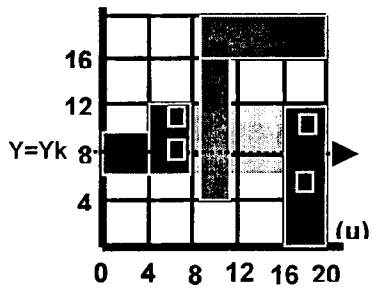
Figure 3 – Virtual Grid Layout

B. Fence Compaction Algorithm.

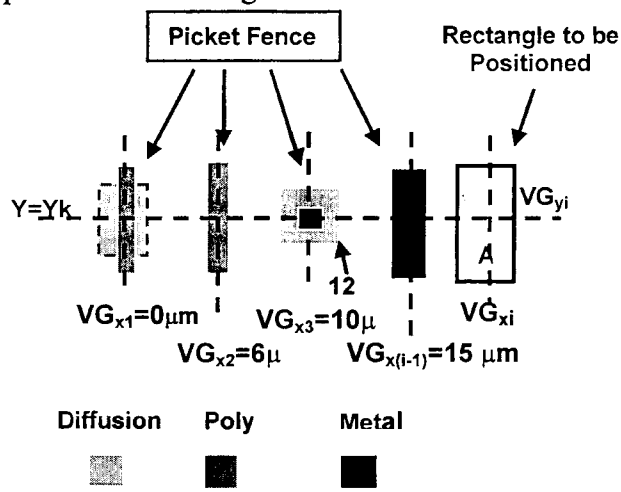
Fence compaction [6] eliminates the need to do *backtracking* in a virtual grid compaction and is linear in run time. The fence compaction technique is similar to the *shadow-propagation* method [1]. The fence algorithm, in effect, combines the graph building step with analysis of the critical path. The virtual grid limits the potential locations of the symbolic elements. This permits the use of a "picket fence" data structure for each grid line.

A "picket fence" data structure records the last placement of each layer for the given process, and thereby keeps track of the necessary parallel neighbor information. There is a fence data structure for each Y grid line. Each fence, in turn, has one "picket" for each layer of the process that is being compacted for. Each picket contains a rectangle that represents the last placement of a rectangle of the corresponding layer along the fence structure's grid line. The contour of a given layer is captured by the pickets for that layer.

During the second compaction pass Y grid lines are placed in much the same way as grid lines were placed during X compaction. Diagonal constraints are handled during this compaction pass and the actual Euclidean distances for the corners being spaced are used. Figure 4 illustrates the contents of a picket fence for a certain Y grid line.



(a) Layout



(b) Elements in a Virtual Grid

Figure 4 – A picket fence consists of a list (table) of all elements on a given Y (X) line, against which the rectangle to be positioned has to be compared.

LAYER	VIRTUAL GRID	POSITION
Metal	VGx(i-1)	17 um
Poly	VGx2	9 um
Diff	VGx3	12 um

(c) Most Recent Layers Table

Compaction starts by assigning the leftmost virtual grid the mask position 0. The other virtual grid lines are assigned mask positions in order by looking at the components that have already been placed[6]. Consider the situation shown in figure 4, where the virtual grid VG_{xi} is being assigned a mask position and VG_{x1} trough $VG_{x(i-1)}$ have been assigned mask positions. The inner loop of the spacing algorithm determines the required spacing from rectangle A (fig. 4b), a rectangle VGxi, and the rectangles to its left. Therefore, the most recent rectangles in the grid line form a *picket fence*

against which distances to rectangle A have to be compared. The most recent layers table keeps track of the position of the last mask element seen on each layer. For example, the most recently seen diffusion element was on VG_{x3}; the required spacing is measured from the element's right edge, which is at 12um. That element hides an earlier diffusion element on VG_{x1}.

The virtual grid can be placed immediately because the constraints on it are part of an acyclic graph. Once VG_{xi} has been placed, the most recent layers table can be updated and the algorithm can place the next virtual grid.

After both x and y compaction, virtual grid compactors use an *xy compaction step* to reduce the cell size because the rectangular barrier introduced around the components is too pessimistic at the corners of the cell. The xy compaction step looks for adjacent components that can be moved closer together and adjusts the positions of their virtual grid lines. The xy compaction step is usually performed simultaneously with the second x or y compaction step since it must look at the same constraints.

C. Split Grid Compaction Algorithm

This method allows objects on a virtual grid line to move independently, but does not allow connections with slopes. Split-grid compaction produces layout closer in quality to a constraint-graph algorithm compaction, at the cost of more CPU time than required for simple virtual grid algorithms.

A split grid compactor starts with a virtual grid layout. The virtual grid is retained to organize the topology of the circuit. The grid lines are split apart into distinct *circuit groups* which are collections of circuit elements that lie on the same grid line and are electrically connected. *The compactor gives locations to groups of circuit elements rather than virtual grid lines*. Split grid compaction is a hybrid approach to compaction; it retains the simplicity and speed of virtual grid compaction and produces area results that are similar to those produced by a graph based compactor – DASL Compactor[7,8].

In the approach presented in [8], groups are identified and placed as close to the fence as possible. Upper bound constraints are handled for the element rectangles in this approach by using "layer decoupling". Layer decoupling allows wires and contacts to slide with respect to a groups center line (contacts and wires can be offset).

A modified "picket fence" data structure is used in [7]. A topological data structure termed grid point data structure (GPDS) is used to facilitate nearest neighbor access (figure 5b). Fast group identification as well as fast compaction is possible since nearest neighbor information can be obtained in constant time. The X compaction pass is similar to the X compaction pass presented above for virtual grid compaction except that groups are being placed instead of virtual grids (the fence structure is the same for both). During the Y compaction pass there is one fence structure for each X group (not each X virtual grid). Y group placement is similar to Y grid placement. Run times as fast as those reported for a virtual grid compactor and area results as small as those produced by constraint graph compactor have been reported [7].

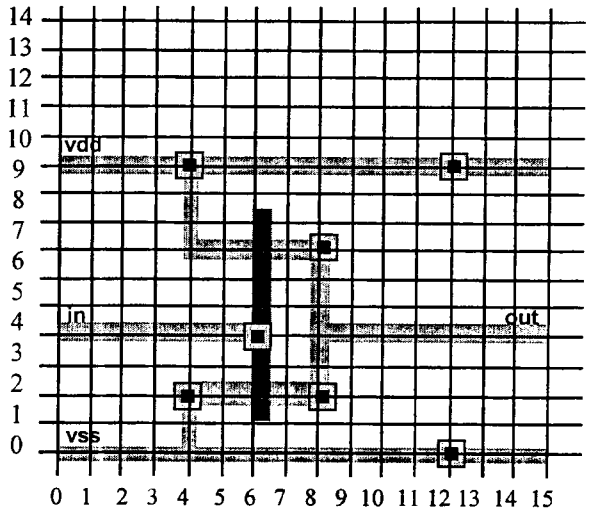


Fig. 5a – Virtual Grid Symbolic Inverter

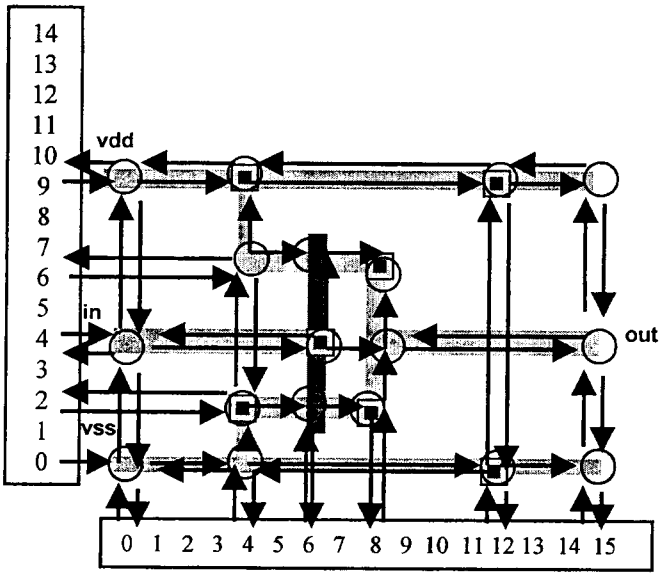


Fig. 5b – Grid Point Data Structure (GPDS) for the cell of figure 5a.

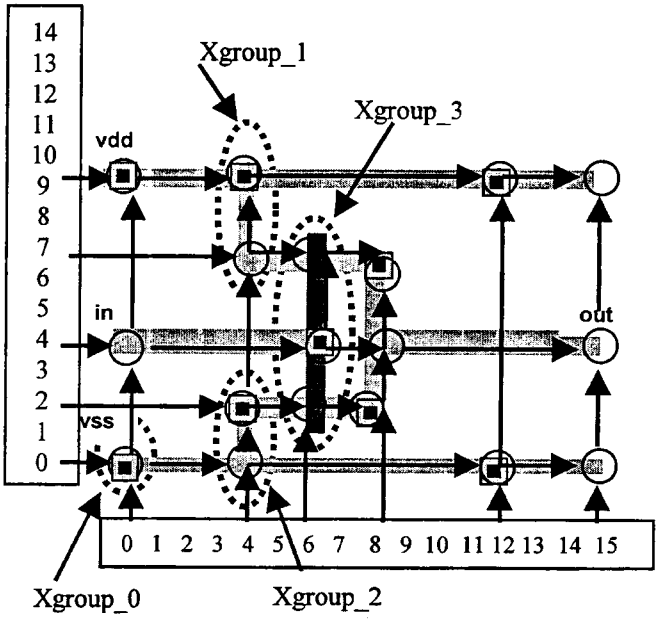


Fig. 6a – Virtual Grid showing Xgroups

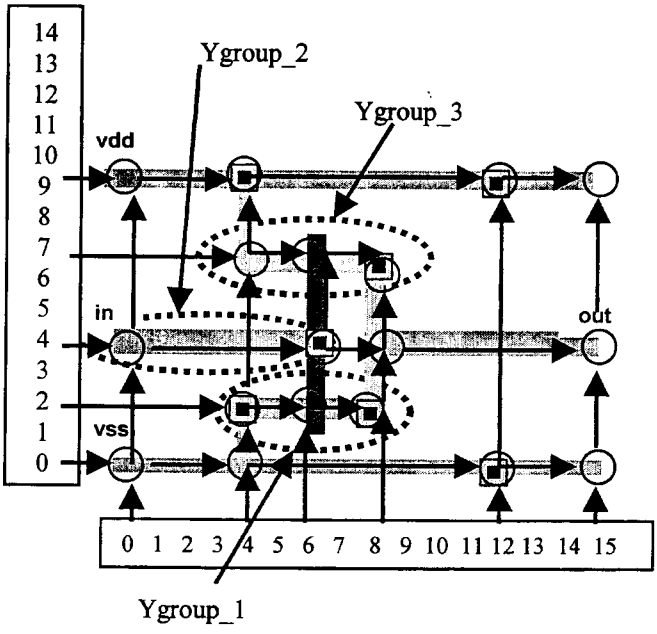


Fig. 6b – Virtual Grid showing Ygroups

In the split grid compaction method, the compactor places distinct circuit groups that fall on the same virtual grid separately, splitting apart the virtual grids where appropriate.

At the center of the split grid compactor [8] is a data structure that allows fast access to the circuit elements. Each occupied grid point is allocated a grid point data structure (GPDS) which contains a list of elements appearing at that point. A wire appears at a grid point if it ends or makes a turn at that point. Contacts and pins appear at a single grid point each. Devices (P and N transistors) appear at each grid point they cover, depending on their length and width. This means that it is possible to specify different size transistors using symbolic layout and a split grid representation. A device will cover at least three grid points (for unit devices) permitting connection to the device's source, gate and drain. Each occupied point is linked to its nearest horizontal and vertical occupied neighbors so

that nearest neighbor information can be determined in constant time. Access to the GPDS matrix is provided by two "edge vectors" (X and Y edge vectors). A pointer slot is provided for each virtual grid line. A given slot will be empty if no elements fall on the virtual grid line, otherwise the first grid point to have an element on it will be pointed to (figure 5b).

As illustrated in figure 6a, a group is defined as a collection of circuit elements falling on the same virtual grid line that need to be placed together. There are two situations that can result in the elements on this grid line being grouped together. The first situation is if the elements are connected by a vertical wire segment. The second is if the elements are connected to a vertical transistor. Figure 6b illustrates some vertical groups. These elements will be placed together, when converting from virtual grid coordinates to real coordinates. Groups are identified by traversing each grid line. Each GPDS on the grid line is visited once. When a new layer is encountered, a new group is initialized. Devices are an exception to this. A device takes up at least three grid points. The three points of a device are given the same group number. A group is ended when no layers associated with the group extend to the next GPDS.

The first compaction pass (X in this description) proceeds in a similar way to vertical grid compaction. A group is compacted by determining the spacing necessary for each element at each GPDS in the group. Elements are spaced with respect to elements in the neighboring X groups (the fence) sharing the same Y virtual grid coordinate. Vertical wires are kept track of dynamically, so that only occupied virtual grid points (GPDS points) are visited. The fence is checked for all points that a wire segment crosses.

After the first compaction pass each of the X groups is given a fence data structure. The fence is sorted in order of the physical placement of the groups. Each X group points to its fence data structure which is linked to its physical neighbors (figure 7).

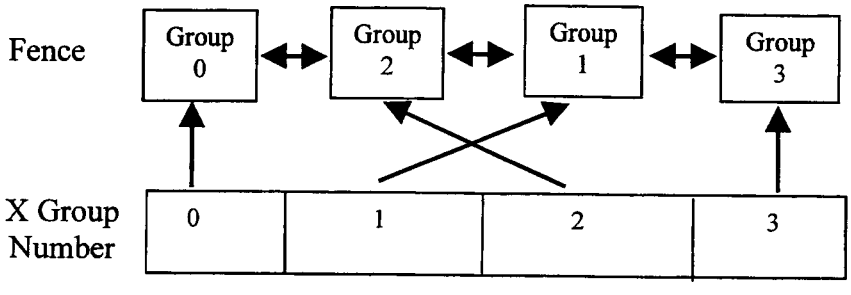


Fig. 7 – Fence for the Y compaction of the circuit of figure 6a. After X compaction assume the X groups are in the following order – 0,2,1,3 (location of group 1 is great than group 2). Each group points to its fence data structure.

To place a Y group, a given element in the group is checked against the corresponding group fences that fall within a design rule window of the location of its X group fence. This design rule window is dependent on the process being compacted for and is wide enough to account for any possible corner design rule interaction encoded in the process file. Since a given process will have some minimal resolution, the number of group fences in the design rule window will be N in the worse case, where N is the design rule window size divided by the process resolution. For diagonal spacing at most N groups will be checked, leading to a linear algorithm.

D. Compression-Ridge Method

This method was pioneered by Akers [9]. In this approach, a region of empty space is identified in the geometric layout, which separates the layout into a left and a right part. Such a region is subsequently removed by translation of the right part of the layout to the left by as much as the spacing constraints will allow. This step is repeated until no more compression ridges are found. A major problem with this algorithm is finding connected regions of white space that can be removed from the layout. Another major disadvantage of this method is that it progresses from the top to the bottom of the layout which amounts to a search through the layout with backtracking (if the compression-ridge method cannot progress further) and thus is computationally complex. Furthermore, it finds only horizontally convex compression ridges. Figure 8 illustrates the compression ridge method. Although this algorithm is historically important as the first compaction algorithm, modern systems have abandoned it for more powerful methods.

Compression ridges have the following properties:

1. Compression ridge is a constant width band of empty space stretching from one side of the layout to the other side.
2. Compression ridge can intersect wires that are perpendicular to it. However, it cannot intersect those wires that are parallel to it.
3. The width of a compressing ridge is such that when the space is removed, no design rules should be violated in the resulting layout.

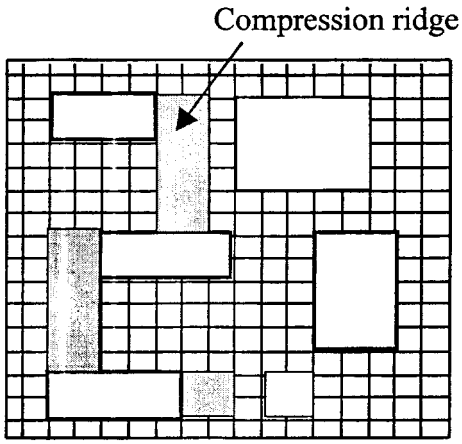


Fig. 8a – Compression ridge

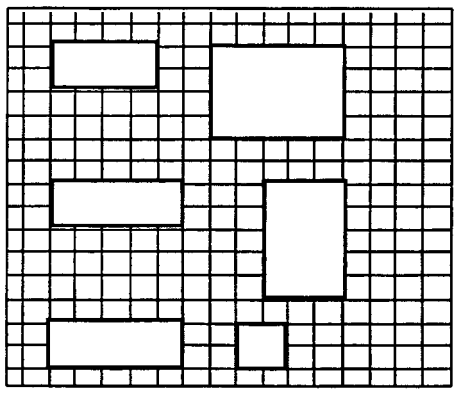


Fig. 8b – After compression

The *compression ridge* is shown by the shaded regions of figure 8a. In this technique, these compression ridges are subsequently removed from the layout until no more ridges are found. One method for finding the compression ridges is to use the *virtual split-grid method.* The vacant space along the grid line can be replaced by a rectangular compression ridge. The width of this ridge is the minimum compression that can be achieved along the grid line. Figure 8b shows the layout after removing the compression ridge.

3.1.2. Constraint-Graph Based Compaction

A. Constraint Graph Compaction Principles

In the constraint graph approach the designer typically places circuit element symbols on a fine grained *physical grid.* The compactor brings circuit elements as close together as design rule spacing requirements will allow. Some constraint graph compactors can also push apart circuit elements that are too close together. Locations are typically given to each circuit element. Some constraint graph

compactors, however, group together circuit elements whose center lines share the same physical grid line, and who are connected geometrically (electrically connected) and give the groups locations. Constraint graph compactors are one-dimensional compactors and require at least one X compaction pass and one Y compaction pass.

In this approach, the layout topology and spacing constraints are represented by a weighted directed graph called the constraint graph (see figures 9a and 9b). The construction of this graph is the time consuming step of constraint graph compaction. Among the approaches that have been used to construct this graph are the *shadow propagation method* [10] and the *scanline algorithm* [11].

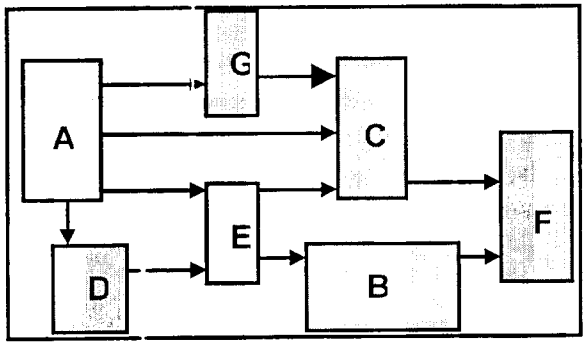


Fig. 9a – Layout with indication of distance restrictions

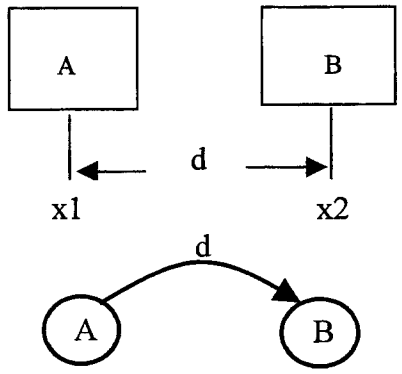


Fig. 10a – Separation Constraint

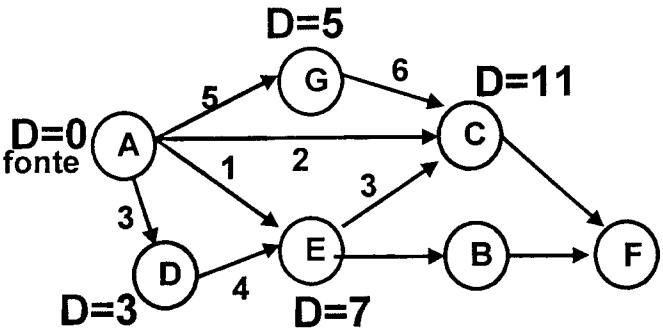


Fig. 9b – Wheighted graph associated to figure 9a

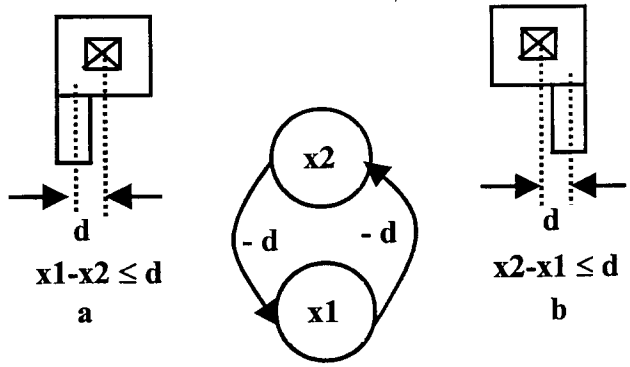


Fig. 10b – Connectivity Constraints – graph with backward edge

The constraint graph G (V,E), is a weighted graph. Each vertex v ∈ V represents a component while edges represent constraints. There are two types of constraints that should be satisfied in the process of compaction: *separation constraints* and *physical connectivity constraints*. Both separation constraints and physical connectivity constraints can be incorporated into a graph representing a 1-D compaction problem. A separation constraint can be represented by a directed edge with a weight equal to the minimum separation (figure 10a). Assuming that A is to the left of B in a coordinate system, this rule can be written as $Bx \geq Ax + d$, where Ax refers to the x-location of element A. It can also be written as $x2 \geq x1 + d$, where x2 represents Bx and x1 represents Ax. Figure 10b shows the *connectivity constraints* that require two components to be within a distance d of each other. The equation for this is: $|x2-x1| \leq d$. Figure 10b illustrates the case that requires a wire to be within a distance d of the center of the contact, and vice-versa. A physical connection can be represented as a cycle of two edges.

The constraint graph includes two additional vertices, L and R, which represent two physical boundaries (Left and Right). L can be thought of as a source of the constraint graph and R can be considered as the sink of the graph.

B. Constraint Graph Generation

The first step necessary in constraint graph compaction is to build the constraint graph. This is the most time-consuming part and is $O(n^2)$ in the worst case. This is due to the fact that, in the worst case, there is an edge between every pair of vertices in the constraint graph. Only a small subset of the potential edges are actually needed for constraint graph compaction. A circuit component group typically will only have spacing requirements with its nearest neighbors. Many techniques for generating the constraint graph efficiently have been proposed [10,11]. In construction of a constraint graph, the connectivity constraints are generated first. The connectivity constraints can be generated by scanning all legal connections in the symbolic layout. The connectivity information is usually stored in a table and the compactor looks up the table to generate all the constraints. Different types of connectivity constraints include wire-wire, wire-via, wire-source connectivity constraints.

Separation constraints are generated once per compaction step. The constraint generation method used should ideally generate a non-redundant set of constraints since the cost of solving the constraint graph is proportional to the number of edges in the graph, or the number of constraints.

Several constraint graph generation algorithms have been proposed, and two of the most well known are the *Shadow-Propagation Algorithm* [1,2,3,10] and the *Scanline Algorithm*[1,2,3,11].

C. Algorithm for Calculating the Critical Path in an Acyclic Graph

Once the construction step is complete, the constraint graph is used to assign x-locations to the vertices so as to satisfy the constraints and minimize the layout width. This is achieved by assigning x-values to the vertices so as to minimize the length of the longest L to R path(s) (the critical path(s)). The length of a path is equal to the sum of the weights of those arcs traversed by the path. Central to the constraint graph approach is the critical path algorithm. This algorithm is used to find for each vertex v, the range of x-locations [l(v), r(v)] that can be tolerated by that vertex. Let *LengthFromL(v)* be the length of the longest path from the source *L* to *vertex v*. Similarly, define *LengthToR(v)* to be the length of the longest path from *vertex v* to sink *R*. Therefore, the range of movement tolerance for each vertex v can be defined as follows:

$l(v) = LengthFromL(v)$
$r(v) = LengthFromL(R) - LengthToR(v)$

It can be shown that l(L)=r(L)=0, and l(R)=r(R) = *LengthFromL(R)*, which is the length of the longest path from the source L to the sink R. Also, based on the principle of optimality, for each vertex v on one of the critical path(s), l(v) = r(v). For acyclic graphs, the critical path problem is straightforward. A semi-formal description of such an algorithm is given in Figure 11.

Algorithm Critical-Path (L, R, G);

(* G: Constraint graph *)
(* L: source vertex of graph G *)
(* R: sink vertex of graph G *)
(* LengthFromL(i): length of longest path from L to vertex i *)
(* LengthToR(i): length of longest path from i to R *)
(* di,j : weight of edge (i,j) (spacing constraint between nodes i and j) *)
(* T+(i): successor vertices of i *)
(* T-(i): predecessor vertices of i *)

Begin

1. (* Initialization: levelize graph *)
   Let K be the number of levels, and Vi the vertices in level i;
   Therefore, V1 = {L} and Vk = {R};
2. (* Forward trace *)
   LengthFromL(L) = 0;
   For K=2 **To** K **Do**
   ForEach i $\in$ Vk **Do**
   LengthFromL(i) <- $\max_{j \in T-(i)} \{\text{LengthFromL}(j) + d_{j,i}\}$
   EndForEach
   EndFor;

3. (*Backward trace and compute tolerance ranges *)
   LengthToR(R) = 0;
   For k= K-1 **DownTo** 1 **Do**
   ForEach i $\in$ Vk **Do**
   LengthToR(i) <- $\max_{j \in T+(i)} \{\text{LengthToR}(j) + d_{i,j}\}$;
   l(i) = LengthFromL(i);
   r(i) = LengthFromL(R) – LenghToR(i)
   EndForEach
   EndFor;

End.

D. Example of Constrain Figure 11 – Calculating the Critical Path in an Acyclic Graph

Once the moving tolerances for all vertices have been computed, the next task is to assign locations to the corresponding elements consistently with their tolerance ranges. To do that, we need to have a moving strategy. There are three general moving techniques: the minimum, the maximum, and the optimum strategies. The minimum moving strategy assigns each element closest to the right boundary of the chip, i.e., for each vertex v, x(v)=r(v). This is the strategy used in the *plowing* technique of Magic[12].

The maximum moving strategy assigns elements closest to the left boundary of the chip, i.e., for each vertex v, x(v)=l(v). Both these strategies are simple and lead to layouts that are densely packed on either side of the chip. This is undesirable since it leads to an increase in the overall wire length.

An optimum moving strategy attempts to assign x-locations to elements according to some optimization criterion. A possible criterion would be to assign each element to the middle of iths range of tolerance, i.e., for each vertex v, x(v)= (l(v)+r(v))/2. Another possible criterion is to use a force-directed approach. In this case, the criterion would be to minimize the sum of the forces acting on each element. The objective minimized in this case is the sum of the squares of the x-distances between the element and its neighbors.

The general steps of the constraint graph compaction approach are summarized in figure below.

Algorithm Constraint_Graph_Compaction;

1. Construct the constraint graph G(V,E);
2. Apply the critical path algorithm and find for each vertex v, [l(v),r(v)];
3. Move each element to within its range of tolerance;

End.

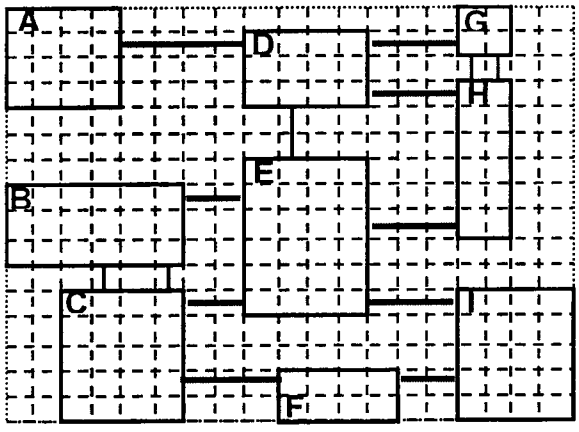


Fig. 12a – Initial Layout

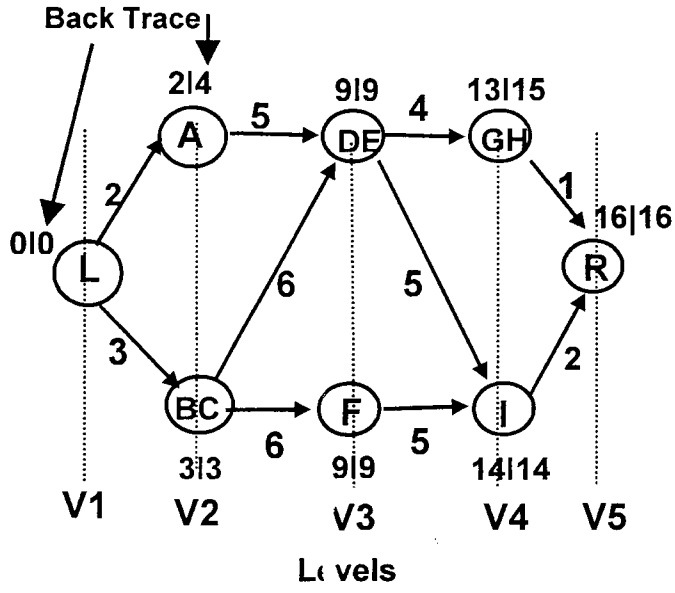


Fig 12b – Constraint Graph of figure 12a

Consider the Layout of figure 12a with minimum horizontal distances centre to centre:
dA,D=5 ; dA,E=5 ; dC,E=5 ; dC,F=5 ; dD,G=4
dD,H=4 ; dE,H=4 ; dE,I=5 ; dF,I=5

Consider the following element width:
wA=4 ; wB=6 ; wC=4 ; wD=4 ; wE=4; wF=4;
wG=2 ; wH=2 ; wI =4

Since blocks B and C are connected by vertical wires, they are grouped to move equally in the x axis. The same applies to blocks D, E, G and H.

The minimum distance $d_{i,j}$ between two groups K_i and K_j is defined as follows:

$$d_{i,j} = \max_{a \in Ki \,\&\, b \in Kj} \{d_{a,b}\}$$

Thus:
dA,DE=max{5,5}=5 ; dL,A=2 ; dL,BC=3, dBC,DE=6 ;
dBC,F=6 ; dDE,GH=4 ; dDE,I=5 ; dFI=5 ; dGH,R=1 ;
e dI,R=2.

The graph has 5 levels. Let Vi be the vertices of level i. Then, V1={L}, V2={A,BC}, V3={DE,F}, V4={GH,I}, e V5={R}.

The next step is to perform a forward trace of the graph, where, for each vertex i, the values *LengthFromL(i)* are determined as described in the algorithm of figure 11. For this example, we obtain the following:

Level 1:

LengthFromL(L)=0.

Level 2:

LengthFromL(A) = *LengthFromL(L)* + dL,A = 0+2=2;

LengthFromL(BC) = *LengthFromL(L)* + dL,BC = 0+3=3.

Level 3:

LengthFromL(DE) = *max* { *LengthFromL(A)* + dA,DE;

LengthFromL(BC) + *dBC,DE* } = *max*{7,9} = 9;

LengthFromL(F) = *max* { *LengthFromL(BC)* + *dBC,F*} = 9.

Level 4:

LengthFromL(GH) = *LengthFromL(DE)* + dDE,GH = 13;

LengthFromL(I) = *max* { *LengthFromL(DE)* + dDE,I ;

LengthFromL(F) + *dF,I* } = *max* {14,14} = 14.

Level 5:

LengthFromL(R) = *max* { *LengthFromL(GH)* + dGH,R;

LengthFromL(I) + dI,R } = max {14,16} = 16.

The next step is to perform a backward trace of the graph. During this step, for each vertex *v*, the values *LengthToR(v), l(v)* and *r(v)* are computed as described in the algorithm of figure 11. Hence, the following is obtained for all vertices of the graph:

vertex *R*:	*LengthToR(R)* = 0;	l(R) = 16,	r(R)=16- 0=16;
vertex GH:	*LengthToR(GH)* =1;	l(GH) = 13,	r(GH)=16-1=15;
vertex I:	*LengthToR(I)* = 2;	l(I) = 14;	r(I) = 16-2=14;
vertex DE:	*LengthToR(DE)* =7;	l(DE)=0,	r(DE)=16-7=9;
vertex F:	*LengthToR(F)* =7;	l(F)=9,	r(F) = 16-7=9;
vertex A:	*LengthToR(A)* =12;	l(A)=2,	r(A)=16-12=4;
vertex BC:	*LengthToR(BC)* =13;	l(BC)=3,	r(BC)=16-13=3;
vertex L:	*LengthToR(L)=16;*	*l(L)=0,*	*r(L)=16-16=0;*

The final step of the algorithm is to find the x-location of the center of each group. Let xmin(i), xmax(i), and xopt(i) be the x-location of the center of group i according to the minimum, maximum, and optimum strategies respectively. For this example, assuming that xopt(i)=(l(i) + r(i))/2, these variables will be assigned the following values:

vertex *R*:	xmin(R) = 16;	xmax(R)=16,	xopt(R) = 16;
vertex GH:	xmin(GH) = 15;	xmax(GH) = 13;	xopt(GH) = 14;
vertex I:	xmin(I) = 14;	xmax(I) = 14;	xopt(I) = 14;
vertex DE:	xmin(DE) = 9;	xmax(DE) = 9;	xopt(DE) = 9;
vertex F:	xmin(F) = 9;	xmax(F) = 9;	xopt(F) = 9;
vertex A:	xmin(A) = 4;	xmax(A) = 2;	xopt(A) = 3;
vertex BC;	xmin(BC) = 3;	xmax(BC) = 3;	xopt(BC) = 3;
vertex L:	xmin(L) = 0;	xmax(L) = 0;	xopt(L) = 0;

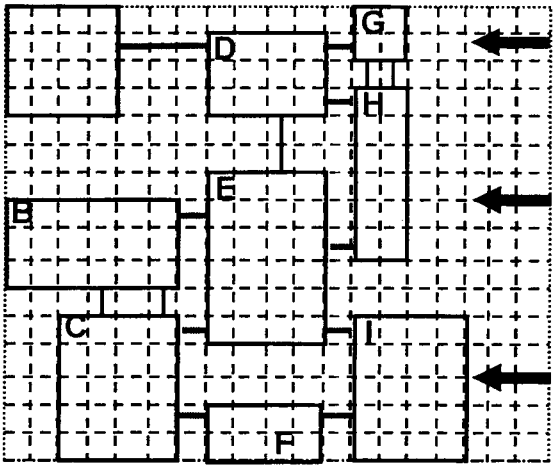


Figure 13 – Resulting Compacted Layout when a horizontal maximum moving strategy is adopted.

E. Wire Jogging

The wire jogging problem has similarities with the problem of routing and there are many papers in this area [13,14]. One of the first approaches to jogging wires was reported by Hsueh [10]. In this approach jogs in wires were introduced at "torque" points on a wire (see fig. 14). Wire jogging had limited success because it could reduce the size in one direction while, potentially, increasing the size in the other direction.

Several attempts have focused on reducing the critical path in a cell; jog points are only introduced along the critical path. Jogs are introduced in the graph domain in [15]. The critical path is updated after each wire is jogged and the new critical path is then searched for the next wire to be jogged. The process terminates when there are no wires on the critical path, or there is not enough room left to introduce a new jog. A vertical jog is introduced in a horizontal wire if there is a gap in the X intervals between the fan-in vertices of the wire vertex and the fan-out vertices. This, in a sense, puts a torque on the wire vertex if the fan-in vertices and the fan-out vertices are also on the critical path, or at least tightly constrained.

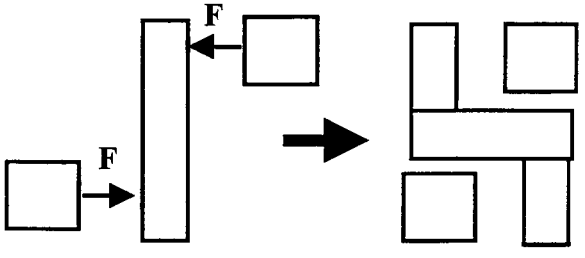


Figure 14 - Applying a torque to a wire results in a wire jog and a more compact layout

F. Wire Length Minimization.

Elements not on the critical compaction pass will find themselves pulled toward a cell edged because they are given their minimal legal spacing. This tends to increase wire length and reduce performance.

One of the first methods used to reduce wire length was the "average slack" method described in [10]. This approach uniformly distributed the empty space in a circuit among the elements that were not on the critical path. A force-based heuristic is described in [16]. It not only considers the effect of each wire layer but it also considers the cumulative effect of multiple wires connecting adjoining

modules (vertices in SPARCS are hierarchical so that a vertex could represent a module). This is effective in minimizing wires in a hierarchical cell.

The approach used in [17] "pulls back" elements that are not on the critical path as far back as they can go. The different wiring layers of the technology are given different weight factors, so that the diffusion layers, for example, are minimized preferentially. This method is iterative.

The method employed in [15] uses a non-iterative, event-driven algorithm to minimize the wire length. Groups of circuit elements will have preferential upward movement due to a wire connecting to it from above that needs its length to be minimized (wire widths are considered in this process as well as wire layers). The group is moved up until it is stopped by another group. These groups are merged if possible and moved up together. When the merged group is stopped the composite group will be broken up and if the top group is not constrained it will continue upward if it has a preferential upward movement. A global optimum is reported for this work and near linear run times are achieved due to its non-iterative approach.

3.1.3 Mixed Constraint Graph and Virtual Grid Compaction

This method used in the DASL system uses a mix of constraint graph and virtual grid compaction techniques [18,19]. This system differs from other systems in its ability to control de placement of substrate contacts and tubs, requiring less (or no) user intervention to produce a process independent design.

In constraint graph compaction if there is nothing to constrain and element's placement, it will slide as far to the left as possible (in X compaction). This is undesirable for substrate contacts which need to be spaced at reasonable intervals. The method controls the size of the tubs and the placement of the substrate contacts.

The DASL system is an outgrowth of the MULGA virtual grid compaction system [7]. In the MULGA virtual grid system tub generation was performed as follows: a minimal tub was assumed to surround each diffusion element during compaction. After compaction the tubs were grown until they intersected the "nearest" substrate contact. This approach works well for a virtual grid compactor but does not work for a constraint graph compactor because substrate contacts tend to slide toward the left (for X-compaction) along a horizontal power rail on which they are placed. This may cause some tubs to be stretched an unnecessarily long distance in order to enclose a substrate contact and can cause design rule violations since the final tubs are different than the tubs that were present during compaction.

The solution developed in the DASL system was to introduce "symbolic tubs" into the virtual grid design that is the input to the DASL compactor. The symbolic tubs are automatically generated and assure that design rule correct results are achieved because the compactor is dealing with the tubs that will be present in the final layout. The substrate contact sliding problem is constrained by minimizing the area of the tubs in a fashion described in [18].

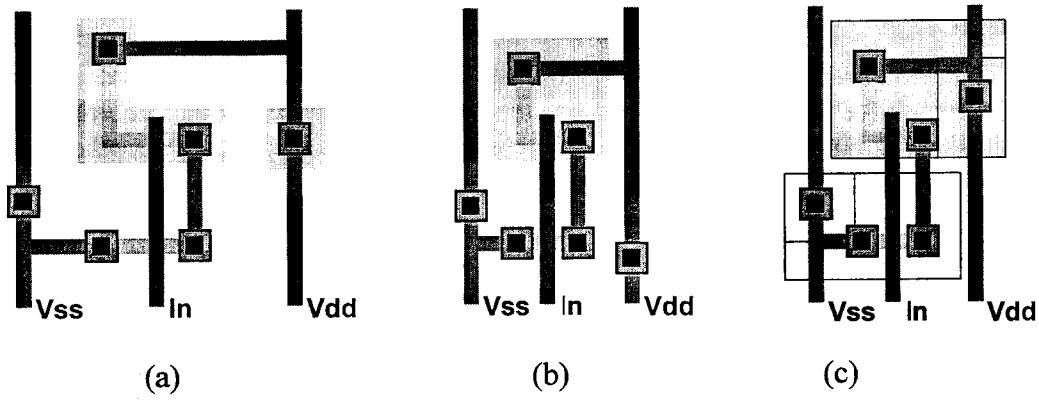


Figure 15. (a) An inverter with minimum tubs around each diffusion element before constraint graph compaction. (b) After compaction, the Vdd substrate contact slides down to the bottom of the Vdd rail since there is nothing to stop it. When Ntubs are grown to include a Vdd substrate contact they overlap the Ptubs. (c) The desired results are shown.

DASL compaction begins by splitting a *virtual grid* symbolic layout into distinct circuit groups. An initial compaction is done during which time a constraint graph is generated and the groups are given their minimal locations. The method employs the "*newave*" [19,20] constraint graph generation technique that combines the perpendicular *plane sweep* approach [16] with the *shadowing* technique[10]. The constraint graph is then used to place groups that are not on the longest path in order to minimize the areas of selected layers (wire minimization).

The first step in the DASL compaction scheme is to identify distinct groups that are on the same grid line. An X compaction group is composed of symbolic circuit elements that are connected along the same vertical grid line and that need to remain together in the mask version of the cell. This is thoroughly described in [7]. One of the grouping operations concerns grouping vertical symbolic tub edges. Tub edges that share the same vertical grid line and overlap will be placed in the same group.

The size of a tub cannot be predetermined because the tub must observe the correct overlap rules with diffusion (or active area) elements and must also be tied to a substrate contact. Therefore, the width and length of a tub is variable and is topology dependent. Since the size of a tub is not predetermined, the edges of a tub are treated as distinct compaction groups and placed separately (figure 16). The correct overlap of a tub to a diffusion element is accomplished by spacing a diffusion element with respect to the tub edge (tub group).

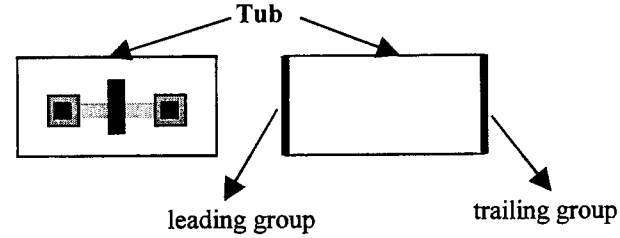


Figure 16. A device surrounded by a symbolic tub is found in (a). The tub groups are found in (b).

3.2. 1 ½-Dimensional Compaction

A compactor based on simulation of *zone refining* is presented in [21, 22] – the *Zorro* compactor. Although the compactor is based on simulation of an engineering process, it is a deterministic algorithm and differs sharply from other simulation based approaches such as simulated annealing and simulated evolution. The key idea is to provide enough *lateral movements* to blocks during compaction to resolve interferences. In that sense, this compactor can be considered 1 ½ dimensional compactor, since the geometry is not as free as in true 2-dimensional compaction.

The algorithm starts with a layout. Starting from one side, blocks are considered row by row and are re-arranged after they have been moved across the open zone. During its movement in the free zone, the blocks travel through the entire width of the layout and hence may be placed anywhere along the boundary. Figure 17 illustrates the process of compaction by zone refining.

The algorithm maintains an XY adjacency graph. In an XY adjacency graph, vertices represent blocks, while edges represent horizontal and vertical adjacency. That is, two blocks have a horizontal edge if they share a vertical boundary. Similarly, two blocks have a vertical edge if they share a horizontal boundary. The labels on the edges represent the minimum allowabel distance between blocks. Four additional vertices are added to keep all the blocks within the required bounded rectangle. Note that free space is ignored in computing the nieghborhood edges between blocks. Figure 18a illustrates an instance of problem along with its XY adjacency graph in figure 18b.

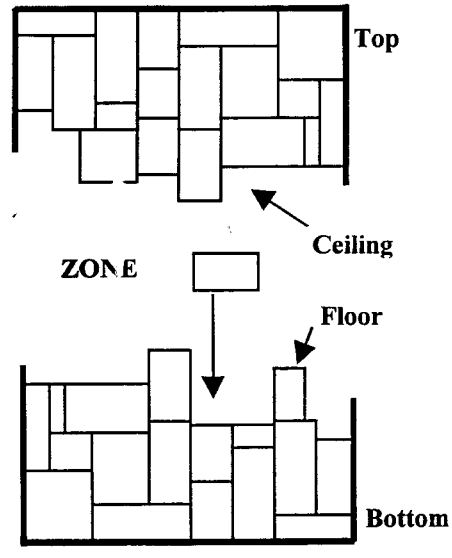


Figure 17 – Compaction by zone refining

The algorithm assumes that the input is a partially compacted layout, which can be obtained by two applications of a 1-D compactor. It maintains two lists called *floor* and *ceiling*. Floor consists of all the blocks which are visible from the top and may become a neighbor of future block. Ceiling is a list of all blocks which can be moved immediately. That is, ceiling is the list of blocks visible from the bottom. The algorithm selects the lowest block in the ceiling list and moves it to the place on the floor, which maximizes the gap between floor and ceiling. This process is continued until all blocks are moved from ceiling to floor.

To illustrate the algorithm, let us consider the example of figure 18. Since C is the lowest block in the ceiling list, it is selected for the move. Figure 18(c) shows that the gap is maximum at the boundary between blocks A and B. Therefore, C is moved between A and B. The modified layout and the XY-adjacency graph are shown in figures (d) and (e), respectively.

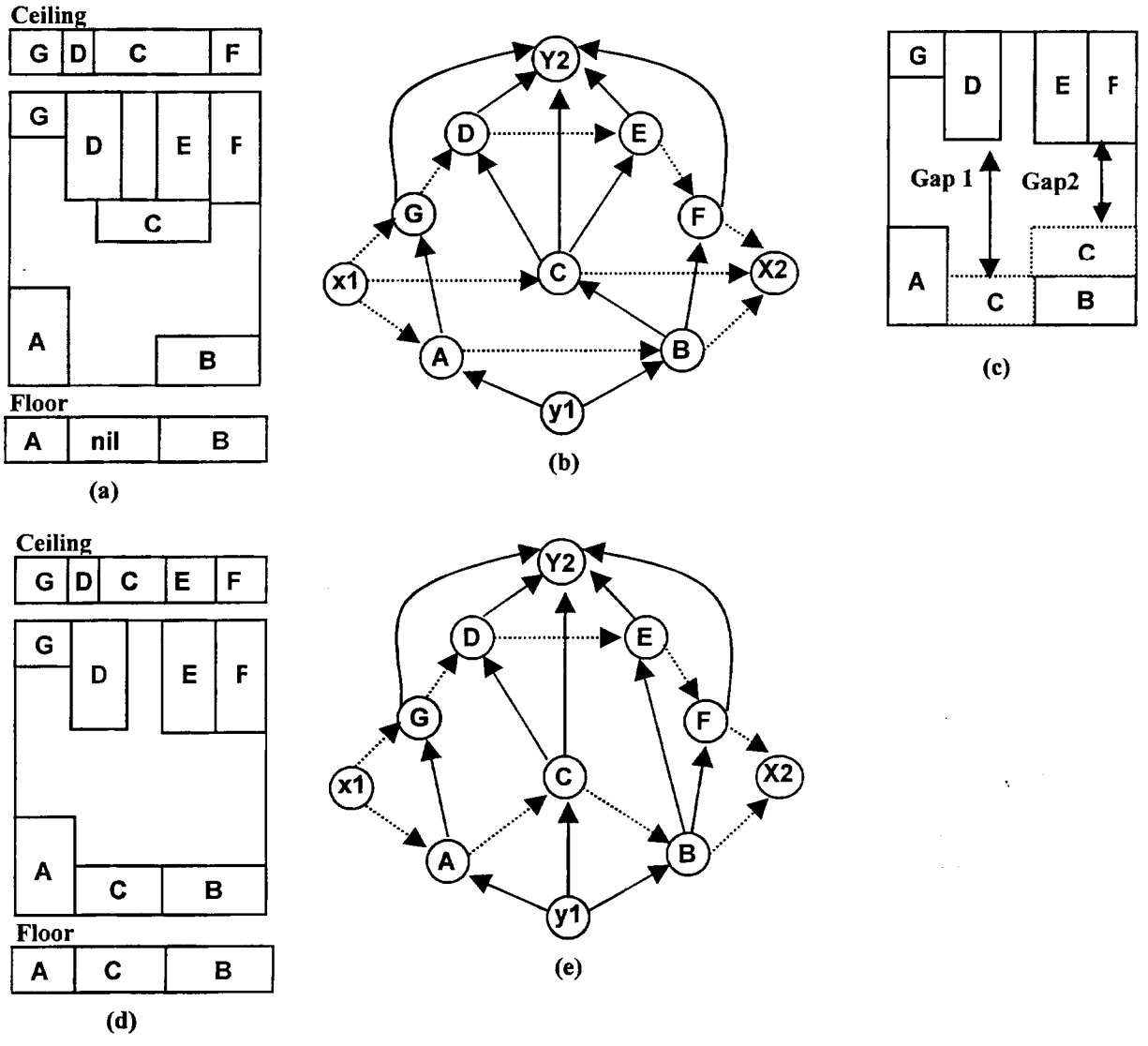


Figure 18 – Problem instance and its XY adjacency graph

3.2. Two-Dimensional (2-D) Compaction

In 2-D compaction, both x and y coordinates can be changed *simultaneously* in order to minimize area, i.e. in 2-D compaction, a single step can move a component in both x and y. In 1-D compaction, the cell is alternately compacted in x any y, while in 2-D compaction components are selected to move as required to improve the layout. Most versions of 2-D compaction are NP-hard [23,24]. The difficulty of 2-D compaction lies in determining how the two dimensions of the layout must interact to minimize the area. To circumvent the intrinsic complexity of this question, some heuristics are used to decide locally how the interaction is to take place.

Given a symbolic layout consisting of rectangular features, all the constraints (size, overlap and separation constraints) can be written using linear constraint equations. After the generation of constraints, the problem can be solved using integer linear programming technique. However, the complexity of the linear programming technique is exponential thereby making it impractical even for a moderate size problem.

3.3. Hierarchical Compaction

Modern designs are modularized hierarchically. Hierarchical compaction works on cells that are constructed from other cells as well as primitive layout symbols. Any of the techniques described so far can be made hierarchical by applying the compaction algorithm to a leaf cell and the using the compacted cell in a larger cell. Hierarchical compaction is also called *cell assembly*. A variety of hierarchical compaction algorithms have been developed for both constraint-graph algorithm and virtual grid algorithm compaction [25,26,27,28,29].

A hierarchical compactor can space fixed sized modules/elements with respect to other modules/elements. Additionally, a hierarchical compactor can also perform *pitchmatching*, that is, stretching cells so they can be connected by *abutment* (putting side by side). This can be performed by recompacting the cells with additional ports to be at the same positions of the other cell. Hierarchical compaction with stretchable cells requires the compactor to build a more complex model of a subcell than that required for a fixed-cell assembly.

4. Comparison Among Symbolic Layout Systems and Benchmarks

Tables 1 and 2 illustrate comparisons among symbolic layout systems, describing their main characteristics and benchmarks [33, 34].

COMPACTOR	Hier. Level B	Hier. Level C	Layout Style S	Layout Style M	Initial Positions MG	Initial Positions CG	Initial Positions VG	Initial Positions R	1D /1.5D / 2D
CUT	B		S			CG			1D
FLOSS		C	S					R	1D
STICKS		C	S					R	1D
SLIP	B			M	MG				1D
CABBAGE	B		S					R	1D
SLIM	B			M	MG			R	1D
MULGA	B		S				VG		1D
Liao/Wong	B		S					R	1D
Schlag et al.		C	S					R	2D
SQUASH	B		S				VG		1D
ROCHESTER	B		S					R	2D
VPACK	B		S				VG		1D
Wolf et al.	B		S					R	2D
Plowing	B			M	MG				1D
Kingsley		C	S					R	1D
Shiele	B			M	MG				1D
DASL	B		S						1D
Zorro	B								1.5D
Symbolics									
MACS									
SPARCS									
ROSE - Compass									
PILGREM – Ic-Dc	B								
???? - Mentor									
LAS - Cadence									

Table 1 – Initial layout representation for different compactors

B = Bottom-Most Mask Level	MG = Mask Grid Positions
C = Cell Level	CG = Coarse Grid Positions
M = Mask Layout	VG = Virtual Grid Positions
S = Symbolic Layout	R = Relative Positions

Cells / Compactor			
Compactor	Area (microns)	CPU (seconds)	Memory
afa			
MACS	143 x 166 = 23768	9	1450K
SPARCS	157 x 180 = 28260	11	356K
Symbolics	160 x 189 = 30240	5^2	164K
Zorro	140.5 x 171=24025.5	430	647K
afavg			
MACS	142 x 145 = 20590	5	1390K
SPARCS	157 x 151 = 23707	8	372K
Symbolics	154 x 154 = 23716	5	160K
Zorro			
c132			
MACS	627 x 354 = 221958	41	300K
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

Table 2 – Compaction Benchmarks. The area is in microns square preceded by the x and y dimension. Run times for the compactors are reported for running on a VAX 8650. The memory usage is the peak memory needed for a given compactor to compact an example. This table was taken from [33].

5. Conclusions

Silicon area has always been a valuable resource, whichever technology is employed. Therefore, minimizing the silicon area demanded by designs is a very important task. One method of achieving this is through the use of layout compaction algorithms. These algorithms can be employed with symbolic layout editors, silicon compilation tools, module generators and technology conversion tools, among others.

This paper presented a survey and a taxonomy of layout compaction algorithms, focusing on their implementation characteristics, performance, advantages and drawbacks. Most algorithms are One-Dimensional (1-D) algorithms and compaction is usually performed in two phases – a compaction phase in the X direction and another compaction phase in the Y direction. Most versions of 2-D compaction algorithms are NP-hard.

Split grid algorithms present a performance that is comparable to constraint graph algorithms. However, to achieve good compaction and performance results, other techniques such as *wire jogging, wire minimization* and *hierarchical compaction* with *pitchmatching* have to be employed.

5. References

[01] Sherwani, N., "Algorithms for VLSI Physical Design Automation", second ed., Kluwer Acamedic Publishers, 1995.

[02] Lengauer, T., "Combinatorial Algorithms for Integrated Circuit Layout", John Wiley & Sons Inc., 1990.

[03] Preas, B.T., and Lorenzetti, M.J., "Physical Design Automation of Electronic Systems", Benjamin/Cummings Co., 1988.

[04] Weste, N.H.E. and Eshraghian, K.,"Principles of CMOS VLSI Design", McGraw Hill, 1987.

[05] Weste, N.H.E, "Virtual Grid Symbolic Layout", 18th Design Automation Conference, July 1981.

[06] Boyer, D.G. and Weste, N.H.E., "Virtual Grid Compaction Using the Most Recent Layers Algorithm", IEEE ICCAD Conference, September 1983.

[07] Boyer, D.G, "Split Grid Compaction for a Virtual Grid Symbolic Design System", Proc. of the IEEE International Conference on Computer-Aided Design, pp. 134-137, Nov. 1987.

[08] Nyland, L.S., Daniel, S.W., and Rodgers, D., "Improving Virtual-Grid Compaction Through Grouping", Proc. of the 24th IEEE/ACM Design Automation Conference, June, 1987.

[09] Akers, S.B., Geyer, J.M. and Roberts, D.L., "IC mask layout with a single conductor layer", Proc. of the 7th Design Automation Workshop", pp. 7-16, 1970.

[10] Hsueh, M.Y. and Pederson, D.O., "Computer-Aided Layout of LSI Circuit Building-Blocks", Ph.D. thesis, University of California at Berkeley, December, 1979.

[11] Malik, A.A., "An efficient algorithm for generation of constraint graph for compaction", Proc. of the IEEE International Conference on Computer-Aided Design, pp. 130-133, 1987.

[12] Scott, W.S. and Ousterhout, J.K., "Plowing: Interactive Stretching and Compaction in Magic", Proc. of the IEEE 21st Design Automation Conference, pp. 166-172, 1984.

[13] Deutsch, D.N., "Compacted Channel Routing", Proc. of the IEEE International Conference on Computer-Aided Design, pp. 223-225, Nov., 1985.

[14] Rivest,, R.L. and Fiduccia, C.M., "A greedy channel router", Proc. of the 19th Design Automation Conference, pp. 418-424, 1982.

[15] Crocker, W.H., Lo, C.Y. and Varadarahan, R., "MACS: a Module Assembly and Compaction System", Proc. of the IEEE International Conference on Computer Design, pp. 205-208, Oct., 1987.

[16] Burns, J. and Newton, A.R., "SPARCS: A New Constraint-Based IC Symbolic Layout Spacer", Proc. of the IEEE Custom Integrated Circuits Conference, pp. 534-539, May, 1986.

[17] Schiele, W.L., "Improved Compaction by Minimized Length of Wires", Proc. 20th ACM/IEEE Design Automation Conference, pp. 121-127, June 1983.

[18] Boyer, D.G., "Process Independent Constraint Graph Compaction", Proc. of the 29th ACM/IEEE Design Automation Conference, pp. 318-322, 1992.

[19] Boyer, D.G., "DASL Constraint Graph Compaction with Symbolic Tubs", ACM International Workshop on Layout Synthesis, May 1990.

[20] Lo, C-Y, "Automatic Tub Region Generation for Symbolic Layout Compaction", Proc. of the 26th ACM/IEEE Design Automation Conference, pp. 302-305, June 1989.

[21] Shin, H., Sangiovanni-Vincentelli, A., Sequin, C., "Two-Dimensional Compaction by Zone Refining", Proc. 23rd ACM/IEEE Design Automation Conference, pp. 115-122, June, 1986.

[22] Shin, H., Sangiovanni-Vincentelli, A., Sequin, C., "Two-Dimensional Module Compactor based on Zone Refining", Proc. of the IEEE International Conference on Computer Design, pp. 201-204, Oct., 1987.

[23] Wolf, W., Mathews, R., Newkirk, J., and Dutton, R., "Two-Dimensional Compaction Strategies", Proc. of the IEEE International Conference on Computer-Aided Design, pp. 90-91, Sept. 1983.

[24] Schlag, M., Liao, Y.Z., and Wong, C.K., "An algorithm for optimal two-dimensional compaction of VLSI layouts", Integration, vol.1, no. 3, pp. 179-209, September, 1983.

[25] Burns, J. and Newton, A. R. "Efficient Constraint Generation for Hierarchical Compaction", Proc. of the IEEE International Conference on Computer Design, pp 197-200, October 1987.

[26] Entenman, G. and Daniel, S.W., "A Fully Automatic Hierarchical Compactor", Proc. of the 22nd Design Automation Conference, pp 69-75, June 1985.

[27] Kingsley, C., "A Hierarchical, Error-Tolerant Compactor", Proc. 21st Design Automation Conference, pp 126-132, June 1984.

[28] Newton, A.R., "Symbolic Layout and Procedural Design", Design Systems for VLSI Circuits Logic Systesis and Silicon Compilation, G. DeMicheli, A. Sangiovanni-Vincentelli and P. Antognetti Ed., pp. 65-112, 1987.

[29] Reichelt, M.W., and Wolf, W.H., "An Improved Model for Hierarchical Constraint-Graph Compaction", Proc. of the IEEE International Conference on Computer-Aided Design, pp. 482-485, Sept. 1985.

[30] CADENCE. "Virtuoso layout synthesizer - LAS – user guide", CADENCE™ Version 4.2, October, 1999.

[31] COMPASS. "ROSE layout synthesizer – user guide", COMPASS™, 1999.

[32] IC/DC – Integrated Circuits Design Concepts company, "PILGREM layout system", http://www.ic-dc.com, 1999.

[33] Boyer, D.G., "Symbolic Layout Compaction Review", Proc. of the 25th ACM/IEEE Design Automation Conference, pp. 383-389, 1988.

[34] Ohtsuki, Tatuo, "Layout Design and Verification", Elsevier Science Publ. Co, 1986.