A Dynamic and Efficient Representation of Building-Block Layout

Wei-Ming Dai, Masao Sato, and Ernest S. Kuh

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, CA 94720

ABSTRACT

Dynamic layout representation is a key problem in developing a building block layout system. We have unified topological and geometrical representations, and developed efficient methods to update topological information after geometrical operations. The experimental results are very promising. This representation is the key for information flow in BEAR — a new building block layout system being developed at U. C. Berkeley.

1. Introduction

While standard cell and gate array layout systems are widely used, building block layout systems are still in the research stage. What are the key problems restricting the progress of the building block layout system?

Is routing the problem? Probably not, this area of layout has received more attention than any other. The sequential routing (maze-running and line-search), as the most classic approach, has been used in VLSI design for a long time [13]. Also the channel routing method has been extensively studied and many software packages are available [9], [13]. It is nice to have one less track in a routed channel, but it will not make or break the system.

Is placement the problem? Although automatic placement is a relatively new area, many approaches have been proposed [20]. For example, the min-cut placement method uses a good bi-partitioning heuristics and works reasonably well. In the worst case, we can place the blocks manually. So the placement is not a key problem from a system point of view.

In this paper, we will state one of the key problems in developing a building block layout system — dynamic layout representation. Although there are in the literature various layout systems, the subject of dynamic representation has not been addressed. Before we explain what we mean by dynamic, we discuss a global optimization step which, we believe, is a crucial part of the next generation of building block layout systems.

Placement defines the capacity of the routing area around the blocks and global routing defines the density (net assignment) of the routing area. Considering the detailed routing, the desirability of a given global routing on a given placement depends on the degree of the match of the capacity and the density. After placement and global routing, we can change the density by rerouting or change the capacity by global spacing (global compaction or decompaction). In order to achieve high density of the final layout, we iterate these two operations to obtain a satisfactory match of the capacity and the density of the routing area before the detailed routing.

Therefore, a dynamic layout representation should satisfy the following requirements:

1. Represent placement and global routing.
2. Transfer the global routing information to detailed routing efficiently.
3. Be easy to calculate the critical paths of the chip dimensions.
4. Be easy to update (placement and global routing) after global spacing or global re-routing.

2. A Survey of Layout Representations

The existing layout representations can be classified into two categories: topological models and geometrical models.

2.1. Topological models

A rectangular floor plan can be represented by a *rectangular dissection* D (Fig. 2.1). A rectangular dissection can be represented by a pair of mutually dual plane acyclic digraphs called *polar graphs*: $G_h = (V_h, E_h)$ and $G_v = (V_v, E_v)$, where V_h and V_v represent the set of horizontal lines and the set of vertical lines of D respectively. There is an arc (v_i, v_j) in E_h (or E_v) if and only if the line corresponding to v_i and the line corresponding to v_j are the top and bottom (or the left and right) sides of a rectangle respectively. Notice G_h and G_v contain one source and one sink (Fig. 2.2).

A rectangular dissection D can also be represented by an undirected plane graph, called a *floor plan graph*. $G = (V, E)$, where V represents the intersections of D and there is an edge (v_i, v_j) in E if and only if the intersection corresponding to v_i and the intersection corresponding to v_j are adjacent (Fig. 2.3).

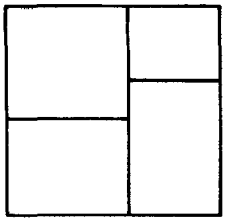


Fig. 2.1 Rectangular dissection D.

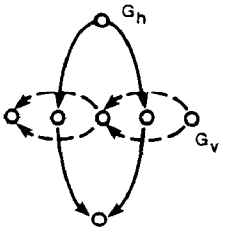


Fig. 2.2 Polar graph for D.

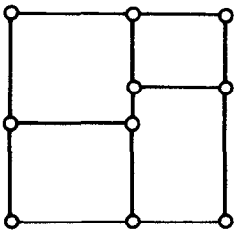


Fig. 2.3 Floor plan graph for D.

The concepts of rectangular dissections and polar graphs were introduced by Brooks et al. [3]. Ohtsuki et al. first applied these concepts to IC layout [12]. Zibert did extensive work on polar graphs for floor planning optimization, especially to hybrid layout [23]. The polar graphs were later used to represent the placement of building blocks [2], [8], [10], [11], [18], [19].

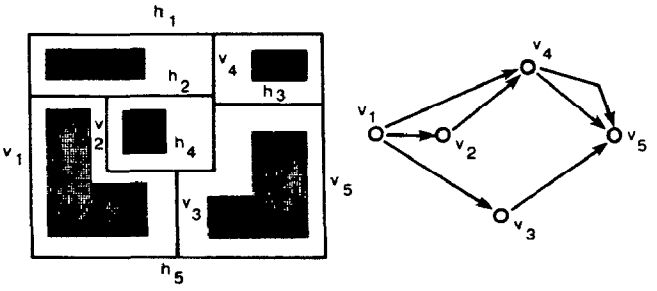
Otten formalized a structure restraint for rectangle dissections, a *slicing structure* (Fig. 2.4 (a)), and characterized the polar graphs for slicing structures in terms of series and parallel graphs [14], [15]. This restriction limits the topology of floor plan or placement, complicates the global spacing algorithms and wastes area when enforcing the slicing structure at the routing stage. A feasible routing order can be obtained for non-slicing structures (Fig. 2.4 (b)) by introducing L-shaped channels [6].

(a) slicing structure (b) non-slicing structure

Fig. 2.4 Floor plan topology.

Lauther refined the polar graphs to reflect the congestion of each line segment. These refined polar graphs (called placement graphs) are used in the placement improvement after global routing [10], [11]. In addition, floor plan graphs (called channel graphs) are used for global routing. But everything is restricted to slicing structures.

While Lauther assumed that all block shapes were rectangular, Preas extended the polar graphs (called channel position graphs) to represent the placement of a subset of rectilinear shaped blocks (shapes with arbitrary rectangles removed from zero to four of the corners). Like Lauther, he used a floor plan graph (called a channel intersection graph) (Fig. 2.5) [19].

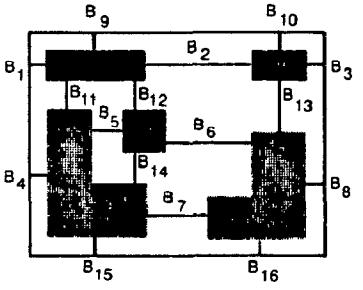


(a) channel intersection graph (b) horizontal channel position graph

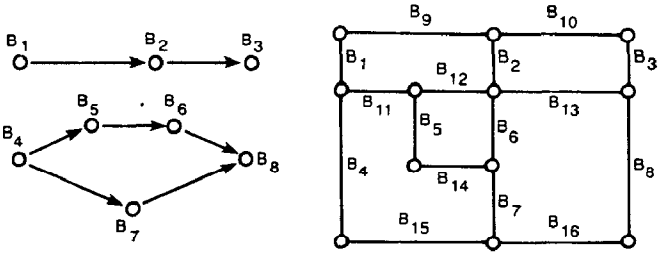
Fig. 2.5 Preas' graphs.

Even limiting block shapes and placement topology as these authors did, keeping the polar graphs and the floor plan graph consistent is not simple. Furthermore, none of them raised the issue of updating global routing information when the topology of the placement is changed.

The BBL system developed at U. C. Berkeley [4] uses the concept of "*bottlenecks*". A bottleneck exists between two blocks or between a block and the chip boundary if there is no other block in between (Fig. 2.6(a)). Bottlenecks identify areas where congestion of routing is most likely to occur and these are the targets for global spacing and global re-routing. In addition to the *bottleneck graphs* (Fig. 2.6(b)), floor plan graphs (in BBL called global routing graphs) are used (Fig. 2.6(c)). There is no block shape limitation or placement topology limitation in BBL. However, when moving blocks, it is not apparent that the correct updating of both graphs can be guaranteed.



(a) bottlenecks B_1, B_2, ..., B_{16}



(b) horizontal bottleneck graph (c) global routing graph

Fig. 2.6 Graphs in the BBL system.

2.2. Geometrical models

Instead of representing placement and global routing information in topological models such as the polar graphs and the floor plan graphs discussed above, we can use geometrical models. Soukup et al. partitioned the empty space into numerous small rectangles by extending all block boundaries until they intersected with another block or the chip boundary (Fig. 2.7) [21]. Persky introduced a heuristic approach to minimize the number of rectangles and to obtain favorable aspect ratios for the rectangles: draw a single horizontal or vertical line, whichever is shorter, from each corner of every block until it intersects another block, a previously drawn line, or the chip boundary (Fig. 2.8) [17]. This algorithm was first implemented in the LTX2 system [5]. While Soukup's model was developed for a sequential routing scheme, Persky's model was biased in favor of channel routing scheme. Both representations are static, that is, difficult to update after block movement.

An interesting geometrical model has been proposed by Wiesel et al. [22]. Each routing layer was divided into a different set of rectangles. On the horizontal layer, the routing area was divided into maximal horizontal strips. On the vertical layer, it was divided into maximal vertical strips (Fig. 2.9). As will be seen later, we make use of these two sets of rectangles in different ways without layer assumption.

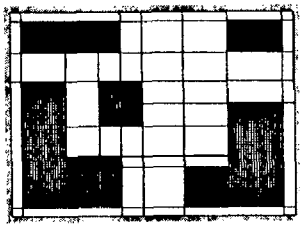


Fig. 2.7 Soukup's model.

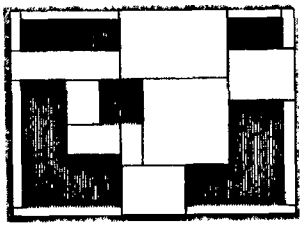
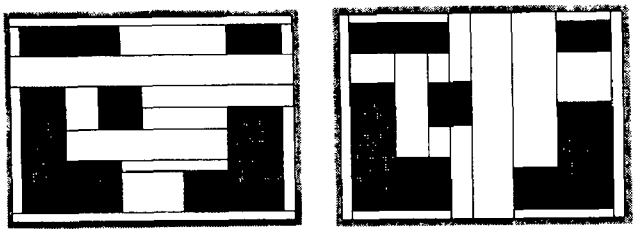


Fig. 2.8 Persky's model.



(a) horizontal layer (b) vertical layer

Fig. 2.9 Wiesel's model.

3. Unified Topological and Geometrical Representation

The difficulty of the layout representation arises when updating topological information (eg. global routing information) after geometrical operations. Our main contribution in this area is to unify topological and geometrical representations to overcome these problems.

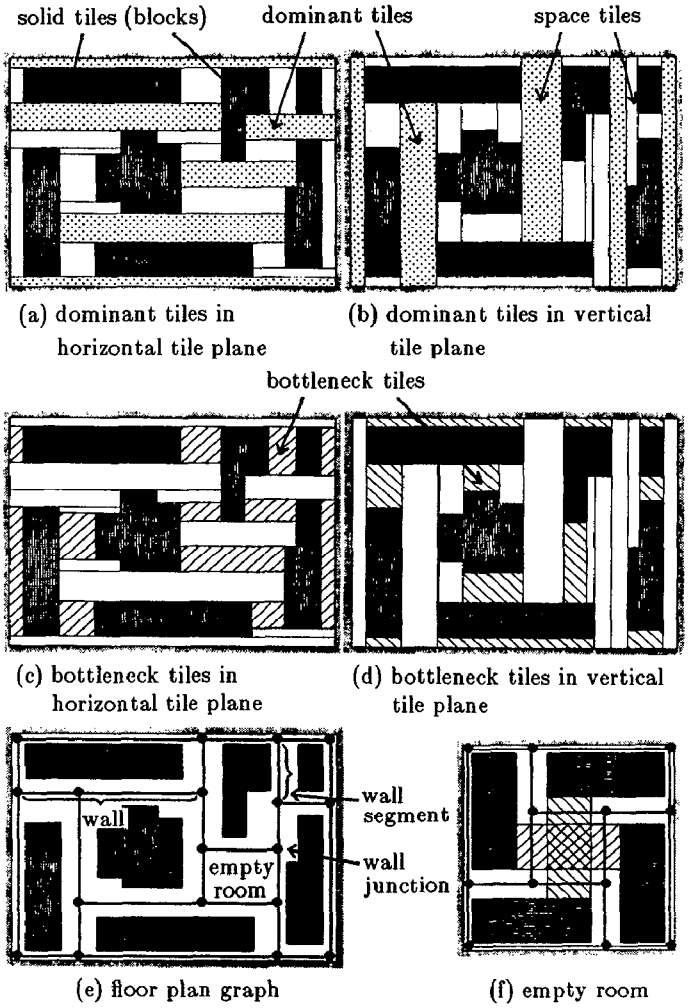
3.1. Geometrical representation — tile planes

The entire area of a layout is covered with rectangles referred to as *tiles*. There are two kinds of tiles: *solid tiles*, which represent blocks, and *space tiles*, which represent empty space for routing between the blocks.

Given a placement of rectilinear-shaped and arbitrary-sized building blocks, we define two tile planes: the *horizontal tile plane*, where all space tiles are maximal horizontal strips, and the *vertical tile plane*, where all space tiles are maximal vertical strips (Fig. 3.1 (a) and (b)). The tile planes may be implemented using the corner stitching data structure [16].

In a tile plane, each space tile has four edges. Two of them are called the *spans* of the tile (completely covered by solid tiles); the other two, the *sides* of the tile. The size of the spans and the sides of a space tile are referred to as the *widths* and the *lengths* respectively (Fig. 3.2 (a)).

We define two particular classes of space tiles: dominant tiles and bottleneck tiles. These concepts play key roles in our unified representation. A space tile is called *dominant* if none of its sides are covered by the side of its adjacent space tiles (Fig. 3.1(a) and (b)). A space tile is called *bottleneck* if both sides are covered by the sides of its adjacent space tiles (Fig. 3.1(c) and (d)). Note that dominant tiles and bottleneck tiles in a tile plane are mutually disjoint and there exist tiles which are neither dominant nor bottleneck.



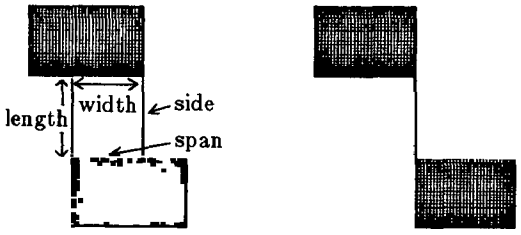
(a) dominant tiles in horizontal tile plane (b) dominant tiles in vertical tile plane

(c) bottleneck tiles in horizontal tile plane (d) bottleneck tiles in vertical tile plane

(e) floor plan graph (f) empty room

Fig. 3.1 Tile planes and floor plan graph.

If the width of a bottleneck tile is equal to zero, the tile is regarded as a *bottleneck line*. More precisely, a *bottleneck line* is a line segment which connects the left (top) edge of one solid tile and the right (bottom) edge of another solid tile on a vertical (horizontal) straight line without intersecting any edge of solid tiles (Fig. 3.2 (b)).



(a) (b)

Fig. 3.2 (a) Bottleneck tile and (b) bottleneck line.

3.2. Topological representation — floor plan graphs

Using the dominant tile concept, a *floor plan graph* can be efficiently derived from a pair of horizontal and vertical tile planes. Corresponding to each horizontal or vertical dominant tile, we draw a *wall* for the floor plan graph. At each intersection of a horizontal and a vertical dominant tile, we draw a *wall junction* connecting the tiles' walls. We call a portion of a wall between two adjacent junctions a *wall segment*, and a region bounded by walls but containing no walls a *room* (Fig. 3.1(e)). Most rooms contain solid tiles. Those rooms that do not are called *empty rooms* (Fig. 3.1(f)).

As we mentioned before, similar graphs have been used in many papers, but to the best of our knowledge, neither precise definitions nor construction algorithms for such graphs have been explicitly given in previous publications except [6]. Even the examples they have illustrated were limited to a special class of topologies — without empty rooms.

3.3. The correspondences between tile planes and floor plan graph

By the definition of tile plans and floor plan graphs, the following theorem is obvious.

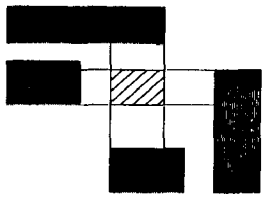
Theorem 1: There is one-to-one correspondence between a dominant tile in a tile plane and a wall in the corresponding floor plan graph.

Later we will show how to dynamically update the floor plan graph using this correspondence.

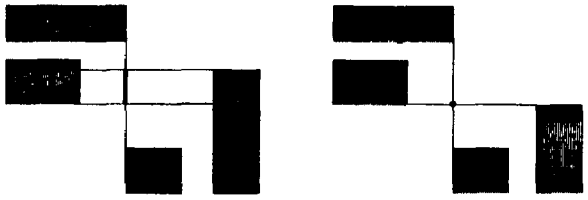
Sometimes a bottleneck tile (or line) in the horizontal tile plane and a bottleneck tile (or line) in the vertical tile plane may intersect. We call such an intersection region a *bottleneck intersection region*. Depending on whether bottleneck tiles or lines intersect, there are three kinds of bottleneck intersection regions: *bottleneck intersection rectangles, lines, and points* (Fig. 3.3).

Now we have the following theorem for characterizing empty rooms.

Theorem 2: There is one-to-one correspondence between a bottleneck intersection region in a tile plane and an empty room in the corresponding floor plan graph (see Fig. 3.1).



(a) bottleneck intersection rectangle



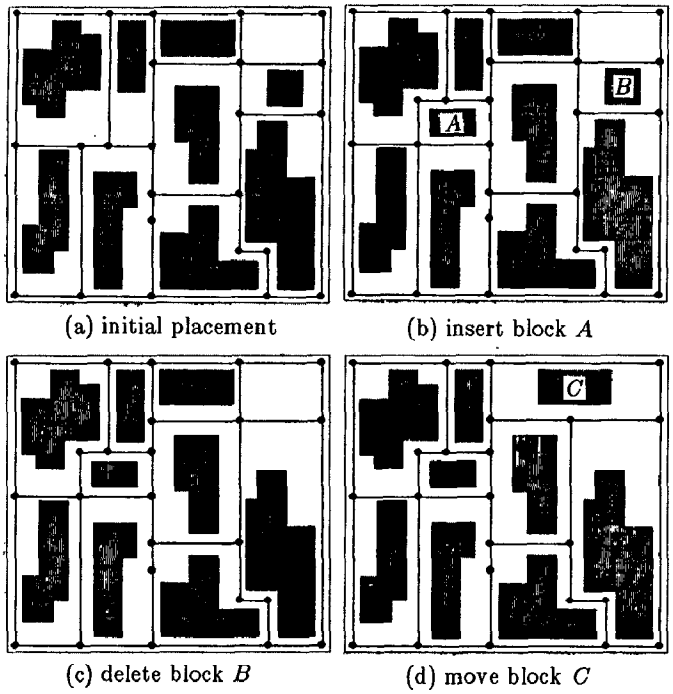
(b) bottleneck intersection line (c) bottleneck intersection point

Fig. 3.3 Bottleneck intersection regions.

In the following, we state yet another correspondence between tile planes and the floor plan graph.

Theorem 3: Corresponding to each bottleneck tile or line in a tile plane, there is only one wall segment in the corresponding floor plan graph; the correspondence is one-to-one if and only if there are no empty rooms.

The above three theorems characterize our unified representation. As an example of the applications of the unified representation, in Figure 3.4, we show how the floor plan graph gets updated after we insert, delete, and move blocks. The response time of these operations is very promising.

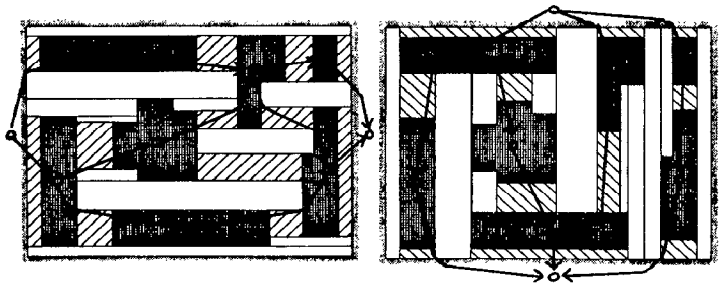


(a) initial placement (b) insert block A

(c) delete block B (d) move block C

Fig. 3.4 Interactive placement (illustrating dynamic updating of the floor plan graph).

A pair of *block adjacency graphs* need not be constructed explicitly since the adjacency of the blocks can be obtained efficiently via bottleneck tiles (Fig. 3.5). The graphs play a similar role as polar graphs; for example, they can be used to calculate the critical paths of the chip dimensions.



(a) (b)

Fig. 3.5 (a) Horizontal and (b) vertical block adjacency graph.

3.4. Pseudo pins and local nets

Where do we store global routing information? It is too much to record the topological paths of the nets in all the tiles, On the other hand, if we attach the information to wall segments or wall junctions of the floor plan graph, it is hard to update when the placement topology is changed.

Since the bottleneck tiles correspond to the wall segments in the floor plan graph, we store the global routing information on the bottleneck tiles. In this way, we have the advantages of both topological and geometrical representation. Specially, as will be seen in the later sections, this representation makes the job of dynamic updating of global routing easier. Either the set of bottleneck tiles on the horizontal tile plane or that on the vertical tile plane is sufficient for specifying global routing information.

For each *bottleneck side*, a side of a bottleneck tile, we record a list of net crossings called *pseudo pins*. From the point of view of each tile, two pins or pseudo pins may not be connected even though they belong to the same net globally. So we need the notion of *local nets*. Each pseudo pin on the side of a tile belongs to three nets, namely, a *global net*, an *internal net* (the net inside the tile), and an *external net* (the net outside the tile) (Fig. 3.6).

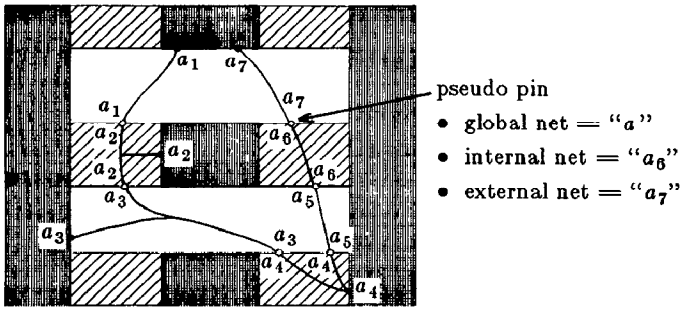


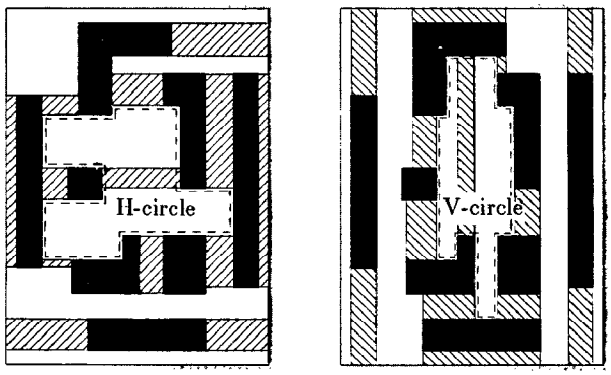
Fig. 3.6 Pseudo pins and local nets.

4. Dynamic Representation of Global Routing

4.1. Circles and chords

Since the global routing information is topological, we introduce the concepts of circles and chords on tile planes. A *circle* is a closure of a region surrounded by a set of block boundaries and a set of horizontal and/or vertical bottleneck sides on the tile planes. The circles formed by block boundaries and horizontal bottleneck sides are named *H-circles* (Fig. 4.1(a)), and those formed by block boundaries and vertical bottleneck sides, *V-circles* (Fig. 4.1(b)). A line drawn from one point to another point on a circle is referred to as a *chord* of the circle. For example, a bottleneck side is a chord.

The pins on the block boundaries and the pseudo pins on the bottleneck sides are the global routing information attached to the circle. When we insert chords in a circle, we partition the circle or subdivide the nets; when we delete the chords, we merge the circle or unify the local nets.



(a) (b)

Fig. 4.1 Circles.

By *circle partition*, we mean partitioning a circle into n circles by inserting $n-1$ chords of the circle. Given a circle C, as the result of inserting k non-intersecting chords, $l_1, l_2, \ldots, l_k$, C is partitioned into $k+1$ circles $C_1, C_2, \ldots, C_{k+1}$ (Fig. 4.2). When we insert a chord in a circle, for simplicity, we assume each local net crosses the chord only once. Under this assumption, the pseudo pins on the chords are uniquely determined by the pins and pseudo pins on C and the position of the end points of the chords. The pseudo pins on the chords can be created efficiently on tile planes by a modified plane-sweep method.

Circle unification is the inverse of circle partition. We unify n circles into one circle by deleting $n-1$ chords in the circle. For each pair of adjacent circles, we delete the chord in between, unifying the local nets of its two sides (Fig. 4.2).

The circle partition and circle unification are the basic operations in the global routing update process. On the tile planes implemented by corner stitching data structures, these two operations can be performed in time and space linear to the number of chords in the circle and number of pins and pseudo pins on the circle.

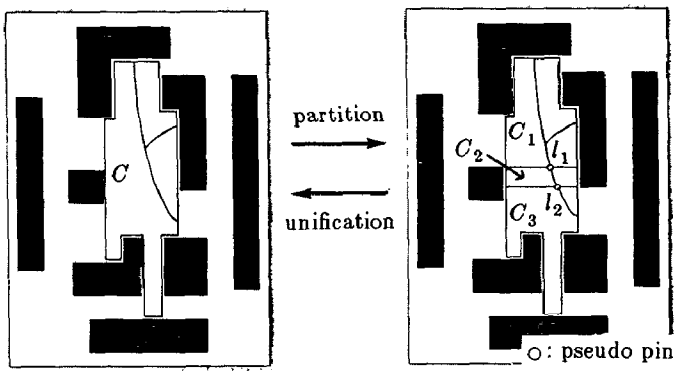


Fig. 4.2 Circle partition and circle unification.

4.2. Local updating method

After moving blocks, only some of the bottleneck tiles may be affected. We call such bottleneck tiles *moving effective bottleneck tiles.*

4.2.1. Moving effective bottleneck tiles

A bottleneck tile is said to be moving effective if its size or its position relative to the attached blocks was changed or the global routing information on their sides was altered.

To begin with, let us consider the simplest case: move a single block.

After moving a block horizontally (vertically), the effect on the horizontal (vertical) tile plane is simple: the horizontal (vertical) bottleneck tiles attached to the block will be either *compacted* (their lengths contracted), or *stretched* (their lengths expanded) (Fig. 4.3 (a)). In this case, the global routing information attached to these bottleneck tiles remains the same.

However, when the widths of bottleneck tiles are changed, updating is not trivial. If the widths of the bottleneck tiles are contracted, we call them *narrowed bottleneck tiles*; if the widths are expanded, *widened bottleneck tiles* (Fig. 4.3 (b)). When a block is moved in an arbitrary direction, both widths and lengths of the bottleneck tiles attached to the block may change (Fig. 4.3 (c)). We call those bottleneck tiles whose sizes did not change but whose positions relative to the attached blocks were changed, *slid bottleneck tiles* (Fig. 4.3 (d)).

Furthermore, if the relative positions of blocks do change, the bottleneck tiles attached to the blocks will be *destroyed* at the origin from which the block moves and will be *created* at the destination to which the block moves (Fig. 4.3 (e)). In particular, if one bottleneck tile was created as the result of destroying two other bottleneck tiles in the same place, we call it a *merged bottleneck tile*; On the other hand, if two bottleneck tiles were created as the result of destroying one other bottleneck tile in the same place, we call them *split bottleneck tiles* (Fig. 4.3 (f)).

Even though some bottleneck tiles are not attached to the moving blocks, they may still be moving effective. The bottleneck tiles which are passed by a moving block are called *passed bottleneck tiles* (Fig. 4.3 (g)).

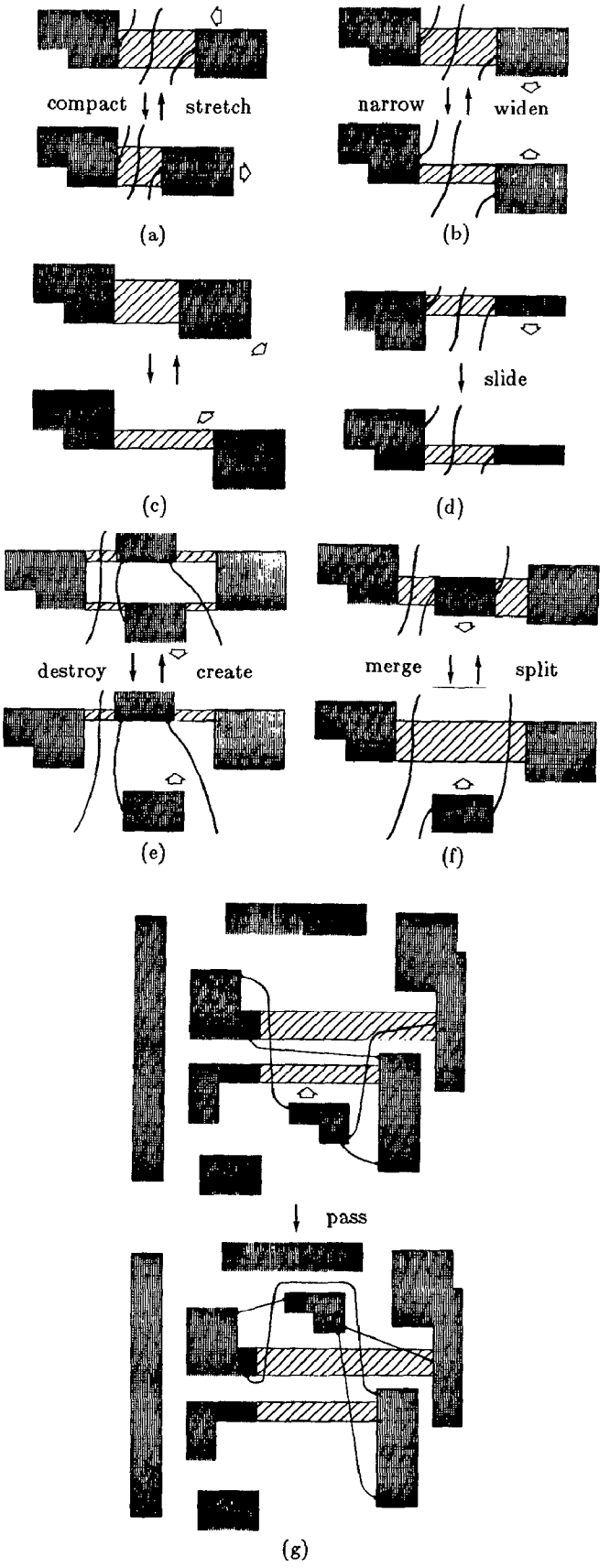


Fig. 4.3 Moving effective bottleneck tiles.

4.2.2. Tile-wise updating

It can be verified that the different cases mentioned above include all cases of moving effective bottleneck tiles. So it is natural to consider a tile-wise updating strategy after a block move.

By updating global routing, we mean validating the pseudo pin information on the bottleneck sides. Thinking of bottleneck sides as chords crossing some circles, updating is nothing more than inserting the chords of the circles, or partitioning the circles.

Instead of inserting chords after the move, we could insert *equivalent chords* before the move. By equivalent, we mean the global routing information we try to obtain for the chords (or the bottleneck sides) after the move can be mapped from these corresponding chords before the move (Fig. 4.4).

If the equivalent chords are inside a bottleneck tile which forms a circle for the chords, the updating is simple:

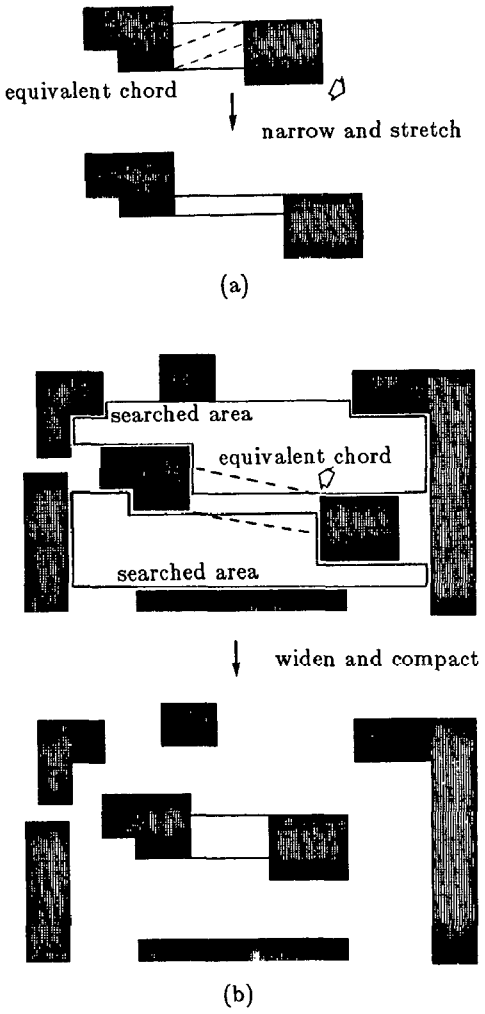


Fig. 4.4 Tile-wise updating after moving a single block.

just look at the information on the sides and spans of the bottleneck tile. Updating narrowed bottleneck tiles is an example of such a case (Fig. 4.4 (a)). Otherwise, we need to find a minimal circle which is crossed by the chords to perform the circle partition operation (Fig. 4.4 (b)). For updating most moving effective bottleneck tiles, such a local search is required.

Since the local search is can not to be avoided, we may prefer to update region-wise instead of tile-wise as discussed above.

4.2.3. Region-wise updating

Updating after a single block move is the primitive operation which will be used later on in more complicated situations.

4.2.3.1. Updating after moving a single block

For the purpose of updating global routing information, we currently assume a block can be moved in any direction without overlapping other blocks. So for a given block move, there is one minimal circle on a tile plane which contains all moving effective bottleneck sides. We call this circle *moving effective circle*. On a tile plane, given the origin and destination positions of the moving block, the moving effective circle can be found efficiently by local search (again time and space linear to the number of tiles inside the circle).

Let C denote the moving effective circle. Let p denote the left end point of the upmost line segment of the moving block boundary and q the right end point of the downmost line segment. Let p_1 and q_1 denote these points before the move and p_2 and q_2 after the move (Fig. 4.5 (a)).

In the following, we only describe the updating on the horizontal tile plane. Updating on the vertical tile plane can be done in a similar way. Notice that both tile planes are updated independently.

The moving effective circle on the horizontal tile plane is obviously an H-circle.

Without loss of generality, we assume the block moves up. Extending a line through p_2 upward we will intersect the circle at a point, say s. Similarly, extending a line through q_1 downward we will intersect the circle at a point, say t. Notice that the global routing information on $s-p_1$ and $s-p_2$ are equivalent, as are q_1-t and q_2-t (Fig. 4.5 (a)).

The following are used to update global routing information in the circle, C. Before the move, we obtain the information on chords $s-p_1$ and q_1-t by circle unification and circle partition. Then we move the block and map the information from $s-p_1$ and q_1-t to $s-p_2$ and q_2-t respectively. The chord $s-p_2-q_2-t$ partitions C into C_l on its left and C_r on its right. Next we update the bottleneck sides in C_l by circle partition (Fig. 4.5 (b)). The bottleneck sides to be updated are nothing more than the chords of C_l. Let r denote the left end point of the lowest bottleneck side in C_l. Finally, we unify the circle, C', $s-p_2-q_2-r-t-s$, and update the bottleneck sides inside C' (Fig. 4.5 (c)).

4.2.3.2. Global spacing

By *global spacing*, we mean global compaction and decompaction or placement modifications performed after global routing to obtain a better match of the placement and the topological routing to minimize the final layout area. Global spacing is much more effective for optimizing the final layout compared with the local optimization of detailed routing. It is also efficient for achieving a better match between the placement and the routing since no detailed wiring is presented.

In contrast to the constraint-graph approach, the *ridge spacing method*, operates on the tile planes, and is composed of small steps which iteratively partition the layout into two pieces, and performs cutting or expanding only on the spaces which lie in between the two pieces. At each step, the topology of the placement is preserved as much as possible.

The basic idea of ridge spacing was first proposed by Akers et al. in 1970 [1]. For a brief survey of this method, see [7]. In [7], the ridge spacing problem has been precisely formulated as the bottleneck path problem based on the concepts of tile planes and space tile adjacency graphs. While previous methods require $O(n^2)$ time to find a monotonic ridge (without optimization), we can find an optimal monotonic ridge in $O(n)$ time. Furthermore, we have generalized the spacing ridges to be non-monotonic , and developed $O(n \log n)$ time algorithms for finding an optimal one.

4.2.3.3. Updating after spacing a single ridge

The problem of updating global routing after spacing a single ridge can be translated into the problem of updating after moving a single block. Among those blocks to be moved, some are *moving critical* in the sense that they will result in moving effective bottleneck tiles (Fig. 4.6 (a)). For a given compaction or decompaction ridge, the moving critical blocks can be efficiently detected on tile planes. When we compact a ridge, we first move these moving critical blocks one by one and update global routing locally using the methods described earlier (Fig. 4.6 (b)). We then move the rest simply by changing the coordinates (Fig. 4.6 (c)). Decompacting a ridge is done in the reverse order: reposition moving critical blocks after we repositioning other blocks.

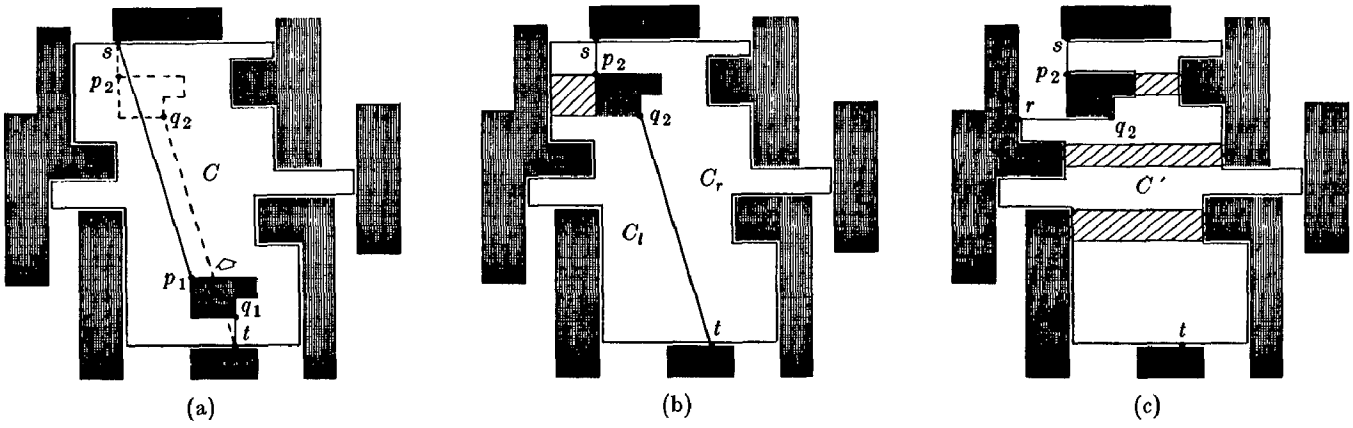


Fig. 4.5 Region-wise updating after moving a single block.

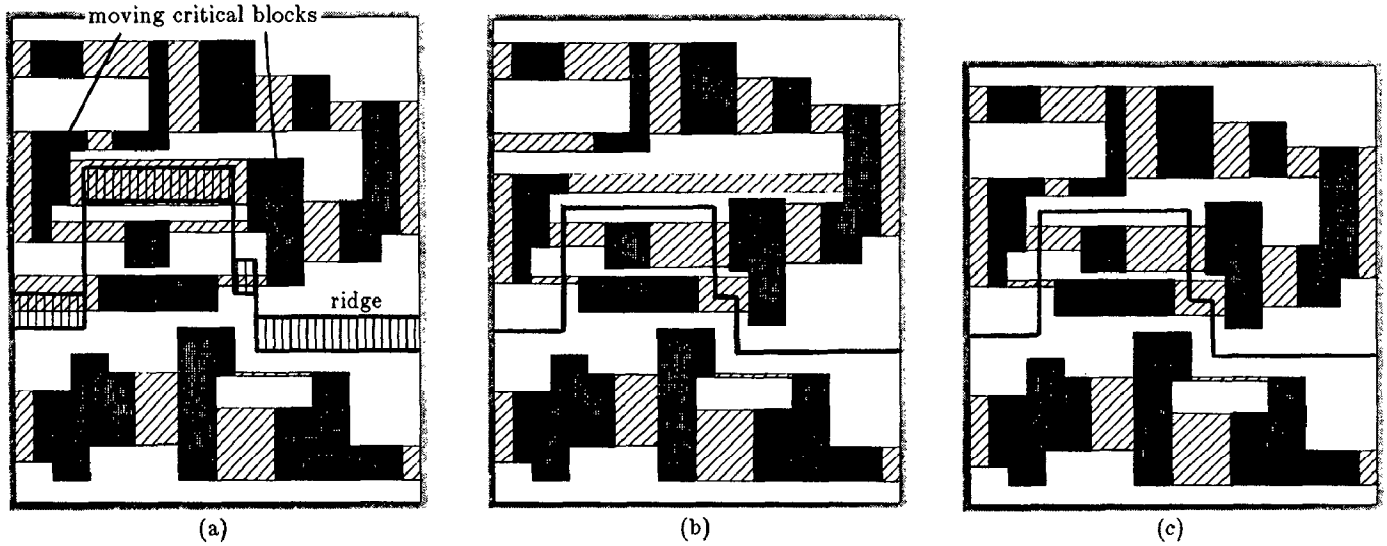


Fig. 4.6 Compaction (a)-(b)-(c) and decompaction (c)-(b)-(a).

4.3. Global mapping method

Since a horizontal tile plane or a vertical tile plane alone holds complete global routing information, we may update one of the tile planes when blocks are moved or nets are re-routed. Later when needed, we translate the whole global routing information on one tile plane to the other tile plane. This translation process is referred to as *global mapping*. Global mapping is especially useful when updating global routing after spacing a sequence of complicated ridges in one direction. Note that there is no need to update the horizontal (vertical) tile plane after spacing a horizontal (vertical) ridge because all the moving effective bottleneck tiles are either compacted (when doing compaction) or stretched (when doing decompaction).

4.4. Geometrical routing region definition

Besides updating global routing, our layout representation supports a wide range of operations throughout the entire layout process. As we mentioned before, one of the requirements for the data representation is to pass global routing information to local routers: a channel router for inter-cell routing and a sequential router for the intra-cell (or over-the-cell) routing.

Passing global routing information to the sequential router is trivial. We transfer global routing information to channels by creating the floating pins on the open sides of channels. This is done by using the global routing information attached to the bottleneck sides around the open sides of the channels. Any one of the tile planes is sufficient for such operation.

5. Concluding Remarks

Not long ago, only a few places had people working on building block layout systems, mainly the Japanese industries and the U. S. universities. Since ASIC (Application Specific Integrated Circuit) is currently one of the main challenges in IC design, many building block layout systems are now under development. If one does not work out dynamic layout representation properly, no matter how powerful the placement and routing techniques used, sooner or later one will face the problem that the placement and routing components can not be suitably integrated, thus the system will not always work. This is the lesson we have learned from our previous experience with the first generation of building block layout system (BBL) developed at U. C. Berkeley [4].

The unified, dynamic, and efficient layout representation discussed in this paper together with other significant results on placement and routing motivated the development of a new building-block layout system from scratch. This new system named BEAR (Building-block Environment Allocation and Routing system) is being developed using the C language for color and black-and-white displays that support the X window manager, which runs under 4.3 BSD UNIX. EDIF will be our primary input and output (also CIF). The preliminary results, especially the interactive features, indicate the dynamic layout representation is very promising.

Acknowledgment

George Carvalho has contributed to the implementation of the layout representation. This research was supported by the Semiconductor Research Corporation under the Grant SRC-82-11-008 and National Science Foundation under the Grant ECS-85-06901.

References

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

[2] K. D. Brinkmann and D. A. Mlynski, "Computer aided chip minimization for IC-layout," in *Proc. of Int. Symp. on Circuits and Systems*, pp. 650-653, 1976.

[3] R. L. Brooks, C. A. B. Smith, A. H. Stone, and W. T. Tutte, "The dissection of rectangles into squares," *Duke Math. J.*, Vol. 7, pp. 312-340, 1940.

[4] N. P. Chen, C. P. Hsu, and E. S. Kuh, "The Berkeley building-block (BBL) layout system for VLSI design," in *Dig. Tech. Papers, IEEE Int. Conf. on Computer-Aided Design*, pp. 40-41, 1983.

[5] B. W. Colbry and J. Soukup, "Layout aspects of the VLSI microprocessor design," in *Proc. of 1982 IEEE Int. Symp. on Circuits and Systems*, pp. 1214-1228, 1982.

[6] W. M. Dai, T. Asano, E. S. Kuh, "Routing region definition and ordering scheme for building-block layout," *IEEE Trans. on Computer-Aided Design of ICs and Syst.*, Vol. CAD-4, No. 3, 1985.

[7] W. M. Dai and E. S. Kuh, "Global spacing of building-block layout," to appear in *Proc. of VLSI 1987*, 1987.

[8] K. Kani, H. Kawanishi, and A. Kishimoto, "ROBIN: A building block LSI routing program," in *Proc. of Int. Symp. on Circuits and Systems*, pp. 658-660, 1976.

[9] E. S. Kuh, Ed., "The special issue on routing and microelectronics," *IEEE Trans. on Computer-Aided Design of ICs and Syst.*, Vol. CAD-2, No. 4, Oct. 1983.

[10] U. P. Lauther, "A min-cut placement algorithm for general cell assemblies based on a graph representation, in *Proc. of 16th Design Automation Conf.*, pp. 1-10, 1979.

[11] U. P. Lauther, "Channel routing in a general cell environment," in *Proc. of VLSI 1985*, 1985.

[12] T. Ohtsuki, N. Sugiyama, and H. Kawanishi, "An optimization technique for integrated circuit layout design," in *Proc. ICCST-Kyoto*, pp. 67-68, Sept. 1970.

[13] T. Ohtsuki, ed., *Layout design and verification*, Advances in CAD for VLSI, Vol. 4, North-Holland, Amsterdam, 1986.

[14] R. H. J. M. Otten, "Complexity and diversity in IC layout design," in *Proc. of Int. Conf. on Circuits and Computers*, pp. 764-767, 1980.

[15] R. H. J. M. Otten, *Layout Structures*, IBM Research Report RC9657, Thomas J. Watson Research Center, Yorktown Heights, N. Y., 1982.

[16] J. K. Ousterhout, "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools," *IEEE Trans. on Computer-Aided Design of ICs and Syst.*, Vol. CAD-3, No. 1, 1984.

[17] G. Persky, C. Enger, and D. M. Selove, "The Hughes automated layout system – automated LSI/VLSI layout based on channel routing," in *Proc. of 18th Design Automation Conf.*, pp. 22-28, 1981.

[18] B. T. Preas and C. W. Gwyn, "Methods for hierarchical automatic layout of custom LSI circuit masks," in *Proc. of 15th Design Automation Conf.*, pp. 206-212, 1978.

[19] B. T. Preas and C. S. Chow, "Placement and routing algorithms for topological integrated circuit layout," in *Proc. of Int. Symp. on Circuits and Systems*, pp. 17-20, 1985.

[20] B. T. Preas and P. G. Karger, "Automatic placement, A review of current techniques," in *Proc. of 23rd Design Automation Conf.*, pp. 622-629, 1986.

[21] J. Soukup and J. Royle, "Cell map representation for hierarchical layout," in *Proc. of 17th Design Automation Conf.*, pp. 591-594.

[22] M. Wiesel and D. A. Mlynski, "An efficient channel model for building block LSI," in *Proc. of Int. Symp. on Circuits and Systems*, pp. 118-121, 1981.

[23] K. Zibert and R. Saal, "On computer aided hybrid circuit layout," in *Proc. Int. Symp. on Circuits and Systems*, pp. 314-318, 1974.