Corner Stitching: A Data Structuring Technique for VLSI Layout Tools

John K. Ousterhout
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
415-642-0865
ARPAnet address: ousterhout@berkeley

Abstract

Corner stitching is a technique for representing rectangular two-dimensional objects. It appears to be especially well suited for interactive editing systems for VLSI layouts. The data structure has two important features: first, empty space is represented explicitly; and second, rectangular areas are stitched together at their corners like a patchwork quilt. This organization results in fast algorithms (linear time or better) for searching, creation, deletion, stretching, and compaction. The algorithms are presented under a simplified model of VLSI circuits, and the storage requirements of the structure are discussed. Measurements indicate that corner stitching requires approximately three times as much memory space as the simplest possible representation.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3803, monitored by the Naval Electronic System Command under Contract No. N00039-81-K-0251

1. Introduction

Interactive layout tools for integrated circuits place special burdens on their internal data structures. The data structures must be able to deal with large amounts of information (one-half million or more geometrical elements in current layouts [6]) while providing instantaneous response to the designer. As the complexity of designs increases, tools must give more and more powerful assistance to the designer in such areas as routing and validation. To support these intelligent tools, the underlying data structures must provide fast geometrical operations, such as locating neighbors for stretching and compaction, and locating empty space for routing. The data structures must also permit fast incremental modification so that they can be used in interactive systems.

Corner stitching is a data structure that meets these needs. It is limited to designs with Manhattan features (horizontal and vertical edges only) but within that framework it provides a variety of powerful operations, such as neighbor-finding, stretching, compaction, and channel-finding. The algorithms for the operations depend only on *local* information (the objects in the immediate vicinity of the operation). Their running times are generally linear in the number of nearby objects; in pathological cases (which are extremely unlikely for actual layouts) the running times may be linear in the overall design size. Corner stitching is especially effective when the objects are relatively uniform in size, as is the case for low-level mask features. It also works well when there is variation in feature size, e.g. hierarchical layouts containing large subcells and small wires to connect them.

Corner stitching permits modifications to the database to be made quickly, since only local information is used in making the updates. Most existing systems that provide powerful operations such as routing and compaction do not provide inexpensive updates: small changes to the database can result in large amounts of recomputation. Corner stitching's combination of powerful operations and easy updates means that many powerful tools previously available only in "batch" mode can now be embedded in interactive systems.

2. A Simplified Model of VLSI Layouts

A VLSI layout is normally specified as a hierarchical collection of cells, where each cell contains geometrical shapes on several mask layers and pointers to subcells. As a convenience in presenting the data structure and algorithms, a simplified model will be used in this paper. Only a single mask layer will be considered, and hierarchy will not be considered. For this paper, I define a "circuit" to be a collection of rectangles. There is a single design rule in the model: rectangles may not overlap. The simplified model makes it easier to present the data structure and algorithms. Section 8 discusses how the simplified model might be generalized to handle real VLSI layouts.

3. Existing Mechanisms

The simplest possible technique for representing rectangles is just to keep all of them in one large list. This technique is used in the Caesar system [5]: each cell is represented by a list of rectangles for each of the mask layers. Even though operations such as neighbor-finding require entire lists to be searched, the structure works well in Caesar for two reasons. First, large layouts are broken down hierarchically into many small cells; only the top-most cells in the hierarchy ever contain more than a few hundred rectangles or a few children [6]. Second, Caesar provides only very simple operations like painting and erasing. More complex functions such as design rule checking and compaction could not be implemented efficiently using rectangle lists.

The most popular data structures for VLSI are based on *bins* [1]. In bin-based systems, an imaginary square grid divides the area of the circuit into bins, as in Figure 1. All of the rectangles intersecting a particular bin are linked together, and a two-dimensional array is used to locate the lists for different bins. The rectangles in a given area can be located quickly by indexing into the array and searching the (short) lists of relevant bins. The bin size is chosen as a tradeoff between time and space: as bins get larger, it takes longer to search the lists in each bin; as bins get smaller, rectangles begin to overlap several bins and hence occupy space on several lists.

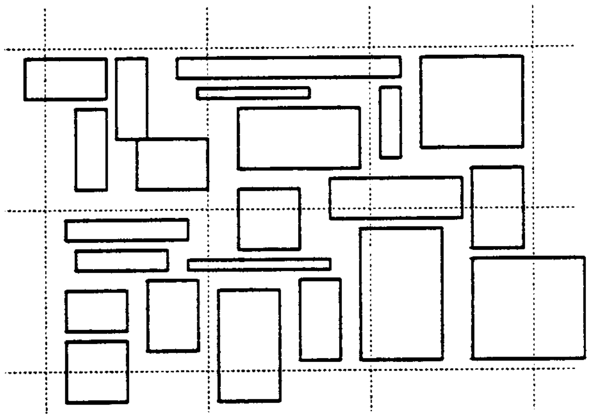

Figure 1. In bin-based data structures, the circuit is divided by an imaginary grid, and all the rectangles intersecting a subarea are linked together.

Bin structures are most effective when rectangles have nearly uniform size and spatial distributions; they suffer from space and/or time inefficiencies when these conditions are not met. A pathological case is a cell with a few large child cells and many small rectangles to interconnect them: if bins are small, then there will be many empty bins in the large areas of the subcells, resulting in wasted space for the bins; if bins are large, then the bins in the wiring area will have many rectangles, resulting in slow searches. Hierarchical bin structures [3] have recently been proposed as a solution to the problems of non-uniformity. Although bins provide for a quick location of all objects in an area, they do not directly embody the notion of *nearness*. To find the nearest object to a given one, it is necessary to search adjacent bins, working out from the object in a spiral fashion. Furthermore, bin structures do not indicate which areas of the chip are empty; empty areas must be reconstructed by scanning the bins. The need constantly to scan bins to recreate information makes bin structures clumsy at best, and inefficient at worst, especially for operations such as compaction and stretching.

A third class of data structures is based on neighbor pointers. In this technique, each rectangle contains pointers to rectangles that are adjacent

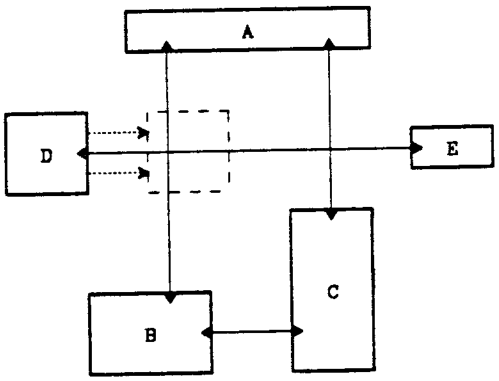

Figure 2. Neighbor pointers can be used to indicate horizontal or vertical adjacency. However, if D is moved right, it is hard to update the vertical pointers without scanning the entire database.

to it in x and y. See Figure 2. Neighbor pointers are a popular data structure for compaction programs such as Cabbage [2], since they provide information about relationships between objects. For example, a simple graph traversal can be used to determine the minimum feasible width of a cell.

Neighbor pointers have two drawbacks. First, modifications to the structure generally require all the pointers to be recomputed. For example, if an object is moved horizontally as in Figure 2, vertical pointers may be invalidated. There is no simple way to correct the vertical pointers short of scanning the entire database. The second problem with neighbor pointers is that they don't provide much assistance in locating empty space for routing, since only the occupied space is represented explicitly. For these two reasons, neighbor pointers do not appear to be well-suited to interactive systems or those that provide routing aids.

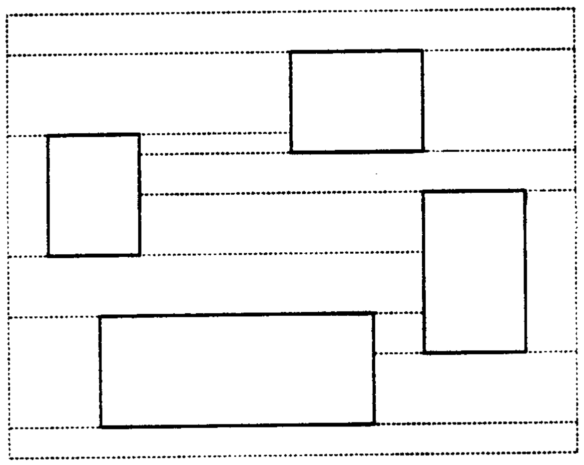

Figure 3. An example of tiles in a corner stitched data structure. Solid tiles are represented with dark lines, space tiles with dotted lines. The entire area of the circuit is covered with tiles. Space tiles are made as wide as possible.

4. Corner Stitching

Corner stitching arose from a consideration of the weaknesses of the above mechansisms, and has two features that distinguish it from them. The first important feature is that all space, both empty and occupied, is represented explicitly in the database. The second feature is a novel way of linking together the objects at their corners. These *corner stitches* permit easy modification of the database, and lead to efficient implementations for a variety of operations.

Figure 3 shows four objects represented in the corner stitching scheme. The picture resembles a mosaic with rectangular tiles of two types, space and solid. Tiles contain their lower and left edges, but not their upper or right edges, so every point in the plane is present in exactly one tile. The tiles must be rectangles with sides parallel to the axes.

The space tiles are organized as ***maximal horizontal strips***. This means that no space tile has other space tiles immediately to its right or left. When modifying the database, adjacent space tiles that are horizontally adjacent must be split into shorter tiles and then joined into maximal strips, as shown in Figure 4. After making sure that space tiles are as wide as possible, vertically adjacent tiles can be merged together if they have the same horizontal span. The representation of space is of no consequence to the VLSI layout or to the designer, and will not even be visible in real systems. However, the maximal horizontal strip representation is crucial to the space and time efficiency of the tools, as we shall see in Sections 5 and 6. Among its other properties, the horizontal strip representation is unique: there is one and only one decomposition of space for each arrangement of solid tiles.

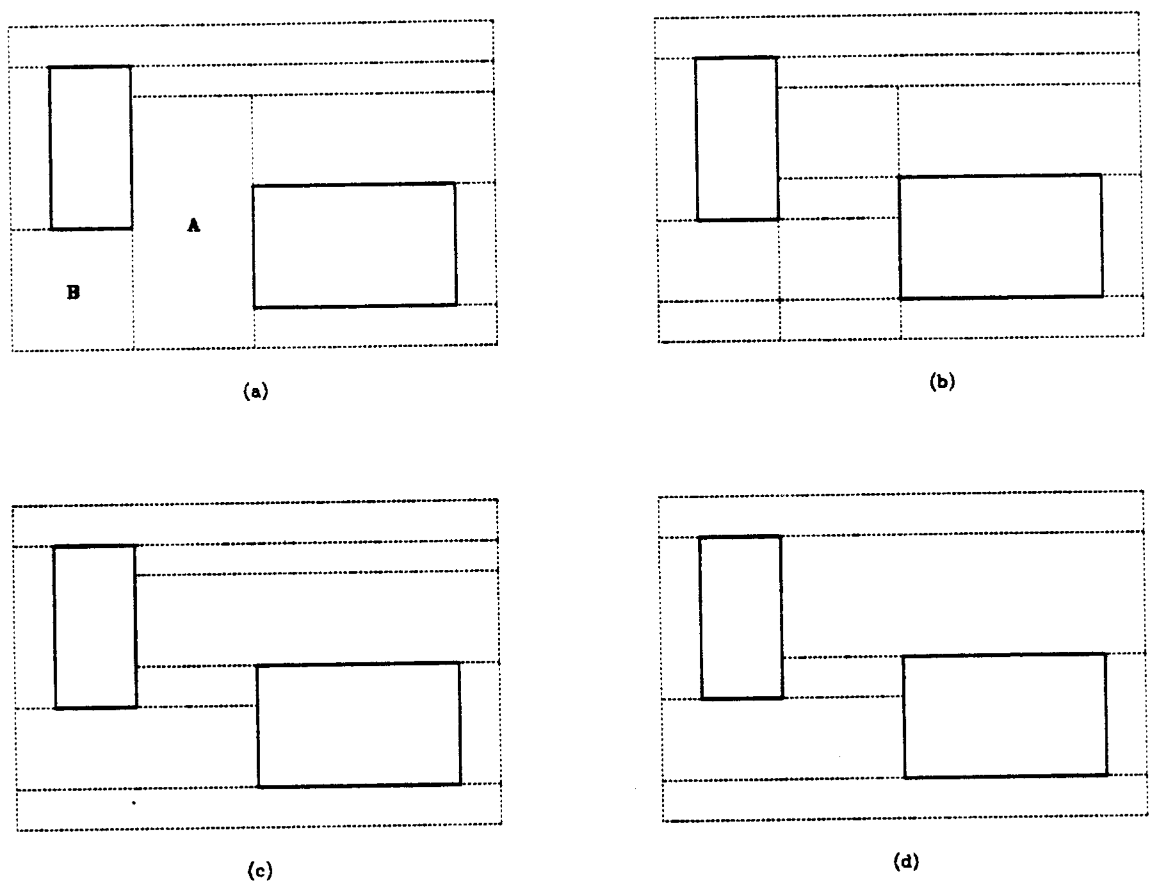

Figure 4. No space tile may have another space tile to its immediate right or left. In this example, tiles *A* and *B* in (a) must be split into the shorter tiles of (b), then merged together into wide strips in (c), and finally merged vertically in (d).

Tiles are linked by a set of pointers at their corners, called *corner stitches*. Each tile contains eight stitches, two at each corner, as illustrated in Figure 5. By linking together all adjacent tiles, corner stitches provide something equivalent to neighbor pointers.

The tile/stitch representation has several attractive features, which will be illustrated in the sections that follow. First, the mechanism combines both horizontal and vertical information in a single structure. The space tiles provide a form of registration between the horizontal and vertical information and make it easy to keep all the pointers up to date as the circuit is

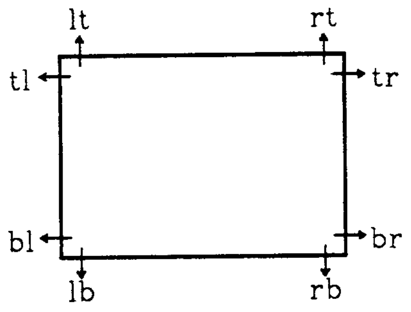

Figure 5. Each tile is connected to its neighbors by eight pointers called corner stitches. The name of each stitch indicates the location of the stitch (**lb** means left bottom corner), and the second letter indicates the direction in which the stitch points. Thus **lb** refers to the pointer at the left edge of the tile pointing out its bottom.

modified. Because the space tiles may vary in size (as opposed to fixed-size bins), the structure adapts naturally to variations in the size of the solid tiles. The maximal horizontal strip representation of space results in clean upper bounds on the number of space tiles and also on the complexity of the algorithms. All tiles have the same number of pointers to other tiles, which simplifies the database management.

5. Algorithms

This section presents algorithms for manipulating the tiles and corner stitches. The algorithms are presented in simplified form here; a few of them are described in more detail in the appendices. The most important attribute of all the algorithms is their locality: each algorithm depends only on information in the immediate vicinity of the operation. None of the algorithms has an average running time any worse than linear in the number of tiles in the affected area. One can devise pathological cases where the algorithms require time linear in the overall layout size, but in practice (particularly for VLSI layouts, which tend to be densely packed) their running times will be small and nearly independent of the size of the layout.

In discussing the performance of the algorithms, the corner stitches provide a good unit of measure. The complexity of the algorithms will be discussed in terms of the number of stitches that must be traversed (or, alternatively, the number of tiles that must be visited) and/or the number of stitches that must be modified.

5.1. Point Finding

Several different kinds of searching are facilitated by corner stitching. One of the most common operations is to find the tile at a given (x,y) location. Figure 6 illustrates how this can be done with corner stitching. The algorithm iterates in x and y, starting from any given tile in the database:

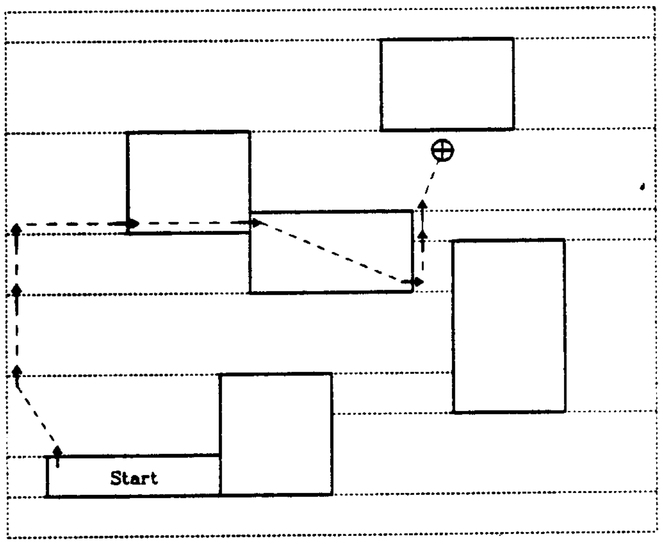

Figure 6. To locate the tile containing a given point, alternate between up/down and left/right motions.

1. First move up or down along the left edges of tiles (following **lt** and **lb** stitches) until a tile is found whose vertical range contains the desired point.

2. Then move left or right along the bottom edges of tiles (following **br** and **bl** stitches) until a tile is found whose horizontal range contains the desired point.

3. Since the horizontal motion may have caused a vertical misalignment, steps 1 and 2 may have to be iterated several times to locate the tile containing the point.

In the worst case, this algorithm may require every tile in the entire structure to be searched (this happens, for example, if all the tiles in the structure are in a single column or row). Fortunately, the average case behavior is much better than this. If there are a total of N space or solid tiles and they are of relatively uniform size, then on the order of $\sqrt{N}$ tiles will be passed through in the average case. For a layout containing a million tiles (which is typical of the fully expanded mask sets of current VLSI circuits) this means a few thousand tiles will have to be touched.

In interactive systems, there is a simple way to reduce the time spent in searches of this sort: keep around a pointer to any tile in the approximate area where the designer is working. When a large design is being edited, the designer's attention is generally focussed on a small piece of the design (e.g. a piece that can be viewed comfortably on a graphic device). If a tile in this area is remembered for reference, then search time depends only on how much is on the screen, not how large the design is.

5.2. Neighbor Finding

Another common searching operation is neighbor finding: find all the tiles that touch one side of a given tile. Neighbor finding is useful for design rule checking, compaction, and tracing out connected nets. Figure 7 illustrates how to find all the tiles that touch the right side of a given tile:

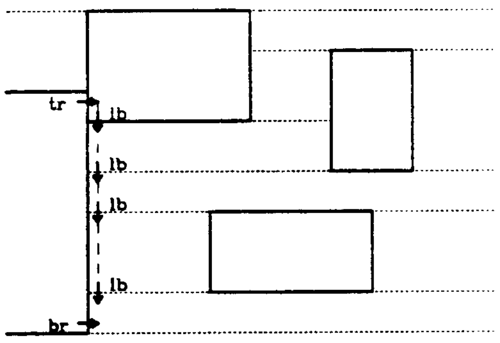

Figure 7. The corner stitches provide a simple way to find all the tiles that touch one side of a given tile.

1. Follow the **tr** stitch of the starting tile to find its topmost right neighbor.
2. Then trace down through **lb** stitches until the bottommost neighbor is found (the bottommost neighbor is the one pointed to by the **br** stitch of the starting tile).

Similar algorithms can be devised to search each of the other sides. The time for this search is linear in the number of neighbors. As shown in Appendix A, in the average case each tile has one or two neighbors along each side. In layouts where tile sizes vary greatly, the number of neighbors will, on average, be proportional to the length of the side.

5.3. Area Searches

A third form of searching is to see if there are any solid tiles within a given area. This can be accomplished in the following manner using corner stitches (see Figure 8):

1. Use the point-finding algorithm to locate the tile containing the lower-left corner of the area of interest.
2. See if the tile is solid. If not, it must be a space tile. See if its right edge is within the area of interest. If so, either it is the edge of the layout or the edge of a solid tile.

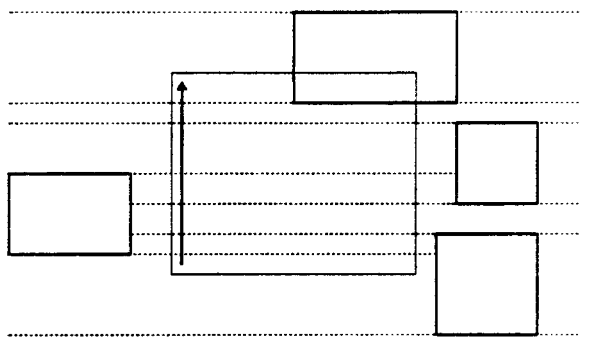

Figure 8. To search a rectangular area for a solid tile, work upwards along the left edge of the area. Each tile along the edge must be either a) a solid tile, b) a space tile that spans the entire area, or c) a space tile with a solid tile just to its right.

3. If a solid tile was found in step 2, then the search is complete. If no solid tile was found, then move upwards to the next tile touching the left edge of the area of interest. This can be done either by invoking the point-finding algorithm, or by traversing the **lt** stitch upwards and then traversing **br** stitches right until the desired tile is found.

4. Repeat steps 2 and 3 until either a solid tile is found or the top of the area of interest is reached.

As with the other operations, the time necessary for this operation depends only on local features: the number of tiles in and around the area of interest. The cost can be measured by counting the number of stitches that must be traversed. The number of iterations through the algorithm will be proportional to the height of the area (assuming, as always, a relatively uniform size distribution). In each iteration, it may be necessary to traverse one stitch in step 2. In addition, step 3 will cause a mislagnment of about 1/2 tile in the average case. Thus, the total running time is linear in the height of the search area, and does not depend at all on the width of the search area. In pathological cases this algorithm could have running time proportional to the total number of tiles in the layout (this happens when there is severe misalignment in step 3).

5.4. Create and Delete

Before creating a new solid tile, we must check to see that there are no existing solid tiles in the desired area of the new tile. The area search algorithm can check this. The second step is to insert the tile into the data structure, clipping and merging space tiles and updating corner stitches as

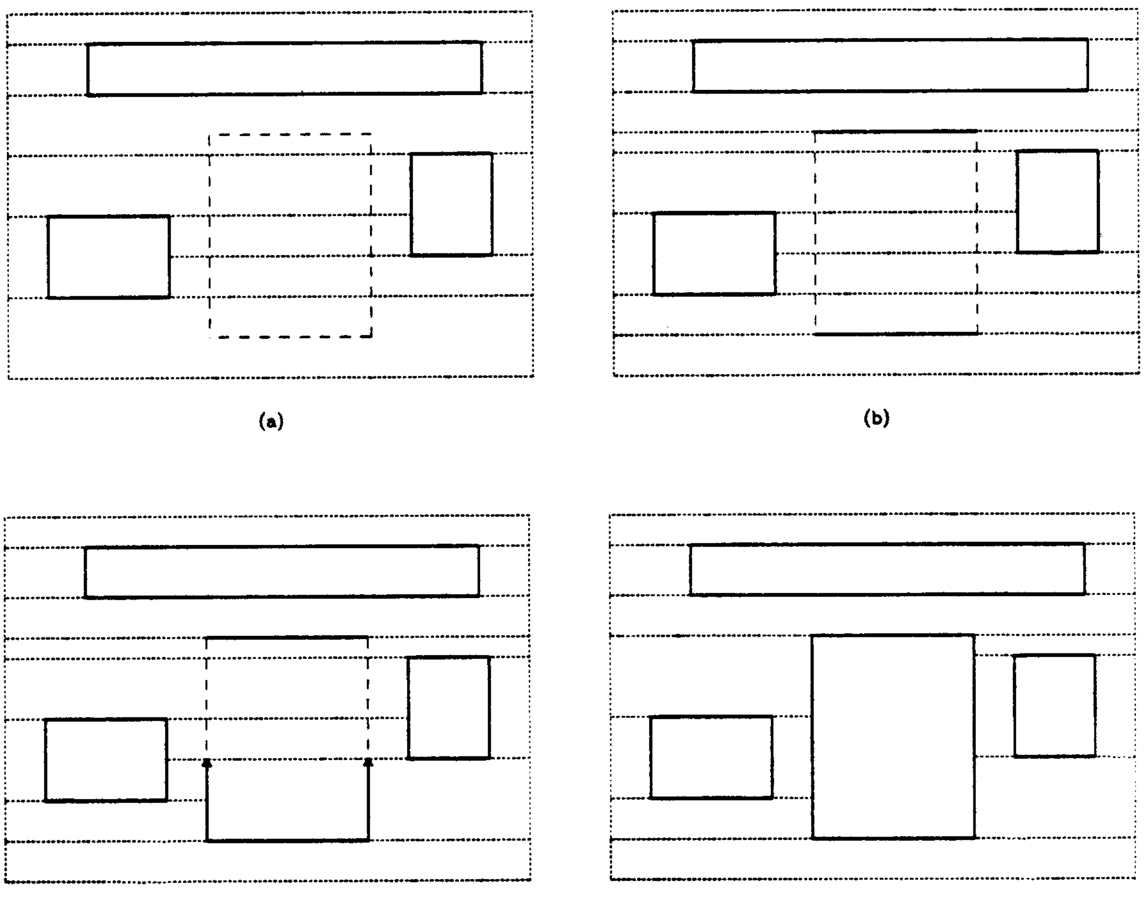

Figure 9. Inserting a new solid tile into the data structure. (a) shows the desired location of the new tile. In (b) the space tiles containing the top and bottom edges of the new solid tile are split. In (c) and (d) the area of the new tile is traversed from bottom to top, splitting and joining space tiles on either side and pointing their stitches at the new solid tile.

shown in Figure 9. The insertion algorithm is as follows:

1. Find the space tile containing the top edge of the area to be occupied by the new tile (because of the strip property, a single space tile must contain the entire edge).

2. Split the top space tile along a horizontal line into a piece entirely above the new tile and a piece overlapping the new tile. Update corner stitches in the tiles adjoining the new tile.

3. Find the space tile containing the bottom edge of the new solid tile, split it in the same fashion, and update stitches around it.

4. Work up the sides of the new tile. Each space tile must be split into two space tiles, one to the left of the new solid tile and one to the right. This splitting may make it possible to merge space tiles vertically: merge whenever possible. After each split or merge, stitches must be updated in adjoining tiles.

The speed of the creation algorithm is determined by the cost of splitting and merging the space tiles that cross the area. The number of space tiles depends on the number of solid tiles in the left and right shadows of the new tile. One can devise cases where the number of space tiles is arbitrarily high, but in practice it can be be expected to be proportional to the relative height of the new tile in comparison to the tiles around it. Appendix B discusses the cost of splitting and merging tiles: in the average case it is constant; for very large tiles it is proportional to the circumference of the tile. This means that in the worst possible case the cost of creating a new tile could be proportional to the total number of tiles in the layout. In the average case the running time will be proportional to the height of the new tile and independent of its width.

Deletion is performed in the inverse manner of creation. First, the solid tile to be deleted is turned into a space tile. Then the space tile must be split along horizontal lines so it can be merged with neighboring space tiles into maximal horizontal strips. It may also be necessary to split some of the adjacent space tiles to form maximal strips. After making maximal strips, it may be possible to merge space tiles vertically at the top and bottom of the space once occupied by the solid tile. Appendix C discusses the deletion algorithm in detail. As with tile creation, the average running time will be linear in the height of the deleted tile, with worst case time linear in the total number of tiles in the layout.

5.5. Directed Area Enumeration

For many applications, such as compaction and layout rule checking, it is useful to enumerate all the tiles in a given area, i.e. to "visit" each tile exactly once. Furthermore, it is often useful to do this in a particular direction. This section presents an algorithm wherein each tile is visited only after all the tiles to its left have been visited. I call such an enumeration a *directed enumeration*. Corner stitching makes this a linear time operation, as illustrated by the following algorithm for left-to-right directed enumeration. Figure 10 shows the resulting enumeration order for an example case.

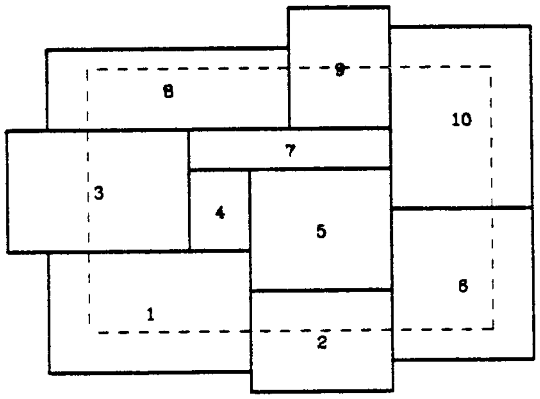

Figure 10. An example of directed enumeration. When doing a left-to-right enumeration of the dashed area, the tiles will be enumerated in order of their numbers.

1. As for the area searching algorithm, use the point finding algorithm to locate the tile at the bottom left corner of the area of interest. Then step up through all the tiles along the left edge, using the same technique as in area searching.

2. For each tile found in step 1, enumerate it recursively, as indicated in steps 3 through 7.

3. Enumerate the tile (this will generally involve some application-specific processing).

4. If the right edge of the tile is outside of the search area, then return.

5. Otherwise, use the neighbor finding algorithm to locate all the tiles that touch the right side of the current tile and intersect the search area.

6. For each neighbor, if the top left corner of the neighbor touches the current tile then enumerate the neighbor recursively (in Figure 10, this occurs when tile 1 is the current tile and tile 2 is the neighbor).

7. Or, if the top edge of the search area cuts both the current tile and the neighbor then enumerate the neighbor recursively (in Figure 10, this occurs when tile 8 is the current tile and tile 9 is the neighbor).

The algorithm for directed enumeration provides the key to snowplowing and compaction. Its running time is linear in the number of tiles intersecting the search area. This can be shown by the following arguments. The checks in steps 6 and 7 guarantee that each tile is enumerated exactly once. However, a tile may be checked several times before satisfying the checks in step 6 or 7: it will be checked once for each tile that touches its left side. The total running time of the algorithm is thus proportional to the total number of adjacencies within the search area. Appendix A uses the properties of planar graphs to prove that the number of adjacencies must be linear in the number of tiles.

The algorithm for directed enumeration does not depend on the fact that space tiles are maximal horizontal strips. In fact, the basic algorithm does not even distinguish between solid and space tiles. It is possible to devise analogous algorithms for each of the other three directions, all of which have linear running time.

5.6. Snowplow

Snowplow is an example of a useful operation that cannot easily be implemented with most existing data structures. When one piece of a large design is moved, it would be helpful if other pieces of the design lying in the

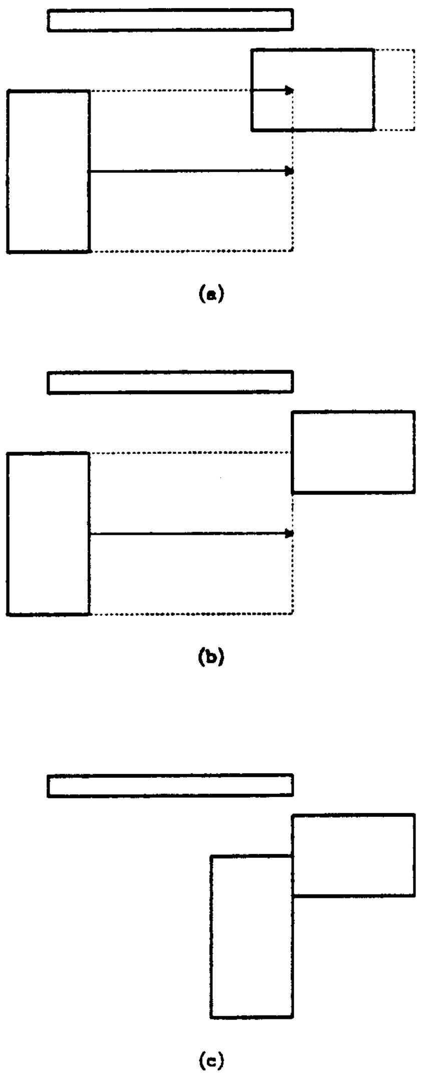

Figure 11. An example of snowplowing: (a) determine the area to be swept out by the motion; (b) recursively move all solid tiles out of this area; (c) move the original tile.

path of motion could be moved as well, as if the original piece were a snowplow. Ideally, such a motion should stretch or shrink the design while maintaining design rules and connectivity. Snowplowing can be accomplished with corner stitching in the following way:

1. Determine the rectangular area that will be swept out by the motion of the original tile (see Figure 11).
2. Use the area finding algorithm to see if there are any solid tiles in the plow area. If a solid tile is found, invoke the snowplow algorithm recursively to move the tile out of the snowplow area. Repeat this step until no solid tiles are found.
3. Delete the original tile from its old location and create it at the new position.

In the worst case, this algorithm could require on the order of N^2 tile moves to move N tiles out of the snowplow area (this happens if each recursive move requires all the tiles that have been previously moved to be moved still farther). A better algorithm can be achieved by using a form of directed enumeration. The enumeration is done in the direction of the snowplow, so that a tile is visited only after all tiles that affect its final location have also been visited. Because the actual algorithm is somewhat complicated, it is presented separately in Appendix D. The running time of the directed algorithm is linear in the number of tiles in the snowplow area in the average case, and has worst case running time linear in the overall layout size.

5.7. Compaction

Most existing algorithms for compaction require N^2 time in the worst case for a layout containing N elements, and have been empirically observed to have average running time close to $N^{1.2}$ [7]. With corner stitching, compaction is linear in the size of the layout. Compaction in a single direction can be achieved in a simple way by snowplowing a large tile across the lay-

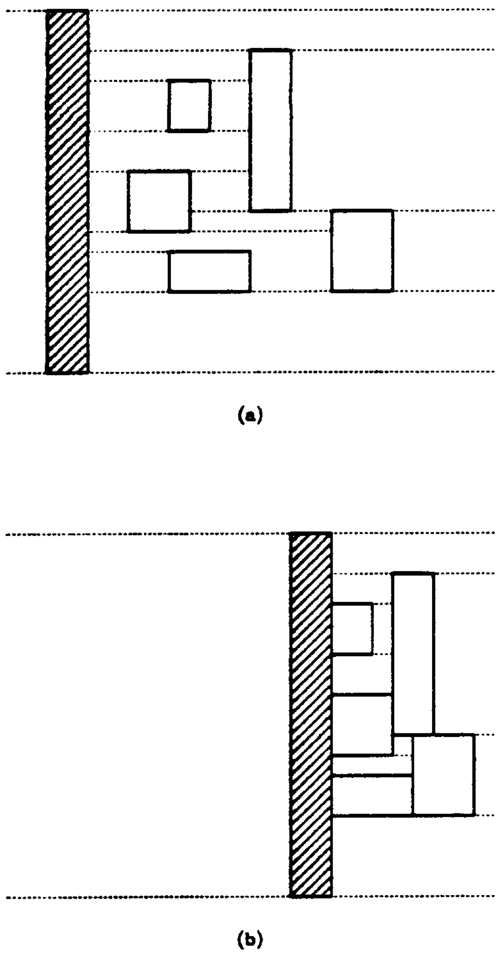

Figure 12. To compact a layout vertically, snowplow a large additional tile (cross-hatched in the figure) across the layout: (a) shows the configuration before the snowplow, and (b) shows the compacted configuration afterwards. The tile acts like a broom and compacts as it sweeps.

out, as shown in Figure 12. The linear running time for snowplowing guarantees that this form of compaction will also be linear in the size of the layout.

There are two keys to the speed of compaction in corner stitching. The first, and most important, is that all the dependencies between tiles are maintained dynamically. In other compaction systems, the dependencies must be reconstructed after each change to the layout; the algorithms for generating dependencies limit the overall speed of compaction. The second key is that the layout is planar. This means that the number of adjacencies is linear in the number of tiles, and hence the whole layout can be scanned in time proportional to the number of tiles.

5.8. Channel Finding

Channel information is constantly available in the form of the space tiles. The corner stitches make it possible to find connected channels and thereby trace out signal paths. Of course, some routers may prefer a different representation of channels than maximal strips; if this is the case, then conversion will be necessary to cast the space tiles into a form suitable for routing.

6. Space Requirements

Because of the enormous size of VLSI designs, a data structure used for VLSI CAD must be space efficient if it is to be effective. For example, even the hierarchical representation of a 45000 transistor chip requires about 1.5 million bytes of main memory in Caesar. Corner stitching requires more information to be kept in the data structure than systems like Caesar. Table I compares corner stitching to the linked-list scheme of Caesar. Each solid

	Caesar	Corner Stitching	Optimized CS
Coordinates	x_1, y_1, x_2, y_2 (16 bytes)	x_1, y_1, x_2, y_2 (16 bytes)	x_1, y_1 (8 bytes)
Pointers	1 link (4 bytes)	8 stitches (32 bytes)	4 stitches (16 bytes)
Total	20 bytes	48 bytes	24 bytes

Table I. In the worst case, corner stitching requires more than twice as much storage per tile as simpler systems. An optimized version of corner stitching needs only four more bytes per tile than Caesar.

tile requires 48 bytes using corner stitching, compared to only 20 bytes in Caesar. Furthermore, Caesar and other systems do not represent empty space, whereas there may be many space tiles in corner stitching. If there are many space tiles, then corner stitching will require too much space to be practical. The following subsections show how to reduce the memory needed for each tile and prove an upper limit on the number of space tiles required.

6.1. Reducing the Storage Requirements

There are two time-space tradeoffs that can be made to reduce the number of bytes needed to represent each tile. First, we need only store one coordinate pair for each tile. In the corner stitch repesentation, only the lower-left x- and y-coordinates need to be stored: to find out the upper x- or y-coordinate, traverse a corner stitch and look at the lower-left coordinates of a neighoring tile. This requires an extra pointer traversal each time an upper coordinate is used, but saves 8 bytes per tile.

It is also unnecessary to have eight corner stitches per tile. All of the operations supported by the eight stitch scheme will work with only four stitches from two diagonally opposite corners, as shown in Figure 13. With the four stitches shown in Figure 13, it is possible to reconstruct all the other stitches. Consider the **lt** stitch of a tile, which is not present in the scheme of Figure 13. It can be generated by traversing the **rt** stitch and then following **bl** stitches until a tile is found whose left edge is no further right than the left edge of the starting tile. The time required for this operation depends on the number of neighbors a tile has.

In practice, the above space savings do not cost much time at all. The lack of some pointers tends to favor certain directions for operations, but the algorithms can all be recoded to operate in the favored directions. In

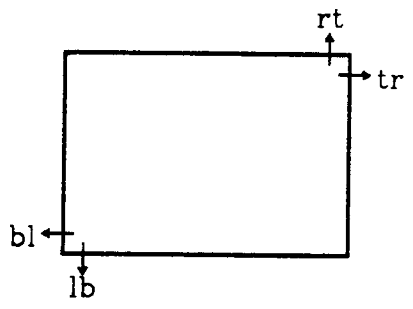

Figure 13. Four corner stitches, two at each of two diagonally opposite corners, are sufficient to support the algorithms.

fact, the code is much simpler in the reduced case (and may even be faster) because there are only half as many pointers and coordinates to modify. The only algorithm that required substantial changes is the algorithm presented in Appendix C for tile deletion. As a result of both space reductions, a tile requires 24 bytes using corner stitching, compared to 20 in Caesar.

6.2. Worst-Case Requirements

If there are N solid tiles in a circuit, then corner stitching will never result in more than 3N+1 space tiles. Furthermore, the horizontal strip representation is at least as efficient (in the worst case) as any other rectangle-based representation of space. In tightly-packed designs, which are typical in VLSI, the number of space tiles is much less than 3N+1.

The proof of the 3N+1 upper limit is due to Carlo Séquin. To see that no more than 3N+1 space tiles are needed for N solid tiles, place the solid tiles one at a time in order from right to left as shown in Figure 14. Initially there is a single space tile. When each solid tile is placed, it can result in no more than 3 new space tiles: the top and bottom edges may each cause a space tile to be split, and a new space tile will be created in the shadow to the left

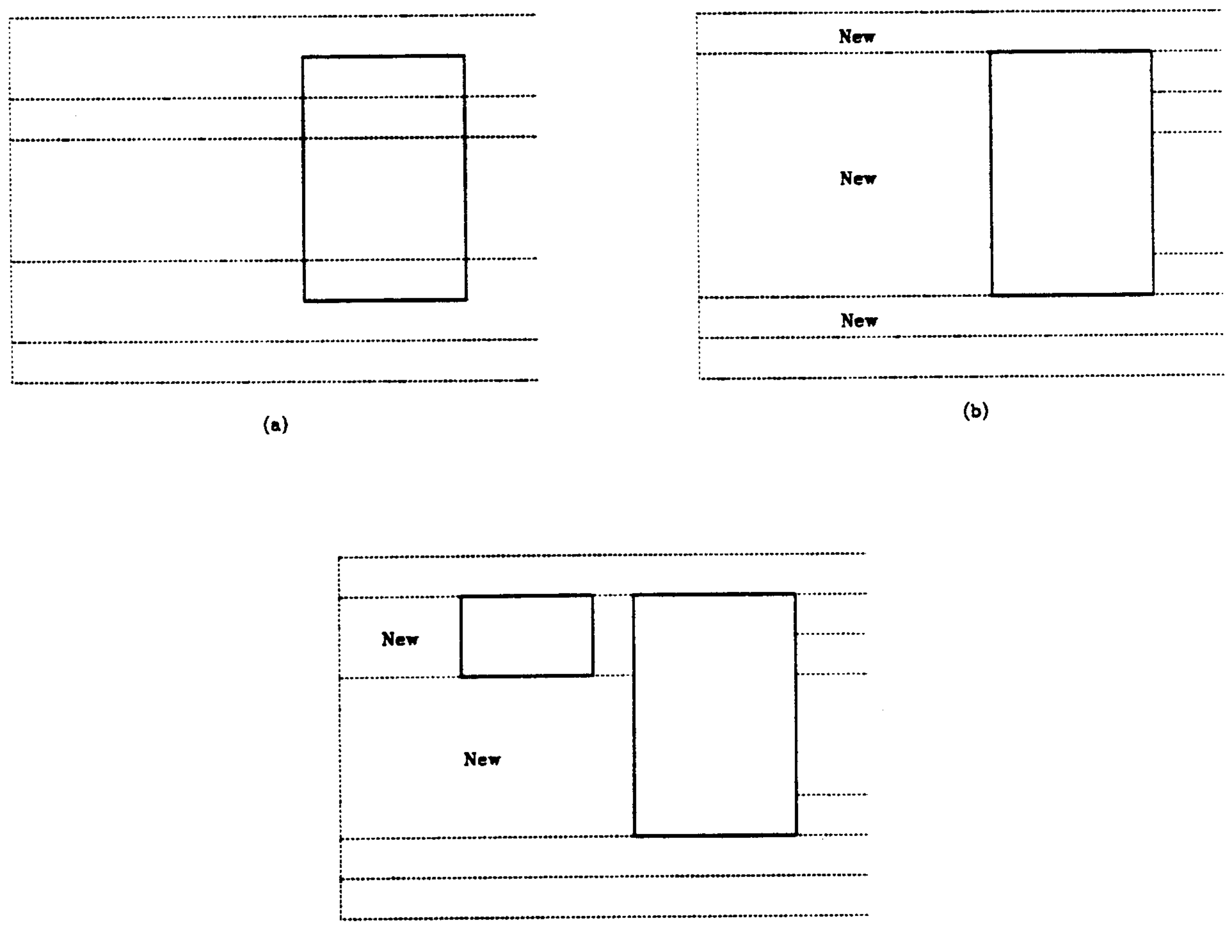

Figure 14. (a) and (b) show that if solid tiles are inserted in order from left to right, each tile causes no more than 3 additional space tiles to be created. However, if edges of the new tile align with edges of old tiles, as in (c), less than 3 additional space tiles will be required.

of the solid tile. Because we place the solid tiles in order, there can be no solid tiles in the shadow. This means that only a single space tile will be created there. Although the solid tiles were placed in a particular order to demonstrate the 3N+1 limit, the final configuration is independent of the order in which the tiles are placed (the horizontal strip property guarantees this). Thus the result is valid regardless of the order of solid tile creation.

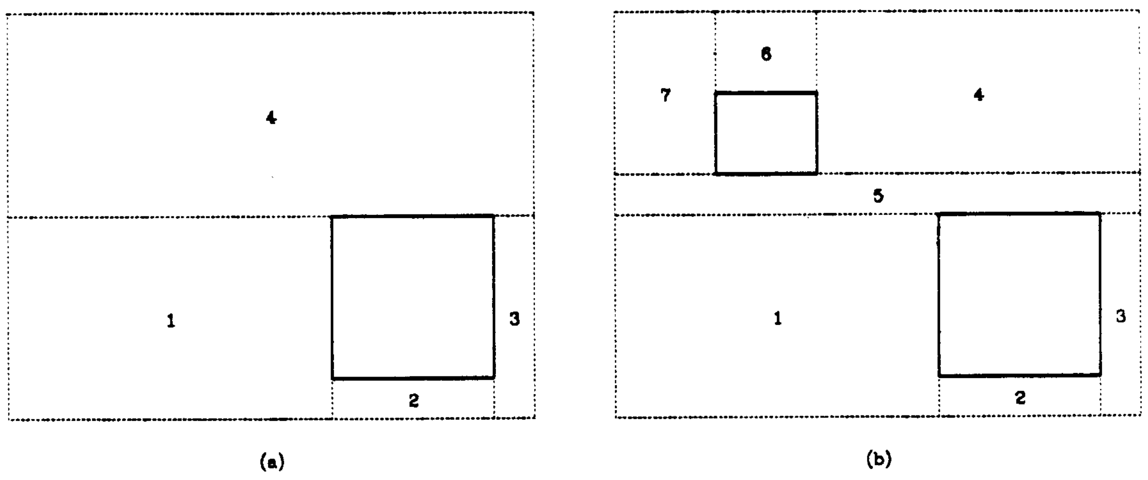

Figure 15. In pathological situations where no two solid tiles have colinear edges, at least 3N+1 space tiles must be used, even in representations other than horizontal strips.

There are many other ways to organize space tiles besides **maximal horizontal strips.** However, in the worst case no representation of space can use less than 3N+1 space tiles. This worst case occurs when no two solid tiles have colinear edges. Figure 15 shows one such situation.

Somewhat less than 3N+1 space tiles are needed for actual VLSI applications. Fewer space tiles are needed whenever edges of neighboring solid tiles align. For example, Figure 14c shows a situation where the placement of a solid tile only adds 2 space tiles instead of 3. In integrated circuits the solid tiles must touch each other to achieve electrical connectivity, so the number of space tiles actually needed is much less than 3N. Table II shows sample data gathered from three pieces of layout by building corner stitched data structures for individual mask layers. On the average, 1.5 space tiles are required per solid tile. This means that the total storage required for geometry in corner stitching will be about three times as great as in systems like Caesar. This result applies even when the mask layers are sparse, as in the global routing example.

7. An Implementation

A simple program was written to test the basic viability of corner stitching. It implements all of the algorithms described above for the simplified assumptions of space and solid tiles. About 1100 lines of C code were required. The code is embedded in an interactive system: using a tablet and color graphics display, tiles can be created, deleted, and snowplowed, and horizontal and vertical compaction can be invoked. For the small layouts used to test the program (50 or fewer solid tiles) response is instantaneous for all operations.

8. Extensions for Real VLSI

The scheme presented here must be extended in several ways before it will be practical for real integrated circuits. This section presents some of the important issues and discusses a few possible solutions. Work is

Layout	Layer	Solid Tiles	Space Tiles	Space/Solid
Global Routing	Polysilicon	2736	4071	1.5
	Diffusion	1250	1916	1.5
	Metal	2305	3616	1.6
ALU	Polysilicon	809	1223	1.5
	Diffusion	1284	1590	1.2
	Metal	495	653	1.3
Register File	Polysilicon	236	305	1.3
	Diffusion	376	433	1.2
	Metal	70	113	1.6

Table II. For actual layouts, corner stitching requires about 1.5 space tiles for each solid tile. The mask layers were measured separately. The first case consists of all the global routing for the RISC I microprocessor (i.e. all the rectangles in the topmost cell of the hierarchy). The second and third cases consist of areas extracted from another microprocessor.

currently underway to construct a viable VLSI layout tool for nMOS and CMOS based on corner stitching.

First, it must be possible to have multiple mask layers. There are several ways of accomplishing this. One alternative is to permit many different types of solid tiles, one type for each possible combination of mask layers. Unfortunately, this scheme will result in enormous numbers of tiny tiles in places where several mask layers cross each other. Another alternative is to keep a separate corner stitched structure for each mask layer. This scheme will be relatively space efficient, but will require frequent cross-registration between planes during operations such as snowplowing and design rule checking. A third alternative is to use a combination of the above two schemes. Layers that interact strongly, such as polysilicon and diffusion, can be kept together in a single structure with different types of solid tiles for each layer combination. Layers that interact weakly, such as polysilicon and metal, can be kept in different structures. Registration between structures need only occur where there are contacts. Under this scheme, the corner-stitched representation corresponds very closely to the electrical circuit, since the transistors (polysilicon crossing diffusion) are represented by tiles of a special type. In addition, each corner-stitched structure can be design-rule checked independently.

A practical implementation of corner stitching must also be able to handle more complex layout rules than the simple non-overlap rule used here. The paradigm for rules checking should be chosen so that it works well both for simple design rule checking and for the snowplow and compaction operations. The implementation of snowplow must be modified so that it maintains the circuit connectivity as well as the design rules. This means that some tiles (those corresponding to wires) will be stretched or shrunk rather than just moved. Other tiles, such as those corresponding to transistors or contacts, cannot be stretched or shrunk without changing the circuit behavior.

Lastly, a practical implementation of corner stitching must support hierarchical designs. This can be accomplished by keeping separate corner-stitched structures for each cell.

9. Conclusion

Corner stitching is a powerful technique for representing geometrical data. Its two most important features are a) it represents empty space explicitly, and b) it links together tiles of various types at their corners. These two features make it possible to implement a variety of important operations that operate purely locally. The efficiency of the algorithms depends only on local information and not on the overall circuit size. The database can be modified incrementally, so that one portion of the design can be changed without invalidating the pointer information of any other piece of the design. Corner stitching is effective both for densely packed and for sparse circuits.

There are two potential drawbacks of the mechanism. The first drawback is that it requires approximately 3 times as much storage as simple mechanisms. Fortunately, designers tend to focus their attention on a small portion of a layout at any given time; since corner stitching uses only local information, it should work quite effectively in a demand-paged environment. The second drawback to corner stitching is that it requires designs to be Manhattan. This may be considered a serious restriction by some designers, but seems to be gaining more and more acceptance as designs reach very large degrees of integration.

10. Acknowledgements

Michael Arnold, Carlo Séquin, David Ungar, and David Wallace all took part in the discussions that led to the formulation of corner stitching. Séquin developed the proof that 3N+1 space tiles are always sufficient in a design with N solid tiles. Leo Guibas, Dave Patterson, Alberto Sangiovanni-Vincentelli, and Carlo Séquin all provided helpful comments on an early draft of the paper.

11. References

[1] Bentley, J.L. and Friedman, J.H. "A Survey of Algorithms and Data Structures for Range Searching." *ACM Computing Surveys*, Vol. 11, No. 4, 1979.

[2] Hsueh, M.Y. *Symbolic Layout and Compaction of Integrated Circuits*. Technical Report, University of California, Berkeley, UCB/ERL/M79/80, December 1979.

[3] Kedem, G. "The Quad-CIF Tree: A Data Structure for Hierarchical On-Line Algorithms." *Proc. 19th Design Automation Conference*, 1982, pp. 352-357.

[4] Keller, K.H. and Newton, A.R. "KIC2: A Low Cost, Interactive Editor for Integrated Circuit Design." *Digest of Papers for COMPCON Spring 1982*, pp. 305-306.

[5] Ousterhout, J.K. "Caesar: An Interactive Editor for VLSI". *VLSI Design*, Vol. II, No. 4, Fourth Quarter 1981, pp. 34-38.

[6] Ousterhout, J.K. and Ungar, D.M. "Measurements of a VLSI Design." *Proc. 19th Design Automation Conference*, 1982, pp. 903-908.

[7] Sangiovanni-Vincentelli, A. Private communication.

Appendix A: Adjacencies

The running time for several of the algorithms depends on the number of neighbors an individual tile has. One can construct situations where a tile has an arbitrarily large number of neighbors, so it is not possible to state any absolute upper bounds. However, graph theory can be used to determine the average number of neighbors. In any connected planar graph,

$$n - e + f = 1$$

where n is the number of nodes, e is the number of edges, and f is the number of faces contained by the edges. A face corresponds to a tile, a node to a corner of a tile, and an edge to a distinct adjacency between two tiles. For T tiles, $f = T$. The number of distinct nodes n can be at most $4T$, but in the interior of the tile structure each corner of one tile must coincide with at least one corner of another tile (a "T" structure). Thus, $n \leq 2T$ and the total number of adjacencies is

$$e = n + f + 1 \leq 3T + 1$$

Note that at the outside of the structure there may be corners that don't coincide with other corners, but for each of these there is also as least one edge that doesn't represent an adjacency (because there is no tile on the other side). Hence the $3T+1$ upper limit is not affected.

The $3T+1$ limit counts each adjacency only once for the two tiles that are adjacent. To compute the number of neighbors per tile, the figure must be doubled. This means that on the average, an individual tile will have about six neighbors, or about one or two on each side. This is regardless of the arrangement of tiles. Of course, if there are many tiles of different sizes, the large tiles may have many more than six neighbors. The average number of neighbors of a tile in a situation like this will be roughly proporational to the perimeter of the tile, which is less than linear in its area.

Appendix B: Splitting and Merging

This section discusses the cost of splitting one tile into two adjacent tiles, or merging two adjacent tiles into a single tile. A tile can be split into two tiles as follows:

1. Make an exact copy of the original tile.
2. Update the coordinates of each tile to reflect the split, and set the tiles' corner stitches to refer to each other.
3. Update the corner stitches in tiles that are now adjacent to the new tile. This done by chasing the stitches around three sides of the original tile and updating the stitches that must point to the new tile.

The algorithm for merging two adjacent tiles into a single larger tile is similar: stitches must be updated along three sides of the new larger tile.

The cost of each algorithm consists of constant factors (copying a tile or changing an x or y coordinate) and the search of neighbors on three sides. Appendix A showed that the number of neighbors was constant when averaged across a whole design, but increases for those tiles that are much larger than their neighbors. In this case the average number of neighbors will be approximately proportional to the perimeter of the tile. Thus the cost of a split or merge is constant if the tile being split or merged is about the same size as its neighbors. If the tile is much larger than its neighbors, then the cost increases in proportion to the tile's perimeter, which is less than linear in its area.

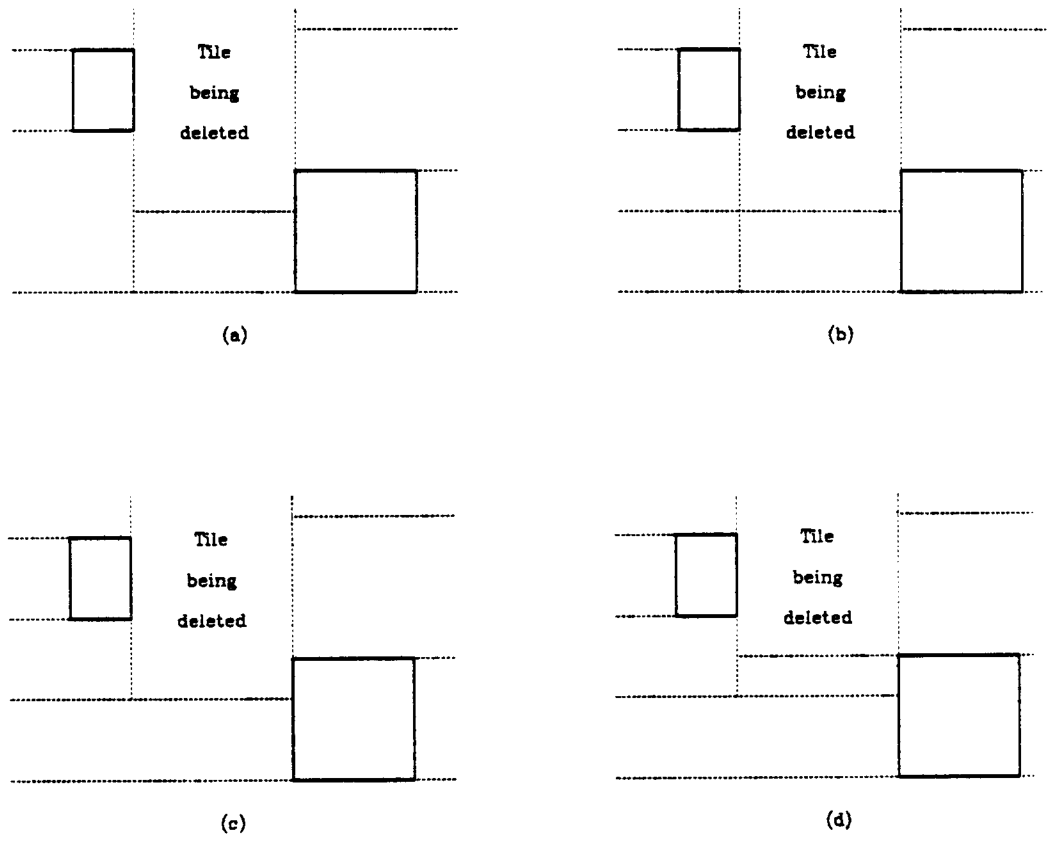

Figure 16. One iteration of the tile deletion algorithm: (a) corresponds to the situation at the beginning of step 3, (b) occurs after neighbors have been split, (c) occurs after the neighbors have been joined, and (d) occurs when the deleted tile is split and the algorithm is about to re-iterate.

Appendix C: The Deletion Algorithm

The complete algorithm for deleting a solid tile is presented below. The algorithm assumes that there are only four stitches per tile (this is the only one of the algorithms that is affected by the elimination of four stitches). Figure 16 illustrates one step in the action of the algorithm.

1. Change the type of the deleted tile from solid to space.

2. Scan the right edge of the tile from top to bottom and generate a list of neighbors ordered from bottom to top. This is done in order to avoid rescanning all the right neighbors in each pass through step 3. In 8-stitch implementations this step is not necessary.

3. Start at the bottom edge of the deleted tile. Use the corner stitches to find the tile just to the left of the edge and the list from step 2 to find the tile just to the right of the edge.

4. If the tile to the left of the edge is a space tile and spans the edge, then split it into two space tiles, one at and above the edge and one below it. Do the same thing for the tile to the left of the edge. Figure 16b shows the result of this operation.

5. If the tile just below the left end of the edge is a space tile (this tile is found by following the **lb** stitch, and is always a space tile except for the first iteration), merge it with its neighbors to the left or right if they are also space tiles. Figure 16c show the result of this operation.

6. During the first iteration it is also necessary to locate the tile just underneath the right end of the edge (follow the **lb** stitch followed by as many **tr** stitches as necessary) and merge it with its neighbor to the right, if possible.

7. Check the tile just below the left end of the edge to see if it can be merged vertically with the tile just below it (a merge can only occur during the very first iteration).

8. If the edge is the top of the deleted tile, then go to step 9. Otherwise, determine whether the left or right neighbor of the tile has the lowest upper edge. Split the deleted tile at this point, and repeat steps 3 through 8 with the upper portion of this tile. Figure 16d shows the result of the split.

9. When the top of the deleted tile is reached, check to see if the space tile at the top can be merged with the tile just above it, and merge if possible.

As with the other algorithms, deletion could require a large amount of time in pathological cases. For example, Figure 17 shows a situation where corner stitches will have to be examined and modified in every single tile in the layout (running time will be proportional to the overall layout size). However, situations like this are not likely in integrated circuits. If the tiles are about uniform in size, then the number of splits and joins will be roughly constant, and the work in each split and join will be roughly constant, resulting in a total running time independent of the overall size of the layout. Pathological cases correspond to neighborhoods with great complexity, e.g. tiles

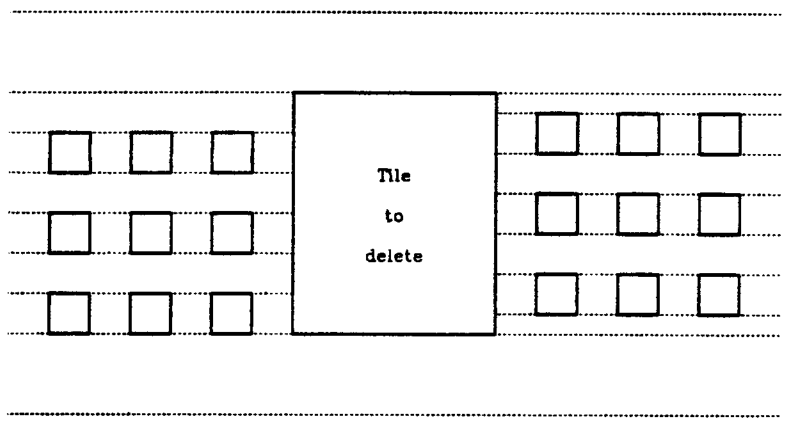

Figure 17. A pathological case for deletion. The running time is proportional to the size of the whole layout: when splitting and merging the wide space tiles it will be necessary to update stitches in every tile in the layout.

with many neighbors.

Appendix D: Snowplowing in Linear Time

A linear-time snowplow algorithm can be generated by extending the algorithm for directed enumeration that was presented in Section 5.5. First, each tile must store a value containing the amount by which the tile must be moved. Initially the motion amounts are all zero. The basic directed enumeration algorithm is used to scan the snowplow area in the direction of the snowplow. When scanning a tile's neighbors (step 6 of the algorithm in Section 5.5), add the current tile's width (if it is solid) or zero (if it is space) to the tile's motion amount, and store that value as the motion amount of each neighbor, unless the neighbor's motion amount is already greater than the new value.

If the original snowplow area contains solid tiles, then the area must be extended in order to include the area that will be snowplowed by the displaced tiles. This can be done during the directed enumeration by using the original edge of the snowplow area plus the motion amount of the current tile as the effective edge of the search area.

At the very beginning of the enumeration for each tile (step 3 in the algorithm), check to see if the tile is solid. If it is, then add an entry to the front of a list of tiles to be moved. The list entry contains the motion amount and a pointer to the tile. At the very end of the enumeration of each tile, just before returning, zero out the tile's motion amount to restore its initial zero value for future snowplows. After the directed enumeration has been completed, the list of tiles is processed in order to actually move tiles. The ordering of the list (opposite to the order of enumeration) guarantees that the space into which each tile is moved will be empty. After the list is moved, then the original snowplow tile is moved.

The running time of the enumeration was shown in Section 5.5 to be linear in the number of tiles in the search area. Processing the list to move the tiles requires each tile to be deleted and then added at some other position. For layouts with uniform tile sizes, the time for each tile will be constant so overall snowplow time will be linear in the number of tiles in the snowplow area. In a pathological case, the time to move a single tile could be proportional to the size of the entire layout, but this cannot happen for all (or even very many) of the tiles in the affected area. This can be seen by noting that the total number of adjacencies in an area is no worse than linear in the number of tiles in the area; if one tile has very many neighbors (and hence takes a long time to process), then some other tiles must have very few neighbors. The total work done in moving the tiles is proportional to the total number of adjacencies in the affected area, and hence can be no worse than linear in the size of the entire layout. Thus the average behavior of this snowplow algorithm is linear in the number of tiles in the affected area, and the worst case behavior is linear in the number of tiles in the whole layout.