Plowing:
Interactive Stretching and Compaction in Magic

Walter S. Scott and John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

The Magic layout editor provides a new operation called *plowing*, for stretching and compacting Manhattan VLSI layouts. Plowing works directly on the mask-level representation of a layout, allowing portions of it to be rearranged while preserving connectivity and layout-rule correctness. The layout and connectivity rules are read from a file, so plowing is technology independent. Plowing is fast enough to be used interactively. This paper presents the plowing operation and the algorithm used to implement it.

Keywords and Phrases: interactive layout editor, stretching, compaction.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00034-K-0251

1. Introduction

Plowing is a new operation provided by the Magic layout editor [OHMST 84] for stretching and compacting Manhattan VLSI layouts. It allows designers to make topological changes to a layout while maintaining connectivity and layout rule correctness. Plowing can be used to rearrange the geometry of a subcell, compact a sparse layout, or open up new space in a dense layout. In a hierarchical environment plowing also allows cell placement to be modified incrementally without the need for rerouting. To avoid dependence on a particular technology, plowing is parameterized by a set of layout and connectivity rules contained in a technology file.

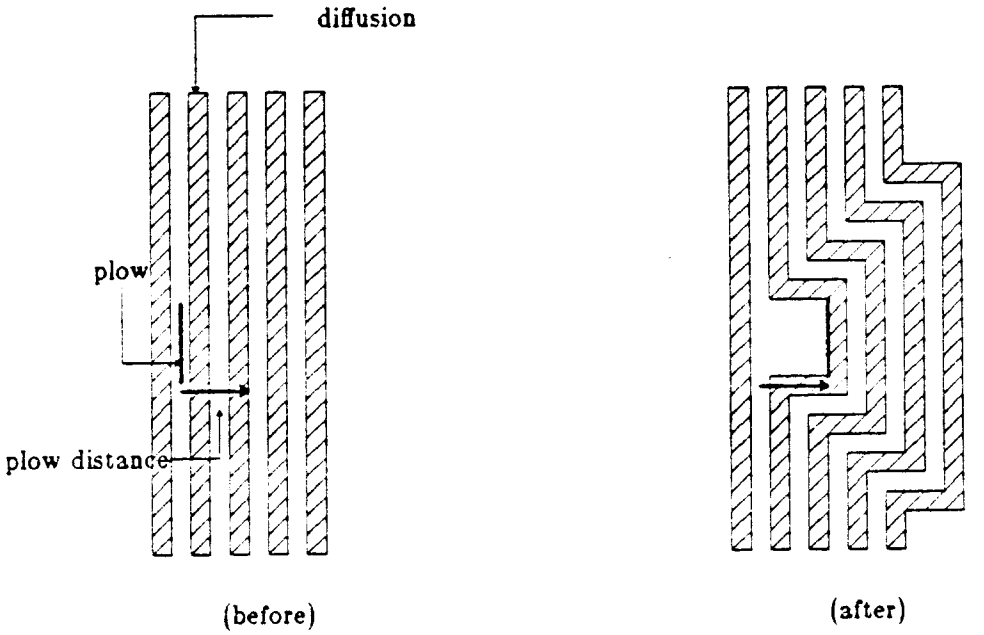

Figure 1. Plowing opens up new space in a dense layout. Geometry is pushed in front of the plow, subject to layout-rule constraints. The connectivity of the original layout is maintained. Jogs are inserted automatically where necessary.

Conceptually the plowing operation is very simple. The user places either a vertical or a horizontal line segment (the *plow*) over some part of a mask-level representation of the layout, and then gives the direction and the distance the plow is to move. Plowing can be done up, down, to the left, or to the right. (The rest of this paper will assume plowing to the right.) The plow is then moved through the layout by the distance specified. It catches vertical edges (boundaries between materials) as it moves and carries them along with it. Since only edges are moved, material behind the plow is stretched and material in front of the plow is compressed. Figure 1 shows how plowing can be used to open up new space. Figure 2 shows how it can be used for stretching. Plowing can be used to compact an entire cell by placing a plow to the left and plowing right, then placing a plow at the top and plowing down.

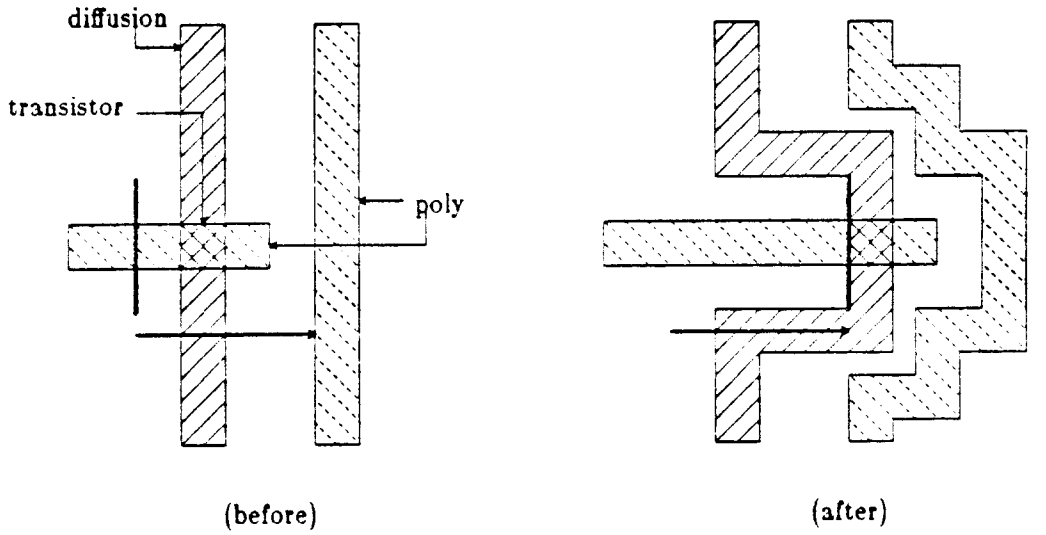

Figure 2. Material to the left of the plow is stretched. Material to the right is compressed. Objects such as transistors do not change in size.

Plowing is so named because each of the edges caught by the plow can cause edges in front of it to move in order to maintain connectivity and layout-rule correctness. These edges can cause still others to be moved out of the way, recursively, until no further edges need be moved. A mound of edges thus builds up in front of the plow in much the same manner as snow builds up on the blade of a snowplow.

Section 2 of this paper discusses plowing in the context of previous work. Sections 3 and 4 introduce the plowing algorithm for a single mask layer. Section 5 extends it to multiple mask layers and hierarchical designs. Finally, Section 6 presents performance measurements and our experience with plowing in the Magic system.

2. Background

VLSI layouts are difficult to modify. Because of this, designers are often committed to the initial choice of implementation, rather than being able to experiment with alternatives. Existing cells often cannot be re-used in subsequent designs because they don't quite fit; it is typically easier to redesign a new cell from scratch than to modify an old one. Bugs in a dense layout are hard to fix, leading to a debugging cycle which can take days.

Many of these difficulties stem from the fact that seemingly small changes to a layout can have disproportionately large effects. Sometimes this is for

electrical reasons. For example, in ratio logic such as nMOS, changes in the size of one transistor may necessitate changes in the sizes of others. However, even purely topological changes—those which preserve the electrical properties of the layout—can require much more work than the size of the change would suggest. As Figure 1 illustrated, merely opening up new space in a layout can cause effects which ripple outward over a much larger area. Rearranging the internal geometry of a cell or modifying the placement of cells in a floor plan can be similarly expensive because of the need to maintain connectivity with the surrounding material.

Previous attempts to cope with the re-arrangement problem have used symbolic design or sticks [RBDD 83, West 81, Will 78]. In the symbolic/sticks approach, designers enter layouts in an abstract form containing zero-width wires, contacts, and transistors. The sticks form is then run through a compactor to generate actual mask information. As part of the compaction, the circuit elements are moved as close together as the layout rules permit. In a sticks design style, cells can be designed loosely without worrying about exact spacings, since the spacings will be determined by the compactor. However, it is not necessarily easy to make major changes to a sticks cell once it has been entered. Virtual grid systems like Mulga and VTVID provide mechanisms for adding new grid lines uniformly across a cell, but it is still difficult to make large topological changes.

The plowing approach has all the advantages of sticks. It allows cells to be designed loosely and then compacted. In addition, plowing can be used to rearrange cells or open up new space, either across the whole cell or in one small portion. Small changes can be made in one area without having to recompact the entire cell (a global recompaction may potentially shift every geometry in the cell). The plowing approach lets the designer see the final sizes and locations of all objects as he is editing; in the sticks approach, it is hard to predict the final structure of a cell from its abstract form, so compaction must be used frequently to see the results of a change to the sticks.

3. Simple plowing algorithm

Plowing works by finding *edges* and moving them. An edge is a boundary, parallel to the plow, between material of two different types. When an edge moves, the material to its left is stretched, and the material to its right is compressed. In this section we will describe how plowing works when only a single mask layer is present. This material will be assumed to have a minimum width of w, and a minimum separation of s. Edges will always be boundaries between this material and "empty" space.

The fundamental step in plowing is to move a single edge. This step involves determining which other edges must move as a consequence of this motion. The following discussion presents plowing as though it moves a given

edge by first recursively sweeping all other edges out of its way, and then sliding the edge into the newly opened space. Section 4 will present a better scheme for ordering edge motions than this depth-first recursion.

3.1. Finding edges

Figure 3 depicts a trivial layout consisting of three unconnected pieces of diffusion. The edge labelled *e* is to be moved to a final position indicated by the arrowhead. This could be either because *e* was caught by the plow, or because it is being moved to make room for some edge to its left. At a very minimum, the rectangular area labelled *A* must be swept clear of any material before the edge can be moved. However, because of the spacing rule, any material inside area *B* would then be too close to the newly moved edge. Con-

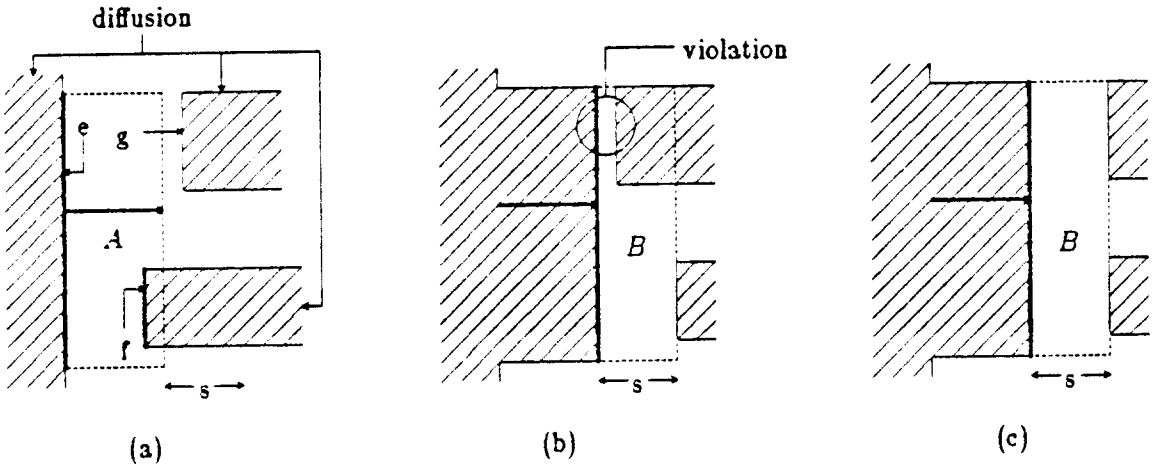

Figure 3. When the edge *e* moves, all edges in area *A* (the area swept out by *e*) must be moved (a). Moving only these edges results in edge *f* moving but not edge *g*. This leaves a layout-rule violation (b) between *e* and *g*. Searching area *B* as well as area *A* avoids this problem. The two areas are referred to collectively as the *umbra* of edge *e*.

sequently, the area to be swept includes both areas A and B. The union of these two areas is referred to as the *umbra* of the edge e*.

Plowing must also search above and below the umbra to prevent the edge from sliding too close to other edges above or below it. Figure 4a shows why this is necessary. If material were moved out of the umbra alone, as in Figure 4b, the result is electrical disconnection. To avoid this, plowing must also move edges out of the areas above and below the umbra. The correct result is

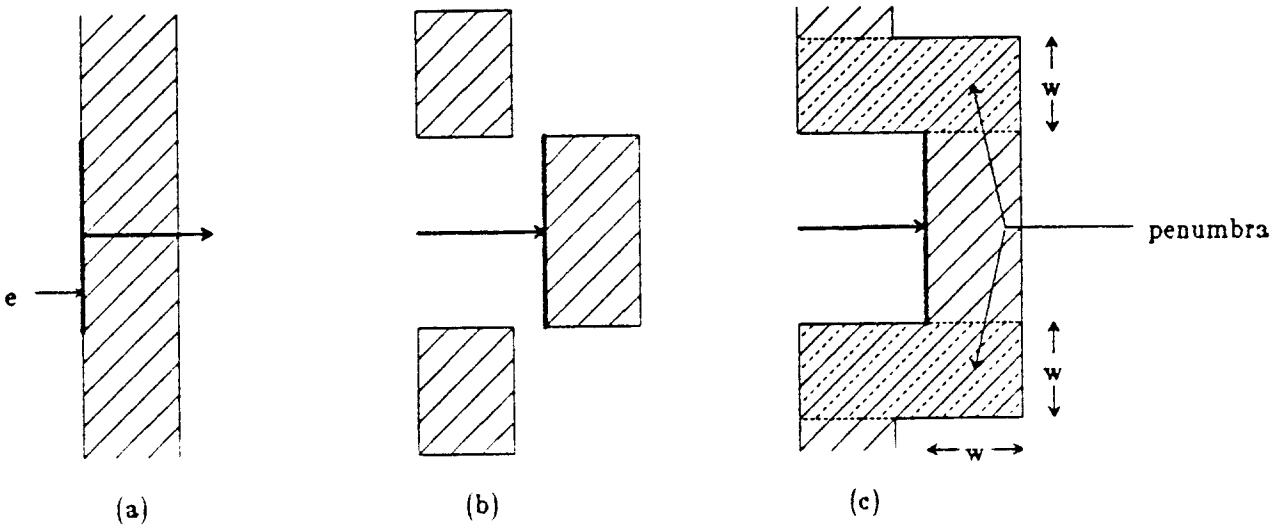

Figure 4. When the edge e moves (a), edges in its umbra must be moved to the right. If only edges in the umbra are moved, however, the result can be electrical disconnection (b). To avoid this, plowing also moves edges in the *penumbra* to the right, giving the correct result shown in (c). This has the effect of inserting jogs automatically. The height of the penumbra is w, the minimum-width for diffusion. If diffusion had been to the left of e instead of to the right, the height of the penumbra would have been s, minimum-separation.

* In a solar eclipse, the *umbra* is that portion of the moon's shadow from which the sun appears to be completely eclipsed. The *penumbra* is the part of the shadow surrounding the umbra from which the sun appears only partially eclipsed. In plowing, the umbra contains edges directly in the path of an edge being moved, while the penumbra contains edges not in the path but nonetheless too close.

shown in Figure 4c. The areas above and below the umbra are referred to collectively as the *penumbra*. Jog insertion is an automatic consequence of searching the penumbra. Moving edges out of the penumbra also prevents electrical shorts, as can be seen by reversing the roles of material and space in Figures 4a-4c.

The left-hand boundary of the penumbra is not always aligned with the edge being moved. Instead, this boundary is formed by following the outline

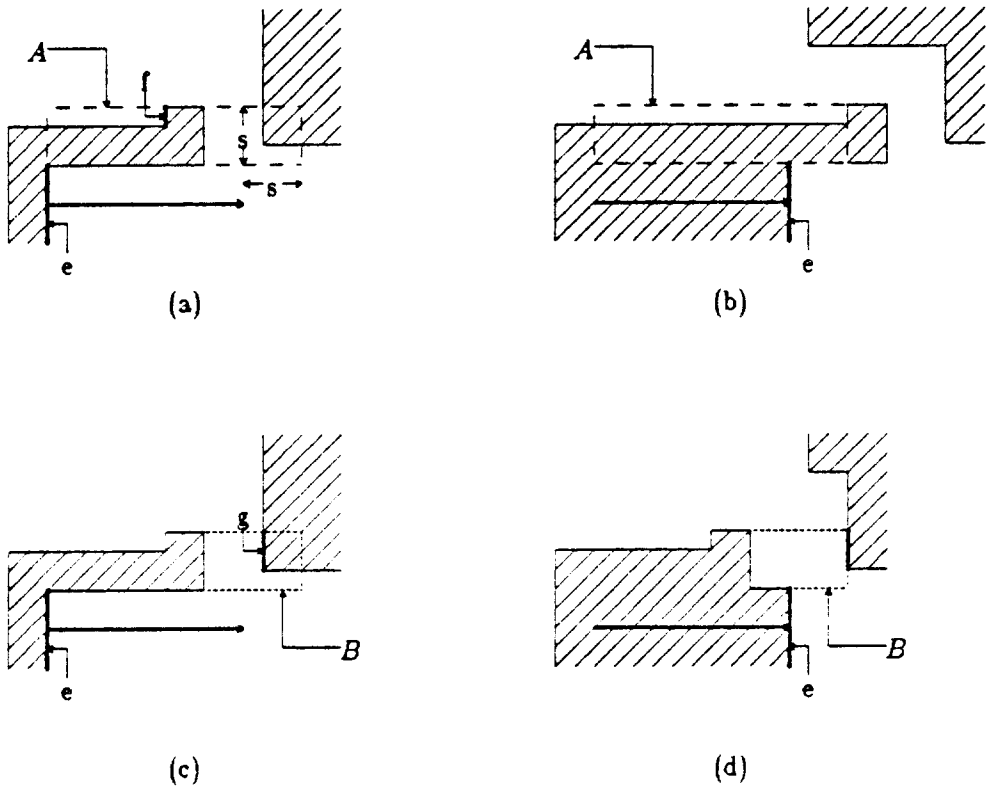

Figure 5. If *e*'s penumbra included all of area *A*, as shown in (a), then edge *f* would be found and moved, resulting in (b). This is undesirable, since *f* need not move in order to preserve layout-rule correctness and connectivity. A better definition of the penumbra would be area *B* only, as shown in (c). Searching this area would result in only the edge *g* being found and moved, as is necessary to preserve layout rule correctness.

of the material forming the edge, as illustrated in Figure 5. This insures that the penumbra contains only those edges which must move in order to preserve layout rule correctness and connectivity. The umbra and penumbra of an edge are collectively referred to as its *shadow*. The shadow of *e* contains all the edges which must move as a direct consequence of moving *e*.

3.2. Sliver prevention

The rules described in Section 3.1 guarantee that plowing never moves one vertical edge too close to another. However, they do allow violations to be introduced between horizontal segments that are formed when material is stretched. These violations take the form of slivers of material or space whose height is less than the minimum allowed. Eliminating such slivers requires that their left-hand edges be moved, as illustrated in Figure 6. The left-hand edge of each sliver lies along the left-hand boundary of the penumbra, so it can be found when tracing the outline of the penumbra.

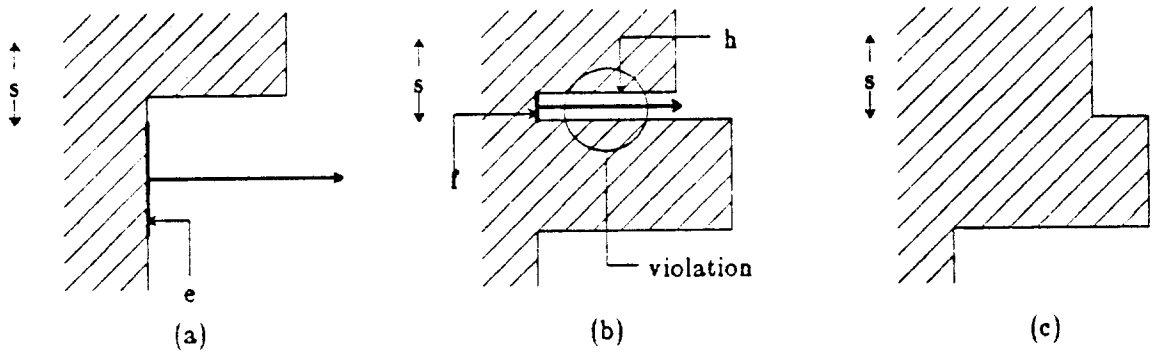

Figure 6. When the edge *e* moves (a), a sliver of space is introduced below the horizontal segment *h*, as shown in (b). To correct this, the left-hand edge of this sliver, *f*, is moved along with *e*, but only as far as the right-hand end of the segment *h* (c).

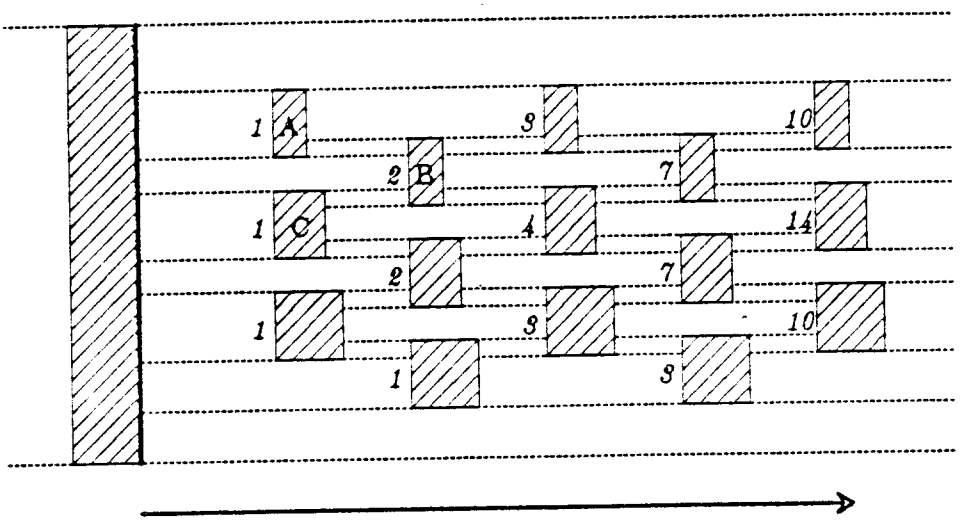

Figure 7. This lattice structure causes exponential worst-case behavior in the depth-first plowing algorithm when edges in the shadow are processed from top to bottom. The objects (*A*, *B*, etc.) must be incompressible to cause this worst-case behavior. Object *B* is moved once when object *A* moves, then slightly farther when object *C* moves. The numbers to the left of each object show how many times each of its edges is moved.

4. Breadth-first vs. Depth-first Search

In the previous section, plowing was described as a depth-first search in which all edges to the right of a given edge were moved before the edge itself. While this approach is conceptually clear, it has poor worst-case behavior. An N-tier lattice structure as illustrated in Figure 7 requires on the order of 2^N edge motions, because plowing performs the recursive search to the right of an edge each time the edge is moved. If, as in the example, each edge must be moved once for each of its two neighbors to the left, the edges at the right-hand side of the lattice are moved a number of times that is exponential in the number of tiers.

Instead, plowing waits until the final position of an edge is known before it performs the search to the right of that edge. This strategy causes the number of edge motions to be linear in the number of edges in the lattice. (A detailed explanation is given in [Oust 84].)

A simple way to insure that edges are moved only once their final positions are known is to use breadth-first search. Magic maintains a list of edges to be moved, sorted in order of increasing x-coordinate. On each iteration, the leftmost edge is removed from the list and the shadow to its right is searched. Any edges discovered by this search are placed in the list along with the amount they must move. Since the final position of an edge can only be affected by edges to its left, the final position of the leftmost edge in the list is always known.

The depth-first algorithm allowed the layout to be modified incrementally as plowing progressed, since an edge was never moved until the area into which it was moving had been cleared. Incremental modification is impossible with breadth-first search, since edges to the right will not be moved as long as there are queued edges to the left of them waiting to be moved. Instead of actually updating the layout as it progresses, the breadth-first version of plowing stores with each vertical edge segment the distance it moves. When the shadows of all edges have been searched, and the distance each edge moves has been determined, plowing invokes a post-pass to update the layout from

the information stored with each edge.

However, if the layout is not modified until all edges have been processed, special care must be taken to avoid the generation of slivers. Figure 8 illustrates the problem. To process each edge correctly, it is important to know what other edges have been already been processed and what their final positions will be. In general, the plowing algorithm must consider edges whose final positions will be in the shadow, rather than those whose initial positions are in the shadow.

The success of the breadth-first algorithm depends on the fact that left-to-right plowing never changes the order of edges along any horizontal line, and never changes any vertical coordinates. Furthermore, edge has stored

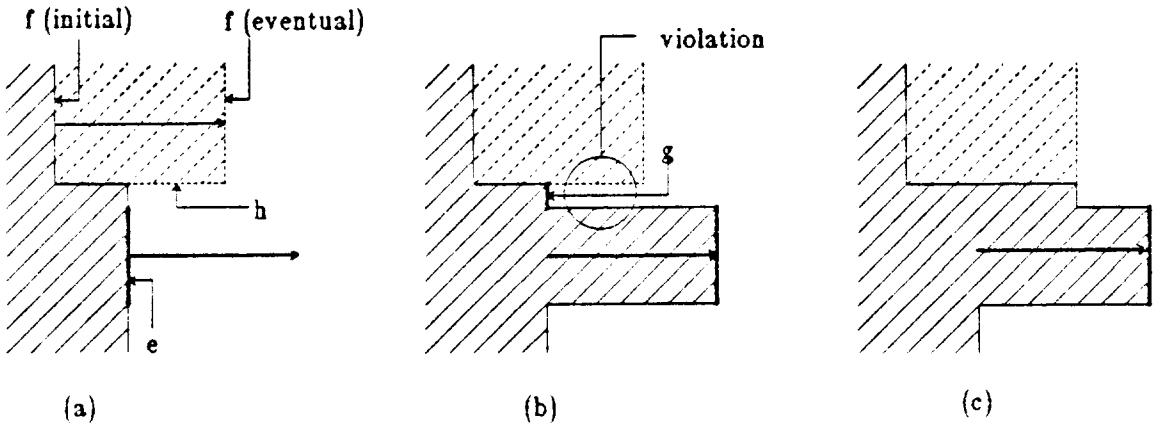

Figure 8. When processing an edge in the breadth-first approach, it is important to use information about the final positions of edges that have already been processed. In (a), it has already been decided to move edge *f*, but the edge will not actually be moved until all other edges have been processed. If edge *e* is processed without considering the new position of *f*, a sliver will result as shown in (b). Instead, the plowing algorithm must consider the eventual positions of edges that have already been processed, to produce the result of (c).

with it the distance it is going to move. As a consequence, plowing can use the initial layout structure for searching, and yet can easily find all objects whose final coordinates fall in a given area.

5. Extensions for real layouts

This section extends the simple plowing algorithm of the previous two sections to handle multiple mask layers. Plowing is also extended to handle features, such as transistors and contacts, whose size should not be changed, and to allow noninteracting mask layers, such as metal and polysilicon, to slide past each other. Finally, since layouts in Magic may be hierarchical, this section closes with a description of how plowing handles hierarchy.

5.1. Multiple mask layers

The simple version of plowing assumed that the shadow extended to the right of the final position of a moving edge by either w (the minimum-width rule) if material lay to the right of the edge, or s (the minimum-separation rule) if material lay to the left of the edge. This insured that the shadow included all edges directly in the path of the edge being moved. Since the same layout rule applied between the edge being moved and any other edge, all edges found during the search of the shadow would have to move.

With more than one mask layer there may be more than one layout rule to apply for a given edge. For example, in our nMOS process, the minimum

separation between diffusion and polysilicon is 2 microns, while that between two pieces of diffusion is 6 microns. Both of these rules apply at an edge between diffusion and empty space.

To insure that the shadow contains all edges which must move, the shadow must extend beyond the area the edge sweeps out by the worst-case layout rule distance applying to that edge. As Figure 9 illustrates, however, not all of the edges found in the shadow search will actually need to move. Each edge found must be checked for its minimum allowable separation from the edge being moved. Fortunately, this can be done very quickly using the same techniques as those used in Magic's incremental layout-rule checker [TaOu 84].

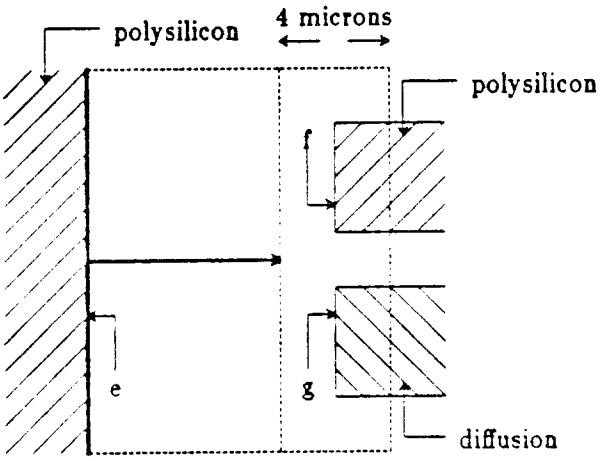

Figure 9. The area of a shadow search is determined by the worst-case layout rule. However, not all edges in that area will have to be moved. Edge *f* must move, because the separation between two polysilicon features must be 4 microns and edge *e* approaches to within 2 microns of *f*. Edge *g* need not move since the minimum separation between polysilicon and diffusion is only 2 microns.

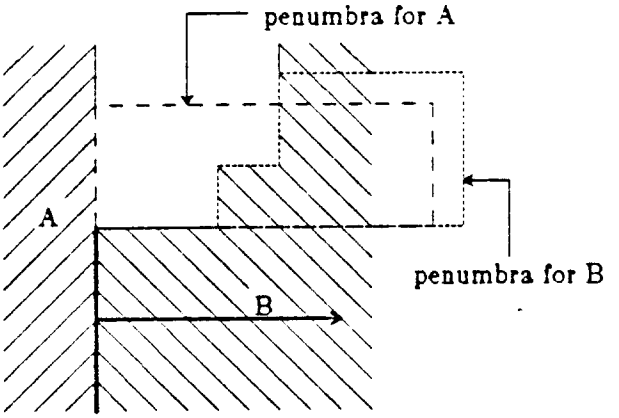

Figure 10. An edge between two different types of material has a penumbra for each. The spacing rules for material of type *A* are applied in *A*'s penumbra. The minimum-width rule for material of type *B* is applied in *B*'s penumbra. The sizes of each penumbra may be different because of the different layout rules applied in each.

If the edge being moved has material on both sides, there is really a penumbra for each type of material. The layout rules applied while searching each penumbra will in general be different. Slivers must be prevented along the boundaries of both penumbra. See Figure 10 for an example.

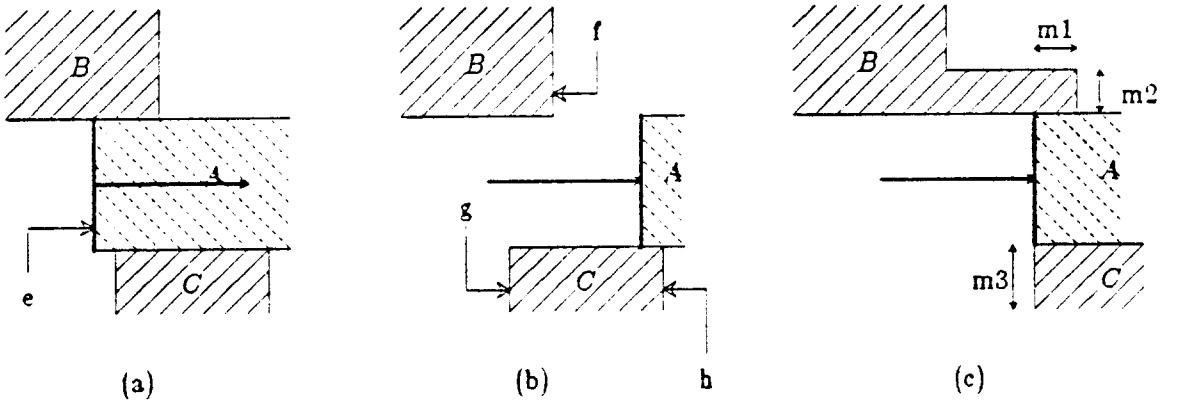

Figure 11. If edge *e* is plowed, material *A* may disconnect from *B* and *C*. To prevent this, a minimum-width segment of edges *f* and *g* is dragged along with *e*. The edge *g* is moved not to maintain connectivity (which would have been achieved by moving *h*), but to prevent *C* from being uncovered. In (c), *m1* is the lesser of the minimum widths for *A* and *B*, *m2* is the minimum width for *B*, and *m3* is the minimum width for *C*.

Multiple mask layers require extra caution to maintain connectivity with material above and below an edge being moved. In the single-layer scheme, the penumbra search guarantees that the material does not become disconnected. However, the penumbra search follows the outline of a single type of material, so it will not by itself guarantee that two adjacent materials of different types will remain connected (see Figure 11).

Special actions must be taken during the penumbra search to handle horizontal edges between different materials. First, if two materials share a horizontal edge, then Magic guarantees that one material does not slide past the end of the other: it maintains a minimum-width connection between the two (this is the case between materials A and B in Figure 11). Second, if one material completely covers the edge with another material (for example, the *A-C* edge in Figure 11), Magic plows the other material as much as is needed to maintain complete coverage. This ensures, for example, that transistors don't get uncovered by plowing polysilicon off one side.

5.2. Inelastic features

Certain features in a layout should not be stretched or compacted. Transistors, for example, have sizes chosen for electrical reasons, as do contacts. Our discussion of edge motion has assumed that the material forming both sides of the edge was stretchable. When material is inelastic, both its left-hand and right-hand edges must be moved in tandem. In particular, if the

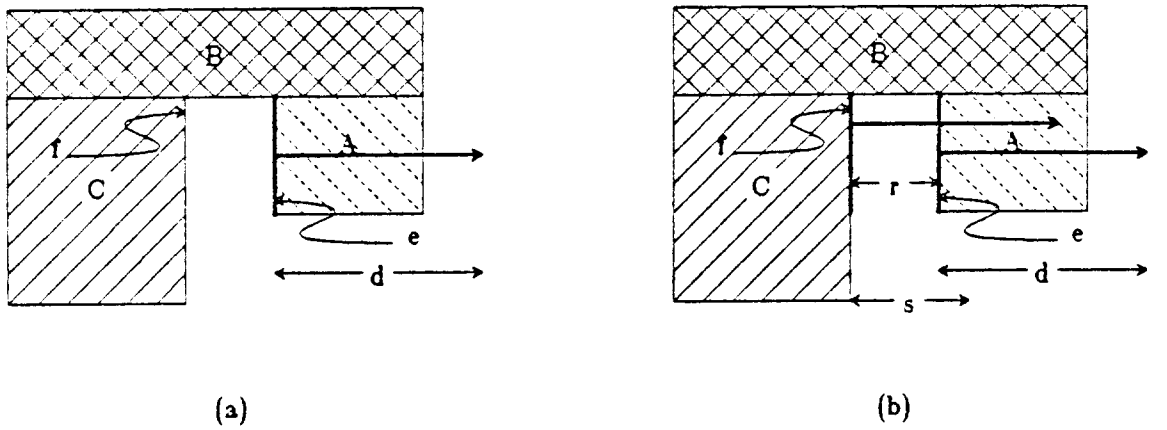

Figure 12. When inelastic objects are present, plowing may have to cope with circular dependencies. Material B is inelastic, and A and C are both minimum-width. When edge e moves by distance d in (a), object B must move by the same distance to prevent A from being uncovered. To prevent C from being uncovered, C's left-hand edge must move, finally causing edge f to move by distance d. Edge e is in f's shadow as a result, but should not be moved a second time.

right-hand edge of a piece of inelastic material moves, its left-hand edge must move also.

A consequence of inelasticity is that moving an edge can cause motion of edges to its left, possibly resulting in a circular dependency. The example in Figure 12 illustrates such a dependency. The depth-first plowing algorithm is completely incapable of resolving such a dependency. The breadth-first algorithm resolves it by comparing the amount an edge is supposed to move with the motion distance already stored with the edge. If the stored motion distance is greater, the edge need not be moved a second time.

If the distance d between edges f and e in Figure 12 is less than s, the minimum separation allowed (ie, there is currently a layout rule violation), looking at the motion distance of e is insufficient. When the shadow of f is

searched, plowing is supposed to move all edges found far enough away so that they cause no rule violations with the newly moved *f*. This would mean that edge *e* would have to move by *d+s-r*, which is more than the motion distance stored with the edge. As a result, the plowing algorithm loops infinitely, each time moving edge *e* by an additional *s-r*. To avoid infinite walks, plowing never moves a shadowed edge (eg, *e*) more than the edge causing the shadow (eg, *f*). This technique prevents infinite looping, but preserves layout rule violations existing in the original layout.

5.3. Noninteracting planes

Section 4 explained that the order of vertical edges along a horizontal line is unchanged by plowing. Thus material being plowed can never slide over other material in its path. There are cases, however, where it is desirable that certain materials in a layout move independently. Metal, for example, does not interact with either polysilicon or diffusion except at contacts, so it should be able to slide over them.

To allow sliding, Magic segregates the mask information in a layout into a collection of non-interacting *planes*. Material in one plane is free to slide past material in any other plane. The nMOS technology, for example, has two planes: one to hold metal wires, and one to hold polysilicon, diffusion, and transistors.

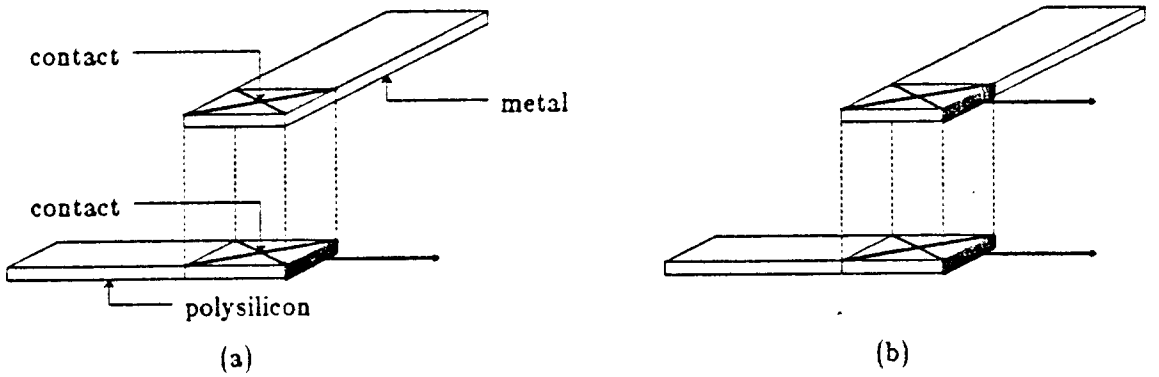

Figure 13. A contact is duplicated on each plane it connects. When an edge of a contact is moved on one plane, it is moved on all other planes as well.

The plowing algorithm operates on each plane independently. The only interaction between planes occurs at contacts, which are duplicated in each plane that they connect. When an edge of a contact is moved in one plane, the corresponding edge of the contact in all other planes is moved by the same amount, as illustrated in Figure 13. This also moves whatever the contact connects to in the other planes, thus preserving connectivity.

5.4. Subcells and hierarchy

One approach for plowing a hierarchical layout, such as that shown in Figure 14a, is to treat it as though it were non-hierarchical and propagate edge motions inside subcells. This might be workable when no subcell is used more than once. However, Magic instantiates subcells by reference, so a change in one instance of a subcell is reflected in all its other uses. Situations in which a subcell is used more than once can produce unsatisfiable sets of constraints, as Figure 14b illustrates.

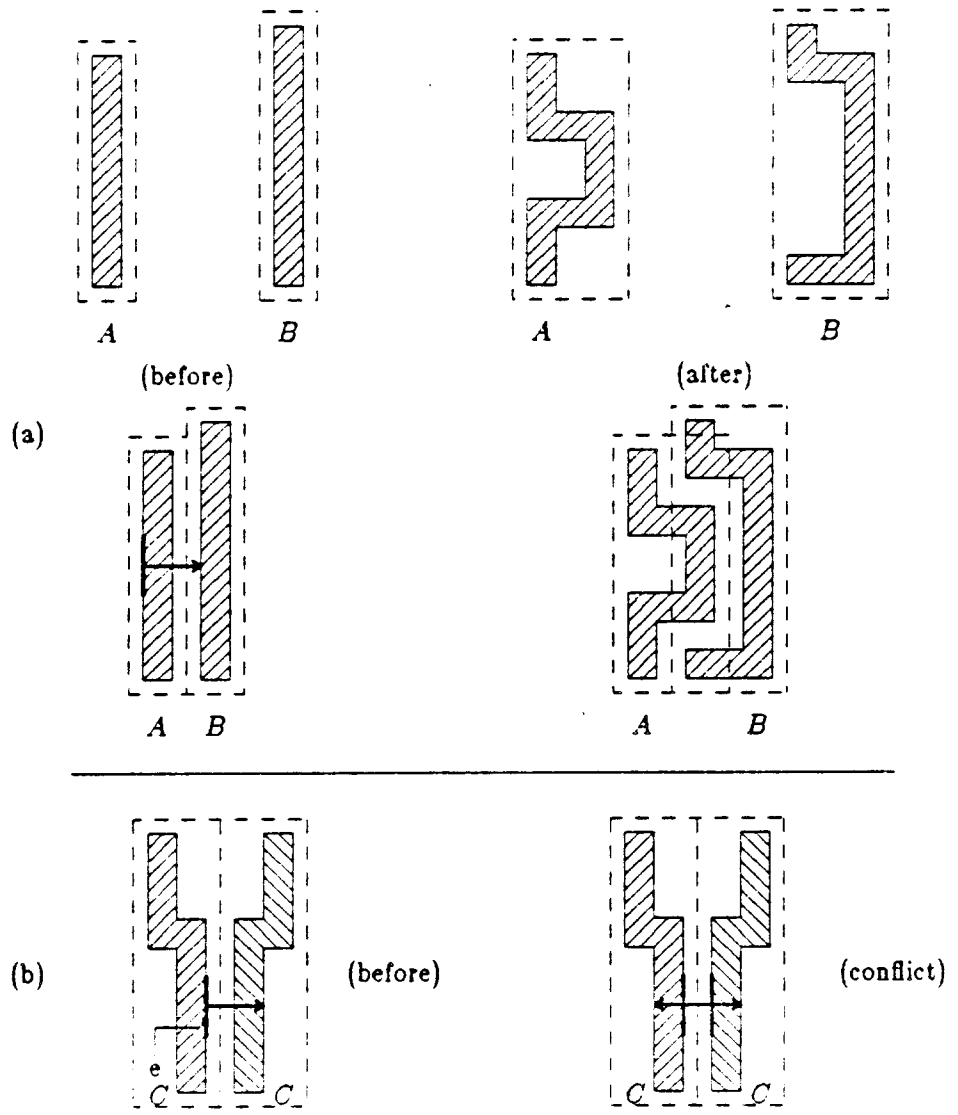

Figure 14. Plowing in the presence of hierarchy. (a) Plowing might treat hierarchy as though it were invisible to the user. Each of cells A and B would be modified. (b) Cell C is used twice, once flipped left-to-right and once in its normal orientation. Both uses refer to the same master definition of C. Moving edge e to the right is impossible, because it requires e to move to the left in order to keep out of its own path. The more edge e is moved to the right in the left-hand use, the worse the violation becomes.

Magic takes a simpler approach, which is to view subcells as black boxes to which connectivity must be maintained by plowing, but whose internal structure should not be modified. A consequence of Magic's approach is that

plowing can be used to modify the placement of cells at the floor plan of a chip, since it only changes the location of subcells, not their contents.

When any mask geometry that abuts or overlaps a cell is moved, the entire cell must move by the same amount. Conversely, whenever a subcell moves, all mask geometry and other subcells that abut or overlap it must also move by the same amount. The net effect is that a cell behaves like flypaper, causing all geometry over its area to "stick" to it and move as a whole when any part of it is required to move.

In addition to preserving connectivity with subcells, when plowing moves other geometry it must avoid introducing any lahout rule violations with the geometry inside a subcell. One approach for dealing with this is to define a *protection frame* [Kell 82] for each cell, an outline around the cell into which no material may be plowed. Magic uses an extremely simple form of protection frame: it assumes that the cell contains all types of material right up to the border of its bounding box.

For example, in our nMOS rule set, the worst-case layout rule involving diffusion is the diffusion-diffusion spacing rule of 6 microns. An edge with diffusion to its left can be plowed to within 6 microns of a subcell before that subcell will itself have to move. The worst-case rule distance involving polysilicon is 8 microns, so polysilicon can only be plowed to within 8 microns of a subcell before the cell must move. Since the contents of subcells are con-

sidered unknown, the closest one subcell can be plowed to another before the other will have to move is the worst-case layout rule in the entire ruleset, which in our ruleset is 8 microns. Of course, if the user wishes to overlap two cells, he can still do that using other editing operations beside plowing.

6. Results and experience

Plowing has been implemented as part of the Magic VLSI layout system. It is written in C under the Berkeley 4.2 Unix operating system for VAXes. A simplified version of plowing (corresponding to that described in Sections 3 and 4) has been operational since October of 1983.

While the full implementation of plowing has not been completed, measurements on the simple version indicate that it is fast enough to be used interactively. An example similar to that presented in Figure 1a, consisting of 48 parallel bars of polysilicon each separated by 4 microns (the minimum separation), took 3.2 seconds of VAX-11/780 CPU time to produce a result similar to that in Figure 1b. Only 1.0 seconds were spent computing the edge motions; the remainder of the time was spent in the post-pass which actually updates the layout.

7. Acknowledgements

Gordon Hamachi, Robert N. Mayo, and George Taylor all contributed to the discussions out of which the plowing algorithm arose. In addition to the above people, Randy Katz, Ken Keller, and Steve and Jean McGrogan all provided helpful comments on early drafts of this paper.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00034-K-0251

8. References

[RBDD 83] Rosenberg, J., Boyer, D., Dallen, J., Daniel, S., Poirier, C., Poulton, J., Rogers, D., Weste, N. "A Vertically Integrated VLSI Design Environment." *Proceedings, 20th Design Automation Conference*, 1983, pp. 31-38.

[Kell 82] Keller, K., Newton, A. "A Symbolic Design System for Integrated Circuits." *Proceedings of the 19th Design Automation Conference*, June 1982.

[Oust 81] Ousterhout, J.K. "Caesar: An Interactive Editor for VLSI." *VLSI Design*, Vol. II. No. 4, Fourth Quarter 1981, pp. 34-38.

[Oust 84] Ousterhout, J.K. "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools." To appear in *IEEE Transactions on CAD/ICAS*, Vol 3, No. 1, January 1984.

[OHMST 84] Ousterhout, J.K., Hamachi, G., Mayo, R.N., Scott, W.S., and Taylor, G.S. "The Magic VLSI Layout System." In this technical report.

[TaOu 84] Taylor, G.S., and Ousterhout, J.K. "Magic's Incremental Design Rule Checker." In this technical report.

[West 81] Weste, Neil. "Virtual Grid Symbolic Layout." *Proceedings, 18th Design Automation Conference*, 1981, pp. 225-233.

[Will 78] Williams, J. "STICKS- A Graphical Compiler for High Level LSI Design." *Proceedings of the 1978 NCC*, May 1978, pp. 289-295.