A Building Block Placement Tool

Jonathan Dufour
Cadence Design Systems
San Jose, CA 95134
Tel: 408-944-7037
email: jdufour@cadence.com

Robert McBride
Hughes Microelectronics Div.
Newport Beach, CA 92658
Tel: 714-759-2009
email:
mcbride@newport.hac.com

Ping Zhang
Dept. of Comp. Sci. & Engr.
University of California,
San Diego
La Jolla, CA 92093-0114
email: zhangp@cs.ucsd.edu

Chung-Kuan Cheng
Dept. of Comp. Sci. & Engr.
University of California,
San Diego
La Jolla, CA 92093-0114
email: kuan@cs.ucsd.edu

Abstract— When designing integrated circuits, sub-components rarely end up being perfectly rectangular. However, currently most block-placers only consider rectangular components, resulting in inefficient area utilization. We propose a placement tool that allows arbitrarily sized and shaped convex components. It extends the rectangle-packing method proposed by Kajitani. We describe the methods used to create the placement and give some performance results.

I. Introduction

A. Introduction

When designing integrated circuits, sub-components rarely end up being perfectly rectangular. However, currently most block-placers only consider rectangular components, resulting in inefficient area utilization. We propose a placement tool that allows arbitrarily sized and shaped convex components. It extends the rectangle-packing method proposed by Kajitani. We describe the methods used to create the placement and give some performance results.

B. Building Block Placement – Problem Description

The inputs of the problem are:

- a set of blocks which have a fixed geometry and fixed pin locations
- a netlist specifying the interconnections between the pins of each block
- constraints on the locations of external pads
- constraints on the size and/or aspect ratio of the chip
- constraints on critical nets.

The problem objective is to determine the location and orientation of all blocks such that the area of the chip is minimized and the design is routable. Furthermore, all design constraints must be satisfied. The constraints on critical nets can be expressed in terms of minimum or maximum path lengths.

C. Previous Approaches to Building Block Layout

Many approaches to building block placement have been proposed. Constructive and force-directed placement approaches are described in [1]. Kuh describes a pattern based approach for placement [2][3]. Onodera describes a branch and bound based placement approach [4]. Sechen's TimberwolfMC placement and global routing tool was one of the first tools based on simulated annealing [5]. Lastly, one of the most recent proposals to placement was a rectangle packing approach described by Kajitani [6]. Each method has its merits, but so far, no method has emerged as the ideal solution. It should be noted that this is by no means a complete list of all approaches. Rather, it is intended to give a flavor of the various approaches to the problem.

II. Methodology

A. Overview

We extend the method described by Kajitani to work with general convex rectilinear components. By convex, we mean that any point within the component can be connected to any other point within the component via a single horizontal and vertical line, each of which is also contained by the component. So, for example, the component in Figure 1.a is convex; the one in Figure1.b is not.

Fig. 1. Convex vs. concave components. a.) convex component. b.) concave component

The general idea behind Kajitani's approach is to first place the components randomly on a grid, and then to use a longest path algorithm to estimate the area required by the compacted placement. To determine component placement, two component sequences called *loci* are derived. These correspond to horizontal and vertical gridlines in the grid. For a more detailed description, the reader is referred to [6].

Kajitani's approach is attractive for a number of reasons. First, the method of changing the configuration is extremely simple. To change the location of a component, we merely permute the two loci. This makes the algorithm very simple to implement using techniques such as simulated annealing. Second, components are not allowed to overlap. This means that it is not possible to generate illegal

ASP-DAC '97
0-89791-851-7$5.00

configurations, though unroutable configurations may still arise.

Our methodology is fairly similar to that of Kajitani. We first generate an initial topological arrangement of the components, then derive two constraint graphs for the arrangement. Finally, like Kajitani, we determine the area required by the final layout by calculating the longest path in the constraint graphs.

B. Creating the Initial Layout

Kajitani's method of using the loci to specify a layout seems very attractive due to its simplicity. Unfortunately, it lacks the flexibility necessary to specify all of the relationships between arbitrary convex rectilinear components. For any two components, Kajitani's method allows us to specify four relationships: *left-to*, *right-to*, *bottom-of*, and *top-of*. However, for non-rectangular components, many more than four relationships can exist, as illustrated in Figure 2.

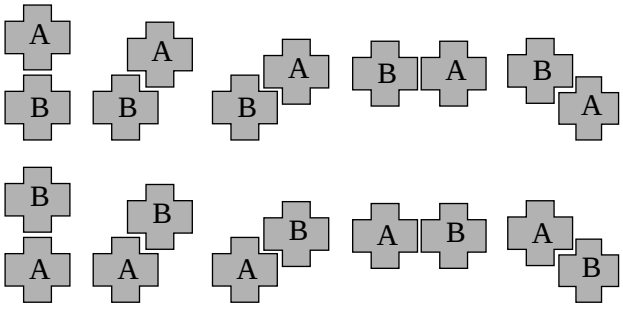

Fig. 2. The ten possible relationships for components A and B.

Consequently, we use a more concrete approach.

We first place the components randomly on a grid. The main purpose of this placement is to allow us to determine the relationships between the components. Using these relationships, we will later construct constraint graphs, which we will use to determine the final layout.

We chose to use the grid primarily because of its simplicity. The use of a grid facilitates determining component proximity. Furthermore, the grid makes overlap detection trivial. When placing a component, we merely check the grid squares that the component will occupy to see if they are empty. This can obviously be a fairly costly procedure if the grid cells are not sized well. However, we felt this cost was outweighed by the ease of implementation.

C. Modifying the Placement

The initial placement is improved iteratively through the use of simulated annealing. Each new configuration is obtained by changing a single component in the current configuration. The two types of refinements allowed are moving a component and changing a component's orientation. The choice to move a component is made with slightly higher probability than the choice to rotate a component.

When moving components, we must consider two factors: which components to move, and where to move them. We implemented two methods for selecting which components to move. The first and simplest method chooses components at random from the entire list of components in the design. This method attempts to minimize the interconnect length and area for all components in the design. The second method moves only those components on the critical path for the overall chip width or height. This method is specifically geared toward reducing the area required by the chip by moving cells from congested areas to less congested areas. It is fairly easy to determine which components lie on the critical path, given our method of cost estimation and determining the final placement. These will be discussed in more detail in the following sections.

The component selection method used depends upon the area and interconnect length weights specified by the user. These weights are used in the cost function and indicate which the user values more highly, interconnect length or area. If the area is given a greater weight, then the second method will be chosen proportionally more.

The location to which a component is moved is generated randomly. We then attempt to place the component at that position. If another component is encountered, the move is illegal, so another location is generated. Consequently, if the grid is too small, collisions will occur frequently, so the algorithm will waste a lot of time searching for a feasible placement location. If, after twenty attempts, we have not successfully placed the component, we return the component to its original location.

Rotation of components is performed analogously to movement. Each component has 8 possible orientations. The program generates a new orientation randomly, and attempts to place the component in that orientation. If another component is encountered, a new orientation is generated and the procedure repeats until the component is successfully placed.

D. Cost Estimation

The cost estimate for a placement has two components, a layout area term, and an estimated interconnect length term. The user is allowed to weight these terms, and the program will attempt to take into account these weights when deriving the placement.

The layout area term indicates the total area required by the layout. It is determined by calculating the final position of all the components and finding the area of the layout's bounding box. The way in which we determine the component positions is similar to the method presented in [6], though we must make some adjustments to account for the general convex rectilinear shape of the components. As in Kajitani's approach, we construct two constraint graphs –

one for vertical constraints and one for horizontal constraints. We then use the longest paths algorithm for vertex-weighted directed acyclic graphs to independently determine the *x* and *y* locations of the components. Constructing these constraint graphs will be discussed in detail in a later section.

The second term for cost estimation is the total estimated interconnect length. For each placement, we evaluate the estimated length of each net. For simple two-terminal nets, we estimate the net length as one-half of the perimeter of the bounding box of the net. For multi-pin nets, we attempt to estimate the interconnect length by calculating the total length of the clique containing the nodes and taking a fraction of this proportional to the number of nets needed. Obviously, this will not yield the exact interconnect length needed, but it works adequately.

E. Determining Component Positions

We determine component positions in much the same way as [6]. From the gridded layout, we construct two constraint graphs – one for vertical constraints and one for horizontal constraints. Using these constraint graphs, we can independently determine each component's *x* and *y* positions by calculating the longest path to the component through the constraint graphs. We do not leave room for routing channels but assume each components' placement outline can be expanded proportional to the routing area needed.

To construct the constraint graphs, we perform a left-to-right sweep of the grid. For each component, we extend probes at a 45 degree angle in the upper- and lower-left directions. The probes are extended only from exterior corner points. For example, for the given component we would extend the following probes.

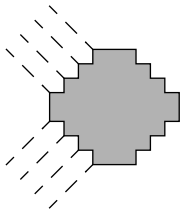

Fig. 3. Extending probes from components.

If these probes encounter another component, then an edge is added between the two components in the appropriate constraint graph. For example, if the probes encounter a horizontal side of another component, then we would add constraints between the components in the vertical constraint graph. If the probes encounter a vertical side of another component, we would add a horizontal constraint.

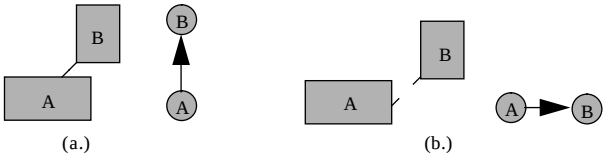

Fig. 4. Constructing the constraint graphs: a.) probe encountering horizontal edge indicates a vertical constraint. b.) probe encountering vertical edge indicates a horizontal constraint.

With each edge, we keep information about the relevant points, specifically, the point from which the probe was extended and the most constraining point on the hit component. We do not need to keep track of the relationship between all pairs of points, just a few key pairs. For the situation in Figure 5, it is sufficient to record that points 3 and 5 of component A are to the left of points 11 and 9 of component B, respectively and that point 11 of component B is above point 5 of component A.

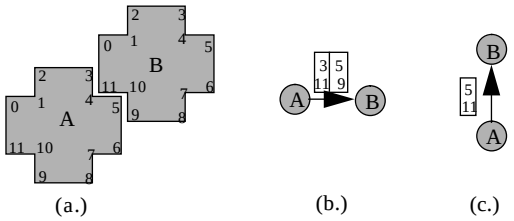

Fig. 5. Constructing the constraint graphs: a.) component arrangement. b.) horizontal constraints. c.) vertical constraints.

For a single probe, we may need to add multiple constraints. For example, in Figure 6, the probe from point 5 of component B encounters the horizontal side (4, 5) of component A, producing a vertical constraint between point 5 of component A and point 5 of component B. However, we must also add a horizontal constraint between point 3 of component A and point 5 of component B, and also between point 7 of component A and point 3 of component B.

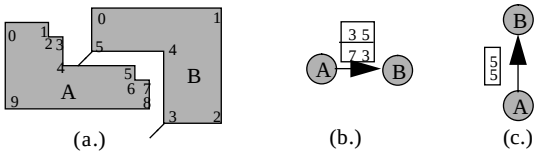

Fig. 6. Constructing the constraint graphs: more detail.

To account for this, we always check the exterior points on either side of where the probe struck the component. We add a horizontal constraint between the left point and the probe point, and a vertical constraint between the right point and the probe point. For example, for the situation in Figure 6, After detecting the vertical constraint between point 5 of component A and point 5 of component B, the algorithm would check to see if component A has an upper-right exterior point to the left of the probe point (point 5 of component B.) In this case, there is (point 3), so a

horizontal edge is added between point 3 of component A and point 5 of component B. Next, the algorithm checks the probe points of component B surrounding the current probe point. In this case, the only other probe point is point 3 of component B. We then find the rightmost upper-right point that is still to the left of point 3. In this case, this would be point 7 of component A. We then add a horizontal constraint between these two points.

To simplify determining which points to check, we maintain a list of traces. These traces are essentially projections of all the exterior corner points onto the positive and negative forty-five degree axes. We maintain four sets of traces, one for upper-left, lower-left, upper-right, and lower-right exterior points. These traces are created at the time the components are initialized and remain static throughout the layout procedure. The main purpose of the traces is to give an idea of the relative positioning between components' points. They are only one-dimensional, which makes them easy to deal with. The example from Figure 6 is given below, showing component A's upper-right traces and component B's lower-right traces. As mentioned previously, to add the constraint between point 7 of component A and point 3 of component B, we find the rightmost upper-right trace of component A that is to the left of the trace for point 3 of component B.

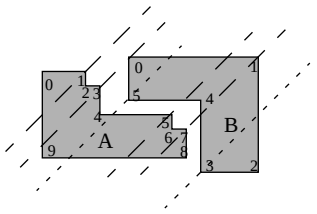

Fig. 7. Using traces to help determine the constraints to add.

Figure 9 gives a brief outline of the code. Section I handles situations like that of Figure 8.a in which the next upper left probe of component *B* (from point 5) would miss component *A*, failing to produce a needed constraint. Similarly, Section II handles the analogous situation in which the lower right probe of component *B* (from point 3) would miss component *A*, as in Figure 8.b.

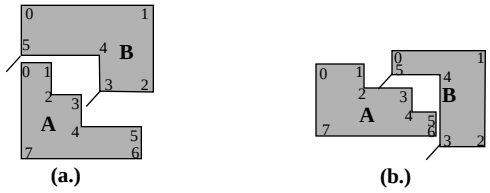

Fig. 8. a.) situation handled by section I of pseudocode. b.) situation handled by section II of pseudocode.

```
AddConstraints(curComponent, hitComponent,hitCompTraceType,
               curCompTrace, curPoint)
{
  // Get points on either side of where the trace hit the component
  hitComponent.GetSurroundingPoints(hitCompTraceType,
                          curCompTrace, &leftPoint, &rightPoint)
  if (leftPoint)
  {
    /* There is a trace on the hit component that bounds the
     * probe we extended out.  Since it is to the left of
     * the probe we extended, that point must be to the left of
     * this one. So add that constraint.                        */
    hitComponent.MakeHorizontalSuccessor(curComponent,
                                         curPoint,leftPoint)

    /* SECTION I: Check for vertical overhang to the left of
     * current trace to see whether we must add a vertical
     * constraints as well. First, get the next trace on the
     * current component above and to the left of current trace*/
    curComponent.GetNextTrace(tracetype, LEFT, &nextTrace,
                              &tracePoint)
    if (nextTrace)
    {
      /* Get point that immediately under/above the nextTrace. */
      hitComponent.GetSurroundingPoints(hitCompTraceType,
                              nextTrace, &unused, &rightPoint2)
      if (rightPoint2)
      {
        if (tracetype == LOWER_LEFT || tracetype == LOWER_RIGHT)
          hitComponent.MakeVerticalSuccessor(curComponent,
                                         tracePoint,rightPoint2)
        else
          curComponent.MakeVerticalSuccessor(hitComponent,
                                         rightPoint2,tracePoint)
      }
    }
  }

  if (rightPoint)
  {
    /* There is a trace on the hit component that bounds the
     * probe we extended out.  Since it is to the right of the
     * probe we extended, that point must be above this one, if
     * the trace type was lower_left or below otherwise.       */
    if (tracetype == LOWER_LEFT)
      hitComponent.MakeVerticalSuccessor(curComponent,curPoint,
                                         rightPoint)
    else
      curComponent.MakeVerticalSuccessor(hitComponent,
                                         rightPoint, curPoint)

    /* Section II: Check to see whether we must add other
     * horizontal constraints as well.  GetNextTrace returns trace
     * below and to the right of current trace.           */
    curComponent.GetNextTrace(tracetype, RIGHT, &nextTrace,
                              &tracePoint);
    if (nextTrace)
    {
      hitComponent.GetSurroundingPoints(hitCompTraceType,
                               nextTrace, &leftPoint, &unused);

      if (leftPoint)
        hitComponent.MakeHorizontalSuccessor(curComponent,
                                       tracePoint, leftPoint)
    }
  } /* if rightpoint */
}
```

Fig. 9. Pseudocode for adding constraints for probes extended left.

It is fairly likely that the probes will not encounter a component with which the current component should have constraints, as in Figure 10.

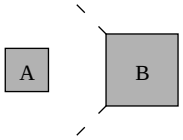

Fig. 10. It is possible for the probes to miss neighboring components.

Consequently, we must also check all of the components to the left of the current component to verify that some relationship is defined between the components in the constraint graph. We do not need to add constraints between every pair of components, but some type of

relationship should exist. For example, we can take advantage of transitivity. If component A is strictly to the left of component B, and component C is strictly to the right of component B, then component C must also be to the right of component A, so no further constraints need to be added to the constraint graph. However, if component A was partially or totally above component B, and component C is to the right of component B, we do not necessarily know any relation between component C and component A, as shown in figure 9.b and 9.c.

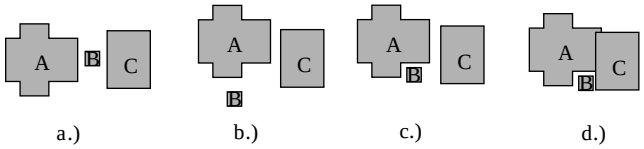

Fig. 11. Adding constraints. a.) transitivity applies; no constraints need to be added. b.) must add constraints. c.) B is to the right of part of A, but we must still add constraints. d.) placement that could occur if we didn't add constraints of c.).

F. Performance Improvements

We can take advantage of the fact that the changes caused by a component's move or rotation are local. Only the area where the component had been located and the component's new area will be affected and must be updated. For large designs, most of the edges in the constraint graph will remain unchanged through a single move. We can take advantage of this fact to improve the performance of the algorithm. Rather than recreating the constraint graphs for each move, we can simply update the appropriate edges.

When repairing the constraint graphs after a move, we must update two areas: the area in which the component was originally located and the area where the component was placed. When updating the area in which the component was originally located, we must update all of the components that had an edge to or from the moved component. We rebuild the constraint graph locally for all of these components. It is not possible for components other than those with edges to the moved component to get corrupted because all other components remain stationary.

To update the constraint graphs, we perform a procedure analogous to that used to create the constraint graphs. We traverse the list of affected nodes from left to right, adding the appropriate constraints between these nodes. As before, for each node, we extend probes in the upper and lower left directions until we hit a component. These constraints are then added to the constraint graph. Again, we must also check the other affected components to ensure that some sort of relationship is defined between them.

We must also update the components in the area to which the component was moved. To do this, we extend probes in all four directions from the moved component to determine its relationship with other components. Again, this is analogous to building the constraint graphs, except that we also extend the probes in the upper and lower right directions. We must also check all the neighboring components to see if we must add a constraint between them.

III. Experimental Results

A. Introduction

We tested the layout tool on several industrial MCM designs. The designs are typically fairly small – each contained about 50 components to be placed. We compare the results produced by the tool against the hand-placed designs.

We placed each design using various weights for interconnect and area, and also using several different values for the inner loop factor. This factor roughly determines how long the annealing process will run and consequently, the quality of the placement. We ran the layout tool 5 times on each design, and average the results for each set of weights. The following table shows the performance results we obtained.

Table 1
Experimental Results

Area Weight	Inter-connect Weight	Inner Loop Factor	Percent Diff. Area	Percent Diff. Inter connect	Execution Time
1	1	2	10.89%	12.32%	414 sec.
1	5	2	14.75%	12.81%	304 sec.
1	10	2	18.24%	16.41%	449 sec.
5	1	2	18.45%	15.32%	325 sec
10	1	2	16.74%	11.91%	409 sec
1	1	4	14.28%	13.13%	870 sec
1	5	4	15.55%	11.91%	807 sec
1	10	4	5.85%	10.81%	939 sec
5	1	4	13.12%	6.97%	493 sec
10	1	4	6.35%	13.15%	1032 sec
1	1	6	6.76%	10.05%	1117 sec
1	5	6	18.03%	13.12%	1310 sec
1	10	6	13.11%	15.23%	1563 sec
5	1	6	10.42	15.06	1367 sec
10	1	6	7.28	9.45	1514 sec

As can be seen from the table, the layouts produced by our tool were consistently inferior to those of the human designers. This is not surprising, since the initial hand-

placed design left little room for improvement. Furthermore, due to the small size of the layouts, a small increase in area would appear more significant than it would for a larger design. Consequently, we would expect that for larger designs, the margin would narrow significantly, since human designers would be overwhelmed by the many design possibilities. Unfortunately, we were unable to find any larger test cases.

The results produced by the algorithm were not entirely consistent, suggesting that more tuning of the simulated annealing algorithm would be useful.

Figure 10 shows a sample placement produced by our tool.

Fig. 12. Sample placement of design 1051703 using an area weight of 1, an interconnect weight of 1, and 6 inner loop iterations.

As can be seen in the figure, our layout tool does achieve a reasonably compact layout.

IV. Conclusions

A. Overview

We have demonstrated the feasibility of extending Kajitani's approach to general convex rectilinear components. We use a simulated annealing approach to place components on a grid, and determine the final placement using the longest paths algorithm. This approach produces decent results in fairly reasonable execution times.

B. Future Work

Although the layout tools works adequately, there are a number of improvements that could be made that would improve performance and the solution quality.

The use of the grid greatly simplifies some aspects of the design. Checking for overlap becomes trivial, and determining components proximity is simplified. However, checking each grid cell can be very costly and time consuming. Consequently, it would be interesting to derive an implementation without using the grid as a basis. This implementation would necessarily be more abstract, but it might be able to achieve better performance than our grid-based approach.

Currently, our approach considers all nets equally. Typically, in designs only a few nets are critical. Consequently, future versions may want to place more emphasis on the critical nets rather than considering all nets equally.

We have not yet tried the tool in conjunction with any router to test routability.

C. Final Comments

The layout tool looks promising. In most cases, it was able to find fairly high-quality solutions in a reasonable amount of time. However, more work would need to be performed on it before it could be used in an industrial setting. One of the greatest concerns involves incorporating more performance information from the designs so that the placement will ensure that all the timing constraints are met. Furthermore, some areas of the design could be speeded up, possibly by the elimination of the grid or some other method.

Acknowledgements

This work was supported in part by grants from the NSF MIP-9529077as well as UC MICRO program.

References

[1] N. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, 1993.

[2] W. Dai, E. Kuh, "Simultaneous Floor Planning and Global Routing for Hierarchical Building-Block Layout," *IEEE Trans. on CAD*, Vol. CAD-6 no. 5, pp. 828-837, 1987.

[3] W. Dai, M. Sato, E. Kuh, "A Dynamic and Efficient Representation of Building-Block Layout," *Proc. 24th ACM/IEEE Design Automation Conf.*, pp. 376-384, 1987.

[4] H. Onodera, Y. Taniguchi, K. Tamaru, "Branch-and-Bound Placement for Building Block Layout," in *Proc. 28th ACM/IEEE Design Automation Conf.*, pp. 433-439, 1991.

[5] C. Sechen, "Placement and global routing of integrated circuits using simulated annealing," Ph.D. Thesis, UC Berkeley, 1987.

[6] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "Rectangle-packing-based module placement," *Proc. IEEE Inteernational Conference on Computer Aided Design*, pp. 472-479, Nov. 1995.