# Deterministic versus Non-Deterministic Placement Algorithms for Automatic Layout Synthesis Tools

Fernando Moraes[1], A. Josep Velasco[2]

[1]FACIN-PUCRS - Av. Ipiranga, 6681 – Prédio 16
90619-900, Porto Alegre, Brazil
moraes@inf.pucrs.br

[2]Departament. Informàtica UAB
08193 Bellaterra, Spain
josep.Velasco@uab.es

## Abstract

In this paper two different approaches applied to cell placement are compared. The first one is a deterministic algorithm, based on a min-cut strategy. The second one is a non-deterministic approach, based on stochastic evolution. The goal of this paper is the comparison of performance of both strategies, studying the silicon area consumption and the CPU requirements, using an automatic layout synthesis tool [1].

## 1 INTRODUCTION

The performance of integrated circuits has been increasing exponentially due to the device and interconnection scaling, new micro-architecture and design methodologies. As the device physical size decreases, the time spent by signals to travel between gates is equivalent or greater than the gate delay for deep-submicron technologies [2]. These parasitic effects and the number of interconnection layers (5 to 6) represent a new challenge for IC design tool developers. In the design flow, the routing length will be a function of the quality of the cell or block placement. In this way, it is necessary to analyze different placement procedures, in order to fix which has the best trade-off for routing length, silicon area and CPU time consumed.

This paper is organized as follows: next Section describes the deterministic approach, a quadrature placement. In Section 3 the non-deterministic approach, based on stochastic evolution is depicted. Results are presented in Section 4. Finally, some conclusions are discussed in Section 5.

## 2 DETERMINISTIC ALGORITHM

The main cost function in an automatic layout synthesis tool is twofold: (*i*) reduce the total routing length and (*ii*) distribute the cells homogeneously in the circuit, in order to have the best equilibrium between horizontal and vertical routing, avoiding congestion areas in one direction.

Among the known algorithms for cell placement [3] (cluster-grow, partition based, simulated annealing, simulate evolution), the most suited for the present problem, resulting in an equilibrium between horizontal and vertical routing without congestion areas, is a partition based algorithm, called quadrature placement.

When a circuit is divided into 2 blocks, the number of common signals between them is called cutsize. The basic algorithm for the quadrature placement is the min-cut (minimum cutsize), presented in [4]. The min-cut procedure divides a circuit into two blocks, having as cost function the reduction of the cutsize, that is, the number of lines crossing partitions.

The quadrature placement alternatively divides the circuit in horizontal and vertical directions, minimizing the cutsize in each direction. Figure 1 illustrates the method. The circuit is initially partitioned in the vertical direction, using the min-cut algorithm, into two blocks with the same area. This first partition reduces the routing density in the middle of the channels, avoiding congestion in these areas. Next, one has two horizontal partitions, where the area in each one is proportional to the number of rows. For example, for 5 rows there might be an area ratio of 3:2, for 4 rows 2:2 and so on. The partitioning procedure stops when no more horizontal partitions are possible (area ratio1:0).

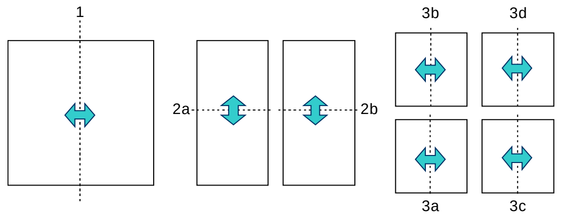

**Figure 1 - Quadrature placement**

This process results in a set of quadrants, with few cells (typically 2-8) in each one. The cells in each quadrant can be placed using a simple algorithm. As an example, cell order can be obtained directly from the connectivity between cells.

To improve the efficiency of the quadrature placement, the pin propagation algorithm [5] must be implemented. Pin propagation tries to place cells with common signals in adjacent quadrants, reducing routing length, and so improving the electrical performance of the circuit. The pin propagation algorithm is our main contribution, since references give no information on how to solve this problem.

The main idea is the following: to make a partition within a quadrant, each already partitioned quadrant is also taken into account. For the horizontal partition, the quadrants are processed line by line, from left to right. For the vertical partition, the quadrants are processed column by column, from bottom to top.

### Horizontal pin propagation

Supposing the program is doing the horizontal partition of the quadrant number ***11*** (Figure 2-b). The signals considered as interface for this quadrant are:

*South pins:*

- south input/output signals in quadrant 11;
- common signals between quadrant 11 and quadrants 1 to 8;
- common signals between the bottom side of quadrants 9 and 10 (already partitioned) and quadrant 11.

*North pins:*

- north input/output signals in quadrant 11;
- common signals between quadrant 11 and quadrants 17 to 32;
- common signals between the top side of quadrants 9 and 10 (already partitioned) and quadrant 11.

The vertical pin propagation (Figure 2-a) is similar to the horizontal one. Instead of south/north interface pins, the east/west external pins are used.

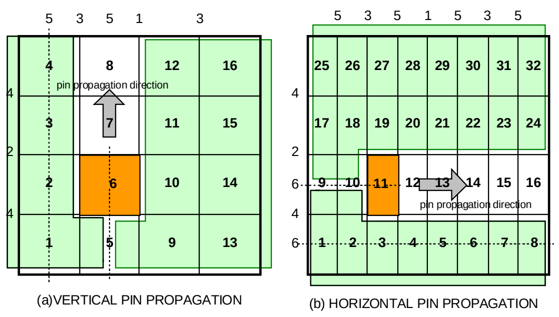
pin propagation direction
(a)VERTICAL PIN PROPAGATION
pin propagation direction
(b) HORIZONTAL PIN PROPAGATION

**Figure 2 - Pin Propagation**

## 2.1 Comparison between deterministic algorithms

This placement algorithm was compared to another deterministic placement procedure (cluster-grow with linear placement [6]). Results indicate a great reduction on the number of routing tracks, since the new method distributes homogeneously the connections in both directions, while the cluster-grow minimizes the cut between rows, congesting the horizontal routing. Table 1 shows the number of routing tracks required to implement a set of benchmark circuits.

Another advantage of quadrature placement is the reduced CPU time required to place the cells. Typically, the quadrature algorithm runs 4 to 6 times faster than the cluster-grow placement.

| Circuit | Number of transistors | T1 *cluster* | T2 *quad.* |
|---|---|---|---|
| Adder | 40 | 6 | 6 |
| Shift8 | 192 | 23 | 22 |
| Ls161 | 222 | 41 | 32 |
| Cont8 | 238 | 42 | 35 |
| Alugate | 424 | 52 | 39 |
| Cla | 528 | 73 | 57 |
| c880 | 1570 | 168 | 109 |
| c1908 | 3146 | 274 | 151 |

- T1: number of tracks - cluster-grow
- T2: number of tracks -quadrature placement

**Table 1 - Routing tracks required using two deterministic placement algorithms**

## 2.2 Routing Length Distribution

The automatic layout synthesis tool generates [1] a histogram displaying the number of nets as a function of the net length. Running the generator over a set of benchmarks (28 to 15000 transistors, 0.25 µm technology), we got an average value of **89%** of connections bellow 200 µm (this routing length, for this process, corresponds to a load of 2.5 gates). Figure 3 shows a histogram for a circuit with 14376 transistors and 4764 nets (ISCAS c7552). In this example, 80.5% of the connections are bellow 100 µm, 9.9% between 100 and 200 µm, 7.4 % between 200 and 500 µm and 2.2% (102 connections) have the total length greater than 500 µm . The information concerning these long nets can be use by an optimization tool, which can resize the cells driving these nets or inserting buffers to reduce delay.

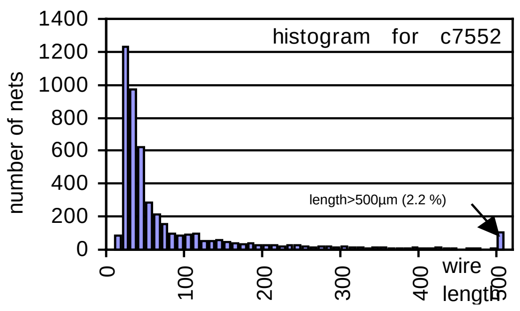

**Figure 3 - Histogram of wire lengths**

This algorithm distributes homogeneously the nets, as expected, with few long nets (less than 3%). These features are important in deep-submicron processes, since long nets represent heavy loads to the drivers (as mentioned a 200 μm net is equivalent to 2.5 gates for a 0.25 μm process).

## 3 STOCHASTIC EVOLUTION

Y. G. Saab and V. B. Rao presented the stochastic evolution in 1991 [7]. Stochastic evolution was formulated as a general technique capable to solve combinatorial optimization problems.

In this particular implementation the algorithm tries to reach the local minimum of the valley where the value of cost is currently located. From there, the control parameter *p* is iteratively increased, allowing each time a bigger "jump" to escape the local minimum. The method somehow inspects many valleys of the cost function, and takes the best ever found. In contrast, deterministic approaches just inspect a single valley of the cost function.

Simulated annealing also "travels" around a good number of valleys. However, the monotone decreasing values given to the control parameter, reduce the effectiveness of the method. Stochastic evolution tries to adapt the main control parameter of the algorithm to the particular environment of the current state.

The stochastic evolution algorithm is outlined in Figure 4.

```
$S$ = Stochastic_Evolution($S_0$,$p_0$)
  $S_{best}$, $S_{cur}$, $S_{saved}$ $\leftarrow$ S0   /* initial state */
  $C_{pre}$, $C_{cur}$, $C_{best}$, $C_{saved}$ $\leftarrow$ cost($S_0$)
  /* control parameters initialization */
  $p \leftarrow p_0$
  $y \leftarrow 0$
  $l \leftarrow Ml$
  $q \leftarrow Q$
  counter $\leftarrow$ 0          /* initialize counter */
  repeat
    $S_{cur}$ $\leftarrow$ perturb($S_{cur}$, $C_{cur}$, $p$, $l$)
    $C_{cur}$ $\leftarrow$ cost($S_{cur}$)
    if $y = \lambda - 1$ then
      if $C_{cur} - C_{best}$ > NEG_INC_ALLOWED $\cdot$ $C_{saved}$ then
        $S_{cur} \leftarrow S_{saved}$
        $C_{cur} \leftarrow C_{saved}$
      else
        $S_{saved} \leftarrow S_{cur}$
        $C_{saved} \leftarrow C_{cur}$
      endif
    endif
    update($p$, $y$, $\lambda$, $l$, $Ml$, $q$, $Q$, $C_{pre}$, $C_{cur}$)
    $C_{pre} \leftarrow C_{cur}$
    if $C_{cur} < C_{best}$ then
      $C_{best} \leftarrow C_{cur}$
      counter $\leftarrow$ counter - R
    else
      counter $\leftarrow$ counter + 1
    endif
  until counter > R
  $S \leftarrow S_{best}$
end
```

**Figure 4 - Stochastic evolution algorithm**

It consists of a set of initialization statements and a main loop where the current state is iteratively modified by function *perturb*, the cost updated and the new state stored if it outperforms the best state found so far.

The set $B = \{b_1, b_2, ..., b_n\}$ is the set of cells to be placed. The placement process searches for a state that minimizes the value of the cost function. Each state $S$ is represented as a two-dimensional matrix with $r$ rows and $k$ columns.

$S_0$, $S_{cur}$ and $S_{best}$ are referred to the initial state, the current state and the best state found, respectively. Similarly, $C_{pre}$, $C_{cur}$ and $C_{best}$ denote the cost of the previous state, the cost of the current state and the cost of the best state found by the algorithm.

The algorithm is controlled by a set of parameters: $p$, $q$, $y$, $R$, $l$ and $\lambda$. $p$, initially set to $p_0$, controls the stochastic acceptance of moves that represent an increase of the cost. Its value is updated in each iteration according to the difference between the cost of the previous and the current state. To help in this update, variable $y$ works as a counter of the number of iterations since $p$ was last reset to its initial value $p_0$. Parameter $q$ also aids to this task. The update of parameters is performed by function *update*. Parameter $l$ establishes the distance between cells to be placed. $Q$ and $Ml$ are the initialization values for $q$ and $l$, respectively.

Constant $R$ is the number of iterations added to the process each time the cost is reduced. Undubitably, it has effect on the duration of the process. Nevertheless, it is not a critical parameter (when given reasonable values) since the number of iterations performed by the algorithm is a combination of this value and the number of times the cost changes. Essentially, the process stops when there is no change in the cost for at least $R$ iterations.

Starting from the initial state $S_0$, the algorithm iteratively generates a new state, evaluates its cost and decides whether the new state is accepted or rejected. The acceptance of new states is controlled at two levels. Function *perturb* performs a set of moves each time it is executed. Moves that decrease the cost are accepted. As a probabilistic algorithm, not all moves with negative gain are rejected. A stochastic decision is made on the acceptance of such moves. If the new state reported by function *perturb* improves the best state found so far, the best state is replaced by the current state. In any case, the new cycle is restarted with the new state.

However, there is no sense in following a sequence of moves that clearly increases the cost. Parameter $\lambda$ is used to control this situation. Each $\lambda$ iterations the value of the cost is checked for excessive negative gains. If the cost is within the allowed margins, the state is stored in $S_{saved}$. Otherwise, the last state stored replaces the current state. The non-deterministic characteristic guarantees that the sequence of moves is different to those in previous attempts. The maximum negative gain allowed when checking the cost each $\lambda$ iterations, is a user defined percentage of the current cost (NEG_INC_ ALLOWED). Control parameters are updated at each iteration in accordance with the evolution of the cost, and the process continues until the stopping criterion is satisfied.

Accounting the number of times the cost is decreased solves the stopping criterion, one of the main drawbacks of probabilistic approaches. For each new best state found, the process is added a user defined number of iterations, *R*. For the process be stopped, at least *R* iterations must be executed without improving the cost.

### Function *perturb*

The function *perturb* is responsible for the moves performed in each iteration. Beginning with the first row, matrix $S_{cur}$ is iteratively scanned from location $S_{cur}(0,0)$ to $S_{cur}(r-1,k-1)$. For each one, function *move* selects a new location in $S_{cur}$ and exchanges their contents. Therefore, function *move* outputs a new state $S_{new}$, which is evaluated and compared with $S_{cur}$. The cost of the new state is computed incrementally from the current state $S_{cur}$.

Whenever the cost of the new state outperforms the cost of the current state, the move is accepted and the current state updated with the new one. The acceptance of a new state with a higher cost is subjected to a stochastic decision. Parameter *p* controls this decision. If the negative gain does exceed *-p* the new state is directly rejected. Otherwise, the probability of acceptance depends linearly of *(gain - p)*. It must be noted that $p = 0$ transforms this probabilistic algorithm in a monotone decreasing method, where no move with negative gain is accepted. Procedure *update* uses this property to force the algorithm to find a local minimum.

### Procedure *update*

The main task of procedure *update* is the update of the control parameter *p*, which is responsible for the acceptance or rejection of moves that increase the cost. The use of the counter *y* has already been discussed when describing the basic structure of the algorithm. Its contents is checked in each iteration. Whenever the count reaches $\lambda$, parameter *p* is reinitialized to $p_0$, distance *l* updated and the counter *y* reset. Parameter *l* is updated by taking the greater integer below *l/2*, if $l > 1$. Otherwise it is set to its maximum value *Ml*, which is user-defined.

A particular distance is maintained during $\lambda$ consecutive iterations. This means $\lambda$ attempts of exchange each cell before changing the distance. In these iterations, *p* is updated according to the changes in the cost. When a change in the current cost is recorded, *p* is set to *0*. This is made to force the algorithm to search for the local minimum. “Jumping” to a “deeper valley” of the cost function will make any improvement of the cost from this point.

The algorithm always tends to a local minimum. From there, movements that increase the cost will be necessary to escape the local minimal. When no change in cost happens in two consecutive iterations, *p* is set to $p = -2 \cdot q \cdot gain / (q - 1)$ where *gain* stands for the maximum negative gain introduced by any proposed move in each iteration. Function *perturb* must keep track of this value. It is very likely that such move has not been accepted. However, the maximum negative gain can be viewed as a boundary. The update of *p* takes this boundary into account. *q* helps in the update of *p*. Initially, it is set to *Q*, a value defined by the user. Only consecutive iterations with no change in the cost change *q*. In this case, *q* is iteratively set to $\lfloor q/2 \rfloor$, provided that $q \geq 4$. This is made to increase *p* and force the algorithm to accept moves with higher negative gains.

## 4 COMPARISON

Both methods, described above, quadrature placement and stochastic evolution placement, have been implemented and integrated within TROPIC [1]. Table 2 shows the results obtained using these approaches:

| Circuit | Cells | density QP | CPU QP | density SE | CPU SE |
|---|---|---|---|---|---|
| **Adder** | 6 | 106326 | 10 ms | 105218 | 45 s |
| **Cont8** | 62 | 61893 | 20 ms | 63742 | 6 min |
| **Alugate** | 94 | 76349 | 60 ms | 73125 | 9 min |
| **cla** | 102 | 75659 | 60 ms | 72490 | 10 min |
| **c432** | 197 | 97605 | 73 ms | 91858 | 15 min |
| **c880** | 566 | 67640 | 90 ms | 68007 | 50 min |
| **c1908** | 1073 | 66862 | 130 ms | 69531 | 175 min |

QP = quadrature placement / SE = stochastic evolution placement

**Table 2 - Comparisons between deterministic and non-deterministic placement algorithms**

In this table, circuits are ordered by the number or cells to be placed by both algorithms (column 2). Column density reflects the silicon area. The unit is transistors/mm$^2$ (0.25 μm). Column CPU shows the CPU time consumed by the algorithms, in a Sun Ultra-Sparc 10.

The tuning of the Stochastic Evolution algorithm is not critical when given reasonable values to the control parameters. These values are listed below:

| | |
|---|---|
| $R$ | 10 – 50 |
| $\lambda$ | 4 – 8 |
| $p0$ | 0 |
| $Q$ | 4 – 16 |
| $Ml$ | ($D/8 \le Ml \le D/4$), **Where**: $D = (\|B\|/r) + r$, $r$ is the number of rows and $\|B\|$ the number of elements to be placed. |

For this comparison, parameters of the Stochastic Evolution algorithm have been given values within these ranges.

This table shows a big difference in the CPU time required by both methodologies. This difference comes from the deterministic and non-deterministic characteristic of Quadrature and Stochastic Evolution, respectively. Stochastic Evolution uses a random initial place that is changed, step by step, exchanging elements. After each exchange, the whole circuit must be evaluated to accept or reject the movement. This is the source of the long CPU times, which is a characteristic of non-deterministic algorithms. This global evaluation is not needed in the deterministic Quadrature approach.

The transistor densities obtained are similar for both methodologies. However, smaller circuits are better for the deterministic approach, while the non-deterministic approach obtains some advantage with bigger circuits (the transistor density for the c1908 benchmark is 4% better when using the SE placement).

On the other hand, in several tests, it has been observed that the Quadrature placement tends to reduce the maximum length of a wire in the circuit, while the Stochastic Evolution approach reduces the overall wire length.

## 5 CONCLUSION

Both methodologies obtain similar transistor densities, and therefore, in this particular application quadrature placement has the advantage of its reduced CPU time.

As stated before, the difference in the CPU time is directly connected to the different philosophy of the algorithms: deterministic and non-deterministic. In turn, Stochastic Evolution has the advantage of a bigger flexibility, since user can modify the evolution of the process through the different parameters.

To effectively reduce the CPU requirements of the Stochastic Evolution, the incremental evaluation of the cost must be improved. However, these CPU time requirements will always be considerably larger than the CPU requirements of a deterministic placement.

It must be noted that because of the different internal processes used, each methodology is focussed towards a slightly different optimization objective. It has been observed that quadrature placement tends to a more homogeneous distribution of the wire connection density, while Stochastic Evolution usually builds some high congestion zones. These zones involve a significant increase in area consumption. This could be one of the reasons of the similar transistor densities obtained. It should be tested, but it is very likely that a higher number of interconnection layers could make bigger differences in transistor densities.

In any case, the advantage than can obtain Stochastic Evolution in bigger circuits does not justify its use in any preliminary test. In this cases quadrature placement should be used. Stochastic Evolution could be interesting only when everything has already been tuned in several tests, and when CPU time is not so critical.

## 6 REFERENCES

[1] Moraes, F.; Robert, M.; Auvergne, D. *A Virtual CMOS Library Approach for Fast Layout Synthesis*. In: VLSI, 1999, Lisbon - Portugal.

[2] Semiconductor Industry Association. The National Technology Roadmap for Semiconductor. Available by WWW in http://notes.sematech.org/ntrs/PublNTRS.nsf, 1997

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

[4] Fiduccia, C.M., Mattheyses, R.M. *A linear time heuristic for improving network* partitions. In: ACM/IEEE Design Automation Conference, Las Vegas, 1982. pp. 175-181.

[5] Dunlop, A. E. & Kernighan, B. *A procedure for placement of standard-cell VLSI circuits*. IEEE Transactions on CAD, v.CAD-4, n.1, Jan. 1985, pp. 92-98.

[6] Moraes, F., Azemard, Robert, M.; Auvergne, D. *Flexible macrocell layout generator*. 4th ACM/SIGDA Physical Design Workshop, Los Angeles, 1993, pp. 105-116.

[7] Youssef G. Saab and Vasant B. Rao. *Combinatorial optimization by stochastic evolution*. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 10(4):525-535, April 1991.

**Acknowledgements**

Fernando Moraes gratefully acknowledges the support of the CNPq through grant no. 522939/96-1.