Post-placement residual-overlap removal with minimal movement

Sudip Nag
Kamal Chaudhary
Xilinx Inc, 2100 Logic Drive, San Jose CA 95124.

Abstract

In this paper we present a novel approach for removing residual overlaps among blocks. We start out by representing the placement in the sequence pair form and describe transformations to the sequence pair to make the placement feasible. This is followed by a distance-based slack allocation to generate a new placement with no overlaps, while being as close to the original placement as possible. Our results demonstrate the efficacy of our approach in transforming layouts with overlaps to overlap-free layouts with minimal object movement.

1 Introduction

Aggressive placement techniques[7][8] often resort to infeasible-intermediate states in the course of iterative placement improvement. Allowing such infeasibility (overlap being a widely-used example for placement) gives lot more flexibility to the optimization process, thereby hastening the progress towards an optimal solution. In order to ensure that the overlaps are completely removed by the end of the placement phase, placement algorithms dynamically increase the overlap-related weights as the optimization proceeds. While this method successfully removes all overlaps in almost all cases, it cannot guarantee that overlaps will be removed in *all* cases. This is especially true for layouts with fixed-area, like gate-arrays and FPGAs. Also, since very few overlaps will remain at the end, one would like to get a feasible placement (satisfies fixed area constraint and with no overlaps) which is as close to the original infeasible solution as possible. There is a need, therefore, of a mechanism which identifies *what minimal placement changes (movement) will result in a feasible placement*. Our work addresses this problem.

Several block placement/floorplanning techniques [1][2][3] exist for directly generating overlap free placements with different block sizes. However, run time considerations limit these approaches to the instances with at most couple of hundred blocks. The placement problem for large FPGAs[10] consists of few thousand blocks with large number of small singular size Configurable Logic Blocks (CLBs), several hundred medium size objects e.g. counters, muxes, adders etc., and few large macros/IP cores. In such an instance use of annealing with overlaps is an effective algorithm, provided it is followed by a overlap removal process for residual overlaps.

Overlap removal is in general a non-trivial problem. To see an illustration of this, let us analyze the example shown in Figure 1. In this example module J is overlapping with modules G and K. These overlaps cannot be removed by just moving the overlapping modules G, J or K since such moves will result in other overlaps. One possible solution is moving J to the right followed by K upwards and finally moving M to the left. This solution is not trivial even for a human designer. The main difficulty arises because of interaction between the horizontal and vertical directions; specifically the decision in one direction can have deleterious effects in the other.

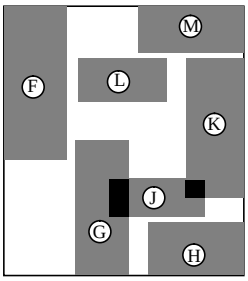
[4] describes a previous work for removing overlaps. Their graph based overlap removal method iteratively uses the following two steps: *a*) remove redundant critical arcs *b*) change the aspect ratio of the modules. Step *b* is clearly not applicable for placement of modules with fixed shapes. Their graph based approach starts with a highly redundant graph (in terms of object relationships) and step *a* attempts to obtain a feasible solution by removing redundant arcs. Repeated application of step *a* can at best result in graphs with zero redundancy.

Dealing with both horizontal and vertical dimensions simultaneously is a complex task. Instead, it is often better to break it to two independent one dimension problems. For that purpose Sequence pair representation of placements introduced by Murata et. al[1] provides an elegant representation. In contrast to [4] our approach *starts* with a zero-redundancy graph model (i.e. between objects we either have horizontal or vertical relationship). Thus, application of step *a* will not result in any change to the graphs. In order to remove remaining overlaps, our graph-based manipulation needs to be much more comprehensive.

Rest of the paper is organized as follows. In section 2 we describe the constraint-graph creation process, which

forms the backbone of our approach. In section 3 we describe the details of our approach for graph manipulations in order to remove overlaps with minimal movement. In section 4 we present the results of our approach. Finally we conclude in section 5.

Figure 1. Why overlap removal is hard

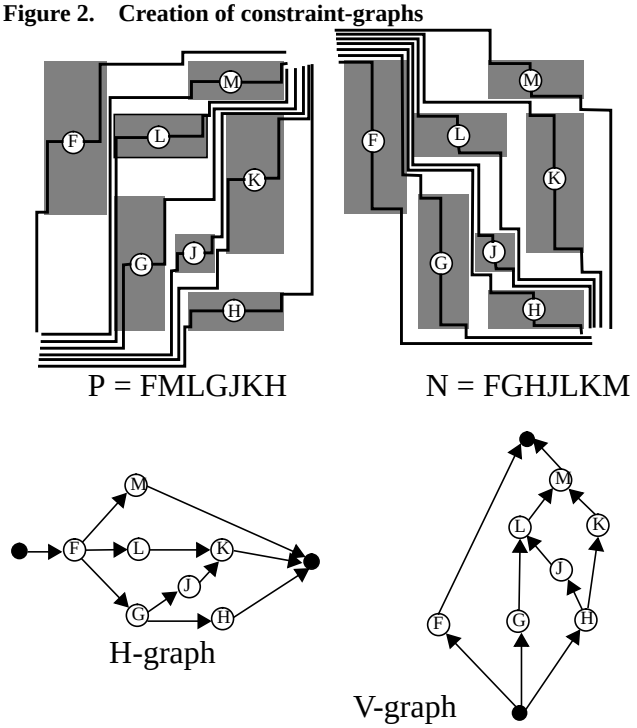


2 Background

Our starting point is representing the placement (with overlaps) in the form of vertical and horizontal constraint graphs, which capture the relationship between objects in the two axes. The rationale for using these graphs is that they allow us to deal with the *2-D* placement problem in terms of two *1-D* problems (of course this requires certain rules/restrictions during the graph formation). For this part we use the method proposed in [1] and extend it to handle cases with overlaps. Graph-formation is used in [1] for post-layout compaction and during annealing moves. The power of this method is the simplicity of the relationship it establishes between placement, object-order, and H/V-constraint-graphs. Either of the three can be used to determine the other two. Based on the placement, two sorted sets are created, a pos-set(*P*) formed by scanning from top-left to bottom-right and a neg-set(*N*) formed by scanning from bot-left to top-right. The process is shown in more detail in Fig. 2. For creating *P*, the following is done. Put a pebble *b* at the center of an object. Move it right to hit the cutting-seg which is an arc of the object. Then, move b upward until it hits an orthogonal cutting-seg. Then, move it right until it hits an orthogonal cutting-seg, and continue turning directions as right, up, right,..., until *b* reaches the right corner of the chip. This forms the right-up locus of *b*. *P* is formed using the loci right-up and left-down ordering. *N* is formed using the left-up and right-down loci ordering. Once sets P and N are formed, we can use them to form the H/V graphs. If two elements appear in order a..b in P and in order a..b in N, then there is a horizontal-arc (in H graph) from a to b. If two elements appear in order a..b in *P* and in order b..a in *N*, then there is a vertical-arc (in V graph) from b to a. Transitive arcs can be omitted. Using these rules, we show the *P/N* sets formation and H/V graph formation for an example in Fig. 1.

Both graphs do not contain any directed cycles. For every pair of modules, there is always an edge (direct or transitive) in H-graph or V-graph, and not in both. This allows independent determination of X and Y coordinates. We will refer to these properties of the graph pair as *L1*.

Figure 2. Creation of constraint-graphs



3 Our Overlap Removal Approach

In our approach, we give more priority to the large (size > 1) objects compared to small (size = 1) objects. This is because it is harder to perturb the locations of large objects. The small objects can be easily fitted at the end. An overview of our approach is given below:

0. Isolate the large objects(size > 1).
1. Create constraint-graphs for them: handle overlaps.
2. Make the graphs feasible by changing object ordering.
3. Determine the final locations based on slack-allocation.
4. Place the small objects(size=1) in remaining slots using bipartite-matching.

We will now describe these in detail.

3.1 Create constraint-graphs

In order to handle overlaps, we derive object relationships (H/V graph) by assuming that the sizes of the overlapped objects are reduced in the optimal direction. Optimal direction is the direction (top/left/bottom/right) which achieves no overlap with minimum reduction. For example in Figure 2, the size of J was reduced in horizontal direction for graph creation. We must clarify that the actual object sizes are not reduced: the reduction concept is used only to determine realistic object relationships.

For each module in a design, a node is created in both the H and V graphs. A value equal to the module's width is assigned to the H node. A value equal to the module's height is assigned to the V node. Each arc in H/V graph is assigned a value representing the horizontal/vertical distance between the modules respectively. A -ve distance implies overlap. For overlap free placement all the arcs need to have non-negative values.

In some cases the overlap removal can be simply achieved by moving modules away from the overlap area. This corresponds to setting -ve arc values to 0 and reducing some +ve arc values. We will refer this case as *Feasible* case. In many cases such simple transformation is not sufficient to remove all the overlaps, Figure 1. being an example. We will refer this case as *Infeasible* case. To remove overlaps in Infeasible cases more elaborate graph transformations are needed. We describe them next.

3.2 Graph Transformations for Feasibility

Initially we set all the arc values to 0, and find the longest path (in terms of arc/node values) in H and V constraint graphs. We set the target for H constraint graph to be the chip width and that for the V graph to be the chip height. If the path length exceeds the target, then the constraint graphs are infeasible. We cannot proceed to the next step unless we make the graphs feasible.

Let P_i be the length of the longest path passing through arc_i. A slack s_i for each arc is defined as the difference between the target and P_i. All arcs with negative s_i are considered critical. Slack computation for all the arcs can be performed in two linear passes[5][6].

The aim of our transformation is to remove all the critical arcs while maintaining the property *L1* of the graphs. Note that in order to remove a critical arc A->B in H graph, it is not sufficient to add an arc between A and B in V graph. *L1* property may require more changes to the graph. An easy way to perform *L1* consistent transformation is to permute elements in the sequence pair for arc removal and then recreate the graphs. Several permutations are possible for removing an arc; each with different effects on the resulting floorplan. Ideally, we would like to keep the changes to sequence pair small so as to keep the final overlap- free floorplan closer to the original floorplan. Keeping these requirements in view we discuss two types of permutations, namely Swap and Move.

3.2.1 Swap Permutation:

There can be two cases for critical arc removal; removing a horizontal-arc or removing a vertical-arc. First let us assume we want to remove a horizontal-arc from A->B. This implies the starting configuration shown in Fig. 3. In this figure, *Pl* refers to the set of objects to the left of A in set P, *Pr* (to right of B) and *Pw* is the set of objects within A and B in set P. A similar definition exists for *Nl*, *Nr* and *Nw*. Set *PwNr* represents the intersection of sets *Pw* and *Nr*. Arc A->B can be removed by swapping elements A and B in either P or N set. Swapping in the P set will result in a new vertical arc from A to B. Swapping in the N set will result in a new vertical arc from B to A.

Without going into the details, swapping A and B in *P* set results in the following arc additions/removals:

H-graph additions: B->NrPw NlPw->A

H-graph deletions: A->B NwPw->B A->NwPw A->NrPw NlPw->B

V-graph additions: A->B NwPw->B A->NwPw A->NrPw NlPw->B

V-graph deletions: B->NrPw NlPw->A

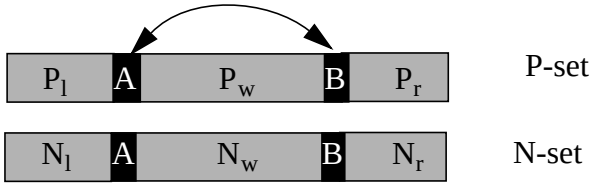


Figure 3. Swapping A/B in P-Set

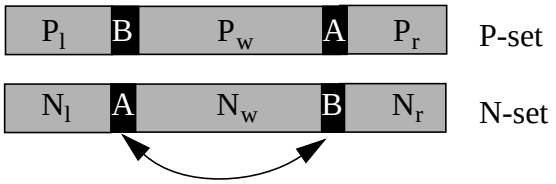


Figure 4. Swapping A/B in N-set

Swapping A and B in the N set as shown in Fig. 4 results in the following arc additions/removals:

H-graph additions: B->A B->PwNw PwNw->A PlNw->A B->PrNw

H-graph deletions: A->PrNw PlNw->B

V-graph additions: B->PlNw PrNw->A

V-graph deletions: A->B PwNw->B A->PwNw A->PlNw PrNw->B

3.2.2 Move Permutation:

Move based layout permutation starts with selecting an object *O* for moving. The selection is based on two criteria: the number of critical paths (paths with -ve slack) passing through *O* and the size of *O*. The intent is to remove *O* from its current location so as to lessen (or remove) the criticality of paths passing through *O*. The next step is to identify an arc where *O* can be introduced with no deleterious effects. We create a list of plausible arcs i.e. arcs with slack greater than size of *O*. We sort this list based on distance from *O*. Finally we test the arcs till a feasible solution is found. Assume we are trying to insert *O* in arc (A->B) in H graph. From the P/N set viewpoint, this is equivalent to inserting *O* between the locations of A and B in Pset and Nset. Each choice gives rise to new arcs from/to *O*. We test the effect of these new arcs on the slack of paths passing through *O*. If we find a location of *O* where any new path slacks is less than the current worst slack, we stop and accept that solution. Although these steps might look too complicated and compute-intensive, in practice, owing to our judicious sorting heuristics, we find a plausible move very quickly.

The above information is used to calculate the effect of a Swap and Move efficiently and exactly. For each affected node, the cost of the added/deleted arcs is computed to determine the best arc to remove. Additional criteria such as number of critical paths passing through the arcs are used to break ties. This process of identifying the critical arc and performing the swap in P or N set is repeated till the graphs become feasible. Once the graphs are feasible, we can proceed to the next step: slack allocation, which is done independently in the two graphs.

3.3 Slack-allocation

Unlike for ASICS, where the goal is to minimize area, for FPGAs the area is fixed for a chosen device. Also, for ASICs it is assumed that the most compact placement is the best, from area as well as wiring viewpoint. In FPGAs, because the I*O* locations are on the edges of a *fixed* boundary, these assumptions do not hold true. In case of FPGAs a compacted layout often results in *degradation* of performance and routability. Further, we do not consider singular objects (CLBs) during this phase. For all these reasons, in our case it is important for the relative spacing between objects to be maintained as much as possible, compared to what they were in the initial infeasible placement. Our slack allocation process achieves this goal.

There is a strong parallel between our slack allocation and what is normally done for timing-related optimization [5][6]. The difference is that in our case distance replaces delay and chip width/height replace required clock frequency/path delay.

We start with all arc-values zeroed out. Slack analysis in this situation determines how much each path can be extended, i.e. by how much, objects on this path can be spread apart while still ensuring that all objects lie within the chip, without overlapping. Once we obtain path slacks, it is distributed amongst the arcs that the path comprises based on the arc-weights. In reality we perform a fast arc-based slack analysis, which determines the arc-slacks directly, in linear time, without explicitly doing any path-analysis.

The weights we choose for an arc is its original value (before it is zeroed out). This ensures that the final arc-values and therefore the final object locations are as close to the original placement as possible. In fact if the original placement had no overlaps, this slack analysis will result in the exact-same placement.

During slack-allocation, a common problem is determining where to allocate residual slacks. Let us assume that 3 arcs a, b and c comprise path p. Also, let us assume Wa=1, Wb=1, Wc=2 and slack Sp of path p is 14. In the first iteration, slacks 3, 3, 6 get assigned to a, b and c respectively. This leaves a slack of 2 to be distributed among a, b, and c. However we can assign only integer values. So, what should be the distribution?

To solve this problem, let us look closely at what happens if we bump-up the distance of an arc *A* by 1 in say the V-graph. Because of this arc increasing, all the critical nodes that *A* feeds to will be pushed up by 1. Critical nodes are those whose most critical fanin path originates from arc *A*. The goodness of the decision of bumping up arc *A* by 1, is dependent on the goodness of the *effect* of that on the fanout nodes/objects. For a particular upstream affected node *N*, let the current distance (from origin in y dir) be *d*. Let its original distance (based on starting placement) be *od*. Therefore the distance of *N* from its original/preferred location changes from (*d-od*) to (*d+1-od*). The change is good if bumping up *d* by 1 brings it closer to *od*. The goodness of the change therefore depends on how -ve $(d+1-od)^2-(d-od)^2$ is. Therefore, the goodness of bumping-up of arc A by 1 depends on negativity of :

$$\sum_{upCriticalNodes} (2\times(d_N - od_N)+1) \quad ..$$

We use this equation to break ties during slack allocation. This can be computed in linear time by examining nodes in topologically sorted order from output to inputs. After slack allocation, the final overlap free locations of all objects is known.

3.4 Bipartite-matching

As mentioned earlier, we use the constraint-graph based overlap removal only for large (size > 1) objects. All singu-

lar objects are then placed in the remaining slots with minimal movement. This is done using bipartite-matching technique[9]. This method is ideally suited for our problem of mapping N objects to S (>= N) slots with a fixed cost-matrix which represents the cost of assigning an object n in slot s. For us the cost is the distance of s from the original position of n (os). Initially we try to find a solution by assuming that object n can only go to slots s which are within a radius R from its original location (os). We slowly bump-up the value of R till a feasible solution is reached. The reason for this approach is to control how far the CLBs can move and also to limit memory/runtime usage for very large chips.

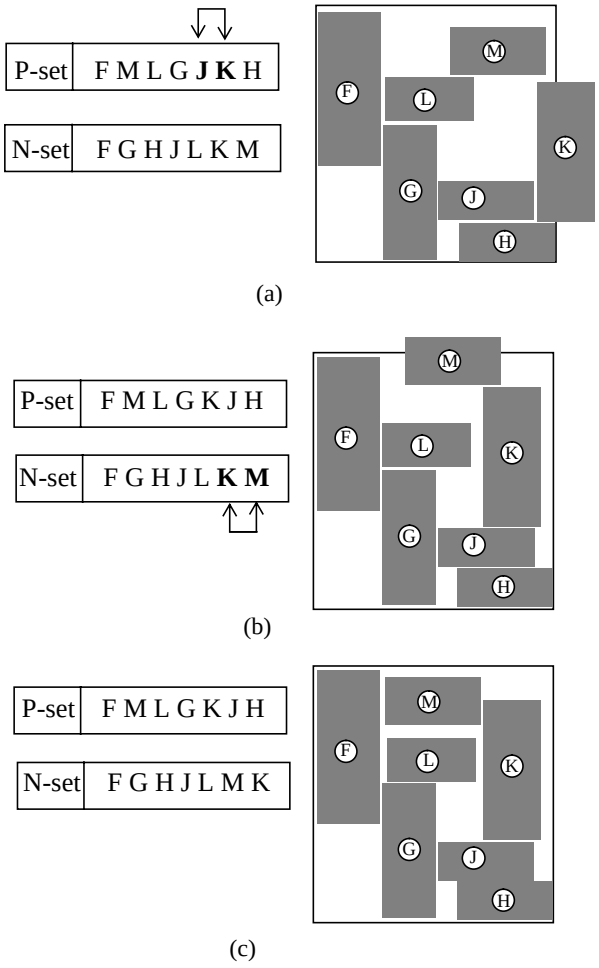


Figure 5. Overlap removal progression

3.5 Example

Fig. 5 shows steps of our overlap removal process for the example of Figure 1. Step (a) depicts the initial situation. The corresponding H/V graphs are shown in Figure 2. Slack analysis on the H graph generates F->G->J->K as the most critical path. Each arc on this path is cost analyzed and arc J->K is chosen as the best swap candidate in P-set. The result of this swap is shown in (b). This transformation changes the relationship between J and K from "right-of" to "top-of". This change makes the H graph feasible but causes the V graph to become infeasible. Slack analysis on the modified V graph identifies H->J->K->M as the most critical path. Cost analysis on this path results in K->M as the best swap candidate in N-set. After this transformation both the H and V graphs are feasible. Slack allocation is then performed to determine the final locations as shown in (c).

Figure 6. Layout with overlaps

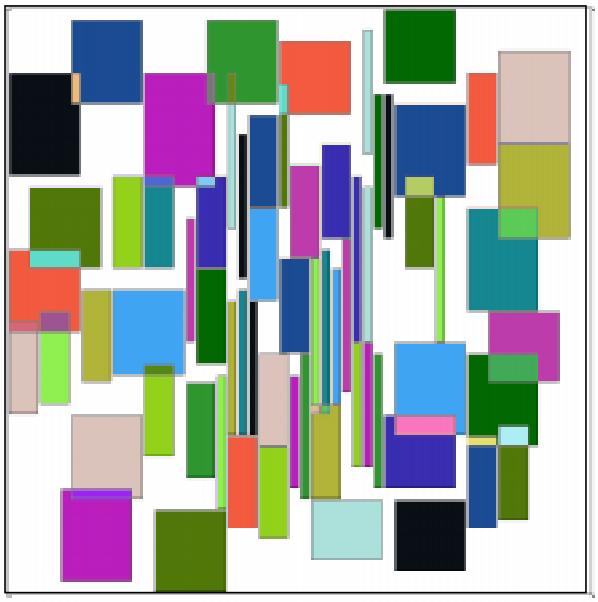
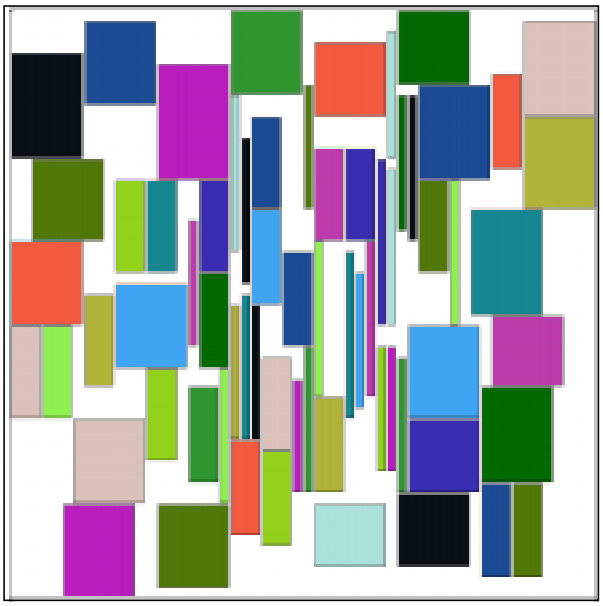


Figure 7. Layout without overlaps and minimal movement



4 Results

We have tried our approach on examples of various sizes and complexity. It was able to remove overlaps in all the cases. For a very hard example (100% full Xilinx XC40125XL), we provided our overlap-removal method with a placement with 9 overlaps (a typical scenario at the end of a placement process). Our tool successfully removes all overlaps while increasing the score by only 5% which is very good considering that we are dealing with a 100% full design. This overlap removal process took less than 2 minutes on an UltraSparc2.

In Fig. 7 we show the result on a FIR filter example occupying 70% of a XC4085XL. The figure on the top shows the layout with significant overlaps. For the sake of clarity, we do not show the singular CLB-size elements, which are considered only during the bipartite matching phase. The figure in the bottom shows the result of our tool: it successfully removes all overlaps with minimal movement.

5 Conclusion

We have presented an algorithm for efficiently removing residual overlaps with minimal placement change. Our method is based on a previously published elegant placement representation. We extend this representation scheme to handle placement with overlaps. We define a novel technique for graph manipulation and distance slack allocation in order to remove overlaps with minimal placement change. Our results show the efficacy of our approach in removing residual overlaps with minimal movement.

References

[1] Hiroshi Murata, Kunihiro Fujiyoshi, Shigetoshi Nakatake and Yoji Kajitani, “Rectangle-Packing-Based Module Place ment,” Proc. of ICCAD, 1995.

[2] W.M. Dai and E.S. Kuh, “Simultaneous Floorplanning and Global Routing for Hierarchical Building Block Layout,” IEEE Transactions on CAD, vol. CAD-6, no 5, pp. 828-837, Sept. 1987.

[3] D.F. Wong and C.L. Liu, “A New Algorithm for Floorplanning Design,” Proc. 23rd Design Automation Conference, pp. 101-107, 1986.

[4] G. Vijayan and R.S. Tsay., “Floorplanning by Topological Constraint Reduction,” *Proc. of ICCAD* 1990

[5] Jon Frankle, “Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing,” *Proc. DAC*, 1992

[6] H. Youssef and E. Shragowitz, “Timing constraints for correct performance,” *Proc. of ICCAD* 1990

[7] C. Sechen and K.W. Lee, “An improved simulated-annealing algorithm for row-based placement,” *Proc. of ICCAD* 1987

[8] J.M. Cohn, D.J. Garrod, R.A. Rutenbar and L.R. Carley, “KOAN/ANAGRAMII: New Tools for Device-Level Analog Placement and Routing”, *IEEE Journal of Solid State Circuits*, 1991

[9] A. Srinivasan, K. Chaudhary, and E.S. Kuh, “RITUAL: A performance driven placement algorithm,” IEEE Trans. CAS, Nov. 1992.

[10]The Programmable Logic Data Book, Xilinx Inc., San Jose California, 1996.