A New Buffer Planning Algorithm Based on Room Resizing*

Hongjie Bai, Sheqin Dong, Xianlong Hong, and Song Chen

Department of Computer Science and Technology, Tsinghua University, Beijing, China,
bhj04@mails.tsinghua.edu.cn,dongsq@mail.tsinghua.edu.cn

Abstract. This paper studies the buffer planning problem for interconnect centric floorplanning. Dead-spaces not held by blocks are the available location for buffer insertion. To make best use of these spaces for buffer requirements, we have to move blocks so that blocks' room size is adjusted and dead-spaces could be redistributed. In this paper, we introduce a new algorithm to move blocks not only within its room, but also in the space currently held by other blocks by pushing away these blocks if necessary without violating the topological and the total area. After applying this method of redistributing dead-spaces, the number of nets satisfying delay constraint can be optimized.

1 Introduction

As the VLSI circuits are scaled into nanometer dimensions, interconnect plays a dominant role in the performance and cost. To ensure the timing closure of design, interconnects must be considered as early as possible. Buffer insertion is among most effective ways to reduce interconnect delay which breaks long interconnects into shorter ones so that overall delay can be reduced. However, most buffer insertion algorithms were designed for post-layout interconnect optimization. Obviously it is not suitable if we have no global planning for tens of thousands of nets that need buffers to be inserted. Also, buffers take up silicon resources, so it is necessary to plan buffer as early as possible in the design of VLSI.

1.1 Previous Work

[3] introduced the concept of feasible region which employs Elmore delay model to generate buffer blocks. [5] extends it by adding independence to feasible region and also tries to relieve routing congestion. [6] proposed an optimal algorithm which assumes only one buffer per net. [7] use multi-commodity flow-based approach which allocates buffers to pre-existing buffer blocks. [8] proposes

* This paper is supported by National Nature Science Foundation of China (NSFC): 60473126, NSFC 60121120706 and National Natural Science Foundation of USA (NSF) CCR-0096383, Hi-Tech Research & Development (863) Program of China 2004AA1Z1050

a routability driven floorplanning. Most of these methods try to allocate buffers into dead spaces, however none of these methods take concern of using dead spaces effectively enough so that many dead spaces are left unused while still many buffers have no space to insert. [2] introduces a dead space redistribution-based method to make better use of dead space for buffer insertion. However, there exists a drawback that it can only move blocks within its room. Figure 1(a) is an example. With the algorithm introduced in [2], block C can't be moved horizontally so that some buffer requirements can't be satisfied. However, if E could move right a little to lend some space to C, all buffers can be successfully inserted as shown in Fig. 1(b) .

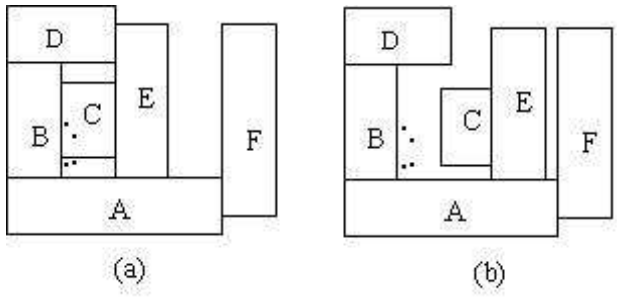

Fig. 1. Buffers can't be placed in (a) if C only moves within its room, but can be successfully placed in (b) if we enlarge C's room

1.2 Our Contribution

Except for moving blocks within their rooms, the rooms' size could be adjusted without changing the blocks' relative positions and buffer planning could be improved greatly. In that case, if we wish to find the optimal solution to insert buffers, we have to try moving blocks which may even push other blocks. So our algorithm is concentrated on moving blocks as much as possible in order to provide dead-spaces for buffer, at the same time, the dead-spaces of the total placement are managed systematically. This new method will be very effective for IP based SOC planning. For instance, for design reuse where topological relations among blocks could not be changed, this method can make blocks move in a range as much as possible to optimize buffer planning.

The rest of the paper is organized as follows, section 2 gives a problem definition and a brief overview to the background including Feasible Region and Corner Block List. Section 3 discusses our algorithm. Section 4 and section 5 give the experimental result and conclusion.

2 Preliminary

2.1 Problem Formulation

Given an initial placement and the nets with timing constraint, for the nets that does not satisfy timing closure, we determine the buffer number needed for each net and the definite buffer locations. Since buffer requires silicon resources, it can only be located in dead-spaces which are not held by blocks. To best satisfy buffer requirements, blocks have to be moved to make more nets meet timing closure while the total area and topology remain unchanged.

2.2 Corner Block List(CBL)

Corner block list (CBL) is a topological representation introduced in [1]. CBL uses three lists S,L,T to represent a floorplan. Each member in S represents a block, L and T are lists of 0 and 1. While applying a CBL to a floorplan, we assign the blocks in S one by one to the up-right corner. The corresponding number in L denotes the direction. 0 makes the new block come from the top direction and 1 makes the new block come from the right direction. The segments lying on the left and bottom boundary of the corner block compose a T-junction which defines the orientation of the corner block. The number of consecutive 1 in T means the number of T-Junctions the new block will cover. The insertion process of a block is shown in Fig. 2. Details can be found in [1]. .

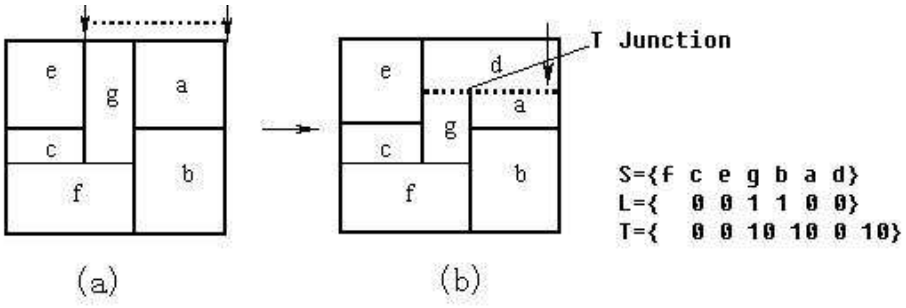

Fig. 2. From CBL to packing, insert corner block d to the corner and it covers one T junction so that it covers g and a

3 Room Resizing

Our algorithm is composed of four steps: (1)Determine maximum move space of each block, (2)try moving some block within its maximum move space, (3)applying traditional buffer insertion algorithm and (4) repeating 1,2,3 to search for an optimized solution. We will explain in details in the following subsections.

3.1 Determine dead spaces and associate them with blocks

Dead space is defined to be space not held by blocks. During process of CBL packing, after placing the corner block from the top direction(L=0), all its covered blocks' top dead-spaces can be determined. Similarly, placing blocks from the right direction(L=1) will determine its covered blocks' right dead space. The blocks on right boundary and top boundary are not covered by blocks and their dead spaces are settled with help of top boundary and right boundary. An example is shown in Fig. 3. In Fig. 3(a), corner block covers A so that X is A's top dead space. In Fig. 3(b), corner block covers D, C and A so that Y is A's right dead space and Z is C's Right dead space.

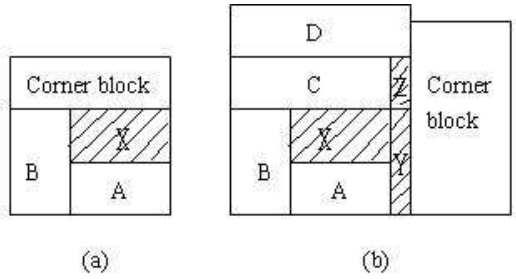

Fig. 3. While placing a corner block, its covered blocks' dead spaces could be determined

After the process of CBL packing, we not only determine all the dead-spaces, but also cut dead-spaces into rectangles naturally. At the same time, each dead-space is associated with a block and none of them intersect with each other. A block together with its dead spaces is defined to be this block's room. Since CBL is left-bottom packed, dead-spaces of a block are either on the top or on the right or both initially. Only moving blocks or redistributing dead-spaces can make a dead-space be on the left or bottom of its associated block. A more detailed description can be found in [4].

3.2 Vertical and Horizontal Constraint Graph

Building constraint graph during CBL Packing is a necessary step for our algorithm because constraint graph is a most efficient way to determine blocks' relative position. Firstly, we build a start vertex S and an ending vertex T. During CBL packing process, if the corner block covers from the top direction, we draw an edge from covered blocks to the corner block in vertical constraint graph, at the same time, we draw edges to the corner block from the blocks which lie left to the left-most covered block. If there are no blocks that lie left to left-most covered block which means the left-most covered block is on the left

boundary, we draw an edge from S to the corner block. New blocks coming from right direction could be done similarly. In the end, we draw an edge from each block on top/right border to T. Considering edge-cost, we define it to be the dead space height/width between two neighboring blocks. An example is shown in Fig. 4, A comes from top and covers B and C, so we draw C->A and B->A in vertical graph in Fig. 4(a). C is the left-most covered-by-A block. E and F are Left to C, so we draw E->A and F->A in horizontal graph in Fig. 4(b). The only non-zero-cost edges are C->B, B->T.

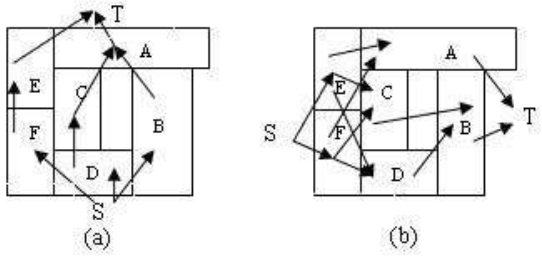

Fig. 4. Vertical(a) and horizontal(b) graph for a packing generated through a CBL

3.3 Maximum-Allowable Move Space (MAMS)

Since each block can be moved in four direction, up, down, left, right, so we keep a note of how much distance a block can be moved in one direction while the total area and topology remain unchanged.

Definition 1. *A block's maximum allowable move space on the right direction (MAMSR) is the maximum distance a block can be moved right, during which the other blocks that block the way may have to be pushed, but the total area and topology remain unchanged.*

The MAMSU, MAMSD, MAMSL, MAMSR which corresponds to the maximum-allowable move space on up/down/left/right direction can be defined similarly. An example is shown in Fig. 5. MAMSR of B is 4 because B can be moved right for distance 4 which pushes C, D and E, the total area and topology is unchanged. Placement after this move is shown in Fig. 5(b). After we move B, MAMSR of B is decreased to 0. Take D for another example, in Fig. 5(a), MAMSR of D is 7, after pushed by B, in Fig. 5(b), MAMSR of D is decreased to 3, but MAMSL of D is increased to 4 from 0.

Initially after CBL packing, each block's MAMSL and MAMSD is zero since we are packing them to left-bottom and none of them can be moved left or

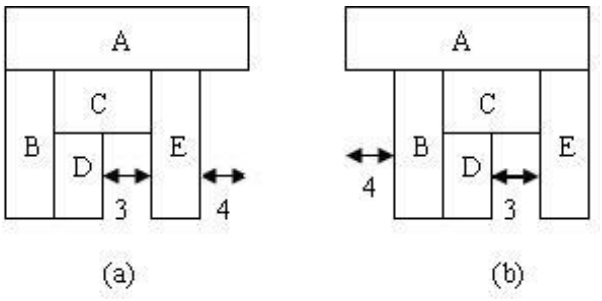

Fig. 5. An example for explaining MAMSR and MAMSL

down. MAMSR and MAMSU have to be calculated through constraint graph, a formula is shown below to determine MAMSR of block A.

$$MAMSR[A] = Min\{MAMSR[B] + edge\_cost(A, B)\} \quad \text{with} \quad (A, B) \in E_h \tag{1}$$

$(A, B) \in E_h$ means that edge A->B is on the horizontal graph. This is a recursive algorithm to determine each block's MAMSR starting from blocks on the right border to left. Take Fig. 5(a) for example:

MAMSR[E]= 4, MAMSR[A]=0,
MAMSR[C]=MAMSR[E]+ edge_cost(C,E)=4,
MAMSR[D]=MAMSR[E]+edge_cost(D,E)=7,
MAMSR[B]=MIN(MAMSR[C],MAMSR[D])= 4.

3.4 Room Resizing

After collecting information from the above algorithms, we have got enough knowledge for our next step, resizing room. Of course, we move blocks to achieve these. Before a block move, we must determine if this move is valid.

Definition 2. *A block move is a valid move if that, after moving this block which may push other blocks, the total area and topology could remain unchanged.*

Theorem 1. *A move is a valid move if and only if the distance is less than MAMS of the corresponding direction of that block.*

Proof. Take a right directed move for example without loss of generality, firstly, MAMSR is the cumulative distance a block can be moved right due to the dead spaces that lie on the right. If the distance exceeds this value, the blocks rightmost may be pushed out of the right boundary which causes the area to be enlarged; on the opposite, if there are enough cumulated dead spaces on the right to handle this move, the total area could remain unchanged. After we move the block right, MAMSR is decreased by that distance and MAMSL is increased by that distance. For example in Fig. 5(a), move B right 3 and Move D right 6 are both valid move, however Move B right 8 or Move C right 5 are both invalid moves. Moving any block left will be invalid because all blocks' MAMSL is zero.

Since dead spaces are the only spaces for buffer sites, the ultimate objective of our block moving and room resizing is to redistribute dead spaces to make it more suitable for buffer requirements. Room resizing and dead space updating algorithm is shown as follows which takes moving right as an example.

```
1.  MoveRight (block A, distance i)
2.    If(A's right dead space is less than i)
3.      Foreach block B that lies right to A
4.        If( i >edge_cost(A,B))
5.           MoveRight(B, i-edge_cost(A,B));
6.        End if
7.        Allocate B's left dead-space
             to its left blocks(including A);
8.      End for.
9.    End if.
10.   Decrease width of A's right dead space by i;
11.   Increase width of A's left dead space by i;
12.   Update coordinates of block A;
13.   Update edge cost in horizontal graph;
```

This algorithm is also recursive that if a block wishes to move right, its right dead space width must be greater than the distance it wishes to move. If this is the fact, then moving will be easy and algorithm starts from step 10; if not, we have to push blocks that lie right and this will give this block larger room and enough right dead-space. We note that in step 7, giving left dead space to its left blocks will cause all its left blocks' right space to be broadened including "A" so that "A" will have enough right dead space for move and step 10 is able to go on.

3.5 Searching for Optimal Solution Using Simulated Annealing

Our final searching algorithm is based on Simulated Annealing (SA) to make each move effective for satisfying timing constraint. We start with an initial packing which is generated through simulated annealing(this is the first phase traditional simulated annealing which specialized on minimizing area and total wire length and it is quite different from the simulated annealing step in our algorithm). Based on this packing, we start to try moving blocks to optimize timing performance. We randomly find a block not on critical path, randomly select a direction it can be moved, and move it at a random distance which does not exceed corresponding MAMS. Then we update dead spaces and update the nets affected by the blocks having been moved. To analyze whether this move is an optimized move, we apply traditional buffer insertion algorithm on this packing. [3] displays details for the buffer-insertion algorithm which firstly calculate the number of buffers each net requires, get feasible region for each buffer and intersect buffer's feasible region with dead spaces. If the intersection is empty, the buffer will fail to be inserted. Only nets with no buffer need or

with all buffers successfully placed is taken as timing-constraint-satisfied. So the effectiveness of each move is evaluated by whether it could make more nets meet timing constraint. With the random move of each blocks, the final result through simulated annealing process could minimize the number of nets which violate timing closure. The SA algorithm is shown below.

```
1.   Find an initial packing which satisfies basic requirement
        such as area and wire length;
2.   Draw constraint graph and determine each blocks MAMS
        and dead-spaces;
3.   While(Temperature>min_temperature)
4.      Randomly select a block not on critical path and a
          direction whose corresponding MAMS is greater than zero;
5.      Move that block on that direction at a random distance
          less than MAMS;
6.      Update MAMS, dead-spaces and nets information;
7.      Apply buffer allocation algorithm;
8.      If optimized accept it; else accept with a probability;
9.      Decrease temperature;
10.   End while
```

4 Experimental Result

The buffer planning and optimization algorithm have been implemented on SUN Ultra-SPARC and benchmarks are MCNC and CSTC(generated by duplicating ami33 and ami49 blocks and nets) benchmarks. The values for parameters are based on a 0.18m technology in the NTRS97 roadmap[11]. Since we are concentrating on two-pin nets, we decompose each net to two-pin nets using minimum-spanning tree. Because these benchmarks have no timing information and are not fit for direct buffer evaluation, we compute optimal delay Topt under optimal buffer insertion [10] (the minimized delay if we insert enough buffers on optimized locations) and assign a constraint delay 1.05 and 1.20 times Topt to the net. We choose this delay assign method so that each net's timing is possible to be satisfied and some buffers are often necessary to achieve this . Since the timing closures are randomly generated, it is not easy to compare it directly with result shown in [2]. However, we have implemented all these algorithms to make comparison in Table 1. The first column "No blocks moved" displays results after applying buffer allocation algorithm on the initial packing after CBL without optimization. The second column "DS redis. within room" displays results on packing generated by algorithm introduced in [2] which moves block within its room. The third column displays results on packing generated by our algorithm which redistributes dead-space based on room resizing. "Met/Total" is the number of nets satisfying timing constraint compared to the total number of nets, "Time" is the time consumed by each algorithm and "Improved" is the improving percent of our algorithm compared to the other algorithms. From the table below, we could see that our algorithm has an average improvement of

3.3% compared to algorithm without optimization and has an improvement of 1.58% compared to algorithm introduced in [2].

Table 1. Experimental Results

	No blocks moved		DS redis. within room		Our algorithm		Improved
	Met/Total	Time	Met/Total	Time	Met/Total	Time	Perc%
AMI33	340/363	<1s	348/363	6s	353/363	19s	3.8%/1.4%
AMI49	443/545	<1s	454/545	15s	463/545	72s	4.5%/2%
CSTC132	1313/1434	1s	1329/1434	52s	1351/1434	120s	2.9%/1.6%
CSTC199	1898/2111	1s	1913/2111	64s	1938/2111	160s	2%/1.3%
Average							3.3%/1.58%

5 Conclusion

In this paper, on the base of dead space redistribution buffer-insertion algorithm introduced in [2], we extend it by resizing rooms to improve number of nets that satisfy timing closure without violating topology and total area. Our algorithm provides steps for determining a block's maximum allowable move space and organize the dead spaces systematically so that buffer planning could be improved gradually with simulated annealing. Experimental results show that our algorithm is efficient for resolving the problem of buffer planning. Additionally, our algorithm is fit for IP based SOC planning since it often requires improving timing performance without violating topology. However, because a move may cause many blocks to move and room resizing and dead space update is more complicate, a faster moving solution and faster buffer evaluation method is in great need, which will be our future work.

References

1. X.L. Hong, S. Dong, G. Huang, et al: Corner block list representation and its application to floorplan optimization. IEEE Trans. on Circuits and Systems II: Express Briefs, Vol. 51, No. 5 , 2004, pp.228 - 233.
2. S. Chen, X.L. Hong, S.Q. Dong, Y.C. Ma: A buffer planning algorithm based on dead space redistributio. ASP-DAC, 2003.
3. J. Cong, T. Kong, and D. Z. Pan: Buffer block planning for interconnect-driven floorplanning. IEEE/ACM ICCAD, 1999.
4. Y.C. Ma, X.L.Hong, S.Q. Dong, S. Chen, Y.C. Cai: An Integrated Floorplanning with an Efficient Buffer Planning Algorithm. ISPD, 2003.
5. P. Sarkar, C. K. Koh: Routability-driven repeater block planning for interconnect-centric floorplanning. ISPD, 2000.

6. X. Tang and D.F. Wong: Planning buffer locations by network flows. Intl. Symp. Physical Design, 2000, pp. 180-185.
7. F. F. Dragan, A. B. Kahng, et al: Provably good global buffering by multiterminal multicommodity flow approximation. ASP-DAC, 2001.
8. C. W. Sham, F. Y. Young: Routability driven floorplanner with buffer block planning. ISPD, 2002.
9. F. Ragiq, M. C. Jeske, H. H. Yang, N. Sherwani: Integrated floorplanning with buffer/channel insertion for bus-based microprocessor designs. ISPD, 2002.
10. C. J. Alpert and A. Devgan: Wire segmenting for improved buffer insertion. Proc. Design Automation Conf, pp. 588C593, June 1997.
11. Semiconductor Industry Association, National Technology Roadmap for Semiconductors, 1997.