

# A Fast Algorithm for Area Minimization of Slicing Floorplans

Weiping Shi, *Member, IEEE*

*Abstract*—The traditional algorithm for area minimization of slicing floorplans due to Stockmeyer has time and space complexity  $O(n^2)$  in the worst case. For more than a decade, it has been considered the best possible.

This paper presents a new algorithm of worst-case time and space complexity  $O(n \log n)$ , where  $n$  is the total number of realizations for the basic blocks, regardless whether the slicing is balanced or not. We also show  $\Omega(n \log n)$  is the lower bound on the time complexity of any area minimization algorithm. Therefore, the new algorithm not only finds the optimal realization, but also has the optimal running time.

*Keywords*—Physical design, floorplanning, area minimization, data structure.

## I. INTRODUCTION

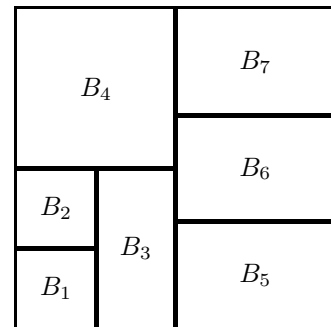
### A. Problem Description

Floorplan design is an important step in the physical design process of VLSI circuits [8]. A *floorplan*  $F$  is a subdivision of an enclosing rectangle by horizontal and vertical line segments into nonoverlapping rectangles. A rectangle not subdivided by any line segment is called a *basic block*. Fig. 1 shows two floorplans of seven basic blocks. Fig. 2 shows one floorplan with five basic blocks. To optimize the layout, we are allowed to move the line segments and choose the dimensions of the basic blocks. Two floorplans  $F_1$  and  $F_2$  are equivalent if for every pair of basic blocks, the above-below relation and left-right relation of the pair of blocks are the same in  $F_1$  and  $F_2$ . The floorplan in Fig 1(a) is equivalent to the floorplan in Fig. 1(b).

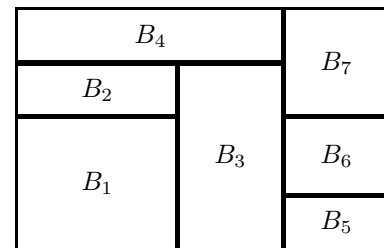
A floorplan  $F$  also associates each basic block  $B_i$  with a set of realizations  $R(B_i) \subset \mathbf{R}_+ \times \mathbf{R}_+$ , where  $\mathbf{R}_+$  is the set of positive real numbers. In the application, each line segment represents a partition of the circuit, each basic block  $B_i$  represents a basic module, and each set of realizations  $R(B_i)$  represent the widths and heights of the alternative implementations for the basic module  $B_i$ . If we choose one realization for each basic block  $B_i$ , then we have a *realization*  $\rho$  of the floorplan  $F$ . From  $\rho$ , we can easily compute a floorplan  $F^*$  equivalent to  $F$  such that the chosen realizations of the basic blocks fit into their associated rectangles in  $F^*$ , and the enclosing rectangle of  $F^*$  has the minimum height  $h(\rho)$  and width  $w(\rho)$  [14]. Given a floorplan  $F$ , the *area minimization problem* asks to find a realization  $\rho(F)$  of  $F$  such that the area  $h(\rho) \times w(\rho)$  is minimum among all realizations of  $F$ .

This research was supported by the National Science Foundation under grant MIP-9309120. Part of this paper was presented at the ACM/IEEE 1995 International Conference on Computer-Aided Design (ICCAD), San Jose, California, November 1995.

Weiping Shi is with the Department of Computer Science, University of North Texas, Denton, Texas 76203, USA.



(a)



(b)

Fig. 1. Two equivalent slicing floorplans.

Floorplans are classified as slicing or non-slicing. A floorplan is *slicing* if either it is a basic block, or there is a line segment that divides the floorplan into two subfloorplans such that each subfloorplan is slicing. Fig. 1 is a slicing floorplan of seven basic blocks. A floorplan is *non-slicing* if it is not a slicing floorplan. Fig. 2 is a non-slicing floorplan called a *wheel*. A slicing floorplan is represented by a rooted binary tree called a *slicing tree*. Fig. 3 is the slicing tree of the slicing floorplan of Fig. 1. Each non-leaf node in the slicing tree is labeled either  $h$  or  $v$ , specifying whether it is a horizontal or a vertical slice. Each leaf node corresponds to a basic block. For every internal node in the slicing tree, the subtree with that node being the root defines a subfloorplan.

Floorplans are also classified as hierarchical or non-hierarchical. A floorplan is *hierarchical* if it is constructed recursively by patterns of fixed sizes, such as a vertical slice, a horizontal slice, or a wheel. Otherwise, the floorplan is *non-hierarchical*. By default, all slicing floorplans are hierarchical. Since the complexity of the circuits keeps increasing. There is a strong need to pursue hierarchical

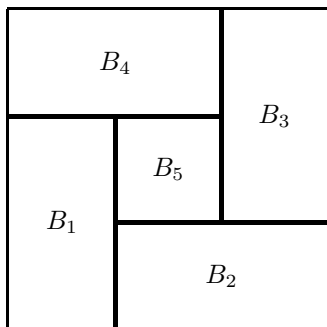


Fig. 2. A non-slicing floorplan called a wheel.

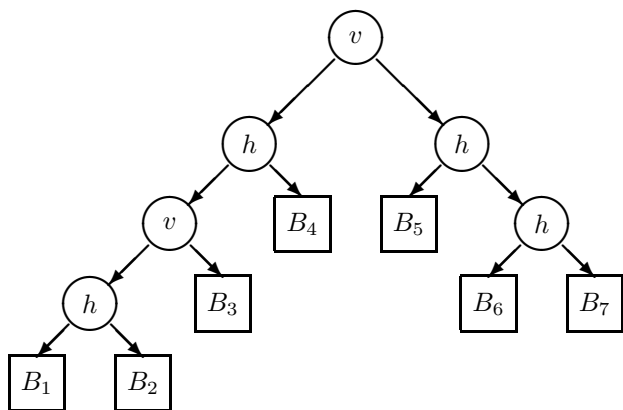


Fig. 3. The slicing tree for the slicing floorplan of Fig. 1.

floorplanning and placement.

### B. Previous Results

Area optimization of floorplans has been studied extensively. For slicing floorplans, Otten [10] and Stockmeyer [14] first proved the minimization problem can be solved in polynomial time. Stockmeyer, in his 1983 paper [14], presented the now widely cited algorithm of time complexity  $O(nd)$ , where  $n$  is the number of basic blocks,  $d$  is the depth of the slicing tree, and each basic block has two realizations. Thus, the time complexity is  $O(n^2)$  in the worst case, or  $O(n \log n)$  for balanced slicing with  $d = O(\log n)$ . The worst case time complexity is indeed  $\Theta(n^2)$  when the depth of the slicing tree is  $\Theta(n)$ . The space complexity is the same as the time complexity. For more than a decade, Stockmeyer's algorithm has been considered the fastest possible. Note that although one can minimize the depth of some part of the slicing tree if that part contains only vertical slices or only horizontal slices [7], some slicing trees cannot be balanced. If we extend Stockmeyer's algorithm to allow more than two realizations for each basic block, then the running time will be  $O(n^2)$  if the realizations for the basic blocks are not evenly distributed among the basic blocks.

For hierarchical non-slicing floorplans, Pan, Shi and Liu [12] recently proved the problem is NP-complete. For the non-slicing floorplan of Fig. 2, where each of the five basic blocks has  $k$  realizations, Wang and Wong [16] proposed an

$O(k^3 \log k)$  time algorithm, which was improved by Chen and Tollis [3] to  $O(k^2 \log k)$ . Pan, Shi and Liu [12] presented a simple  $O(k^2 \log k)$  time algorithm, proved  $\Omega(k^2)$  is the lower bound, and showed any algorithm faster than  $O(k^2 \log k)$  will settle the famous open problem of sorting the entries of the matrix  $X + Y = \{x_i + y_j \mid i, j = 1, 2, \dots, n\}$  [12]. For hierarchical non-slicing floorplans, branch-and-bound algorithms were developed by Wang and Wong [16], and pseudo-polynomial time algorithms were developed by Wang and Wong [17], and by Pan, Shi and Liu [12].

For non-hierarchical floorplans, Stockmeyer [14] proved the problem is strongly NP-complete. It is therefore unlikely that there exists a pseudo-polynomial time algorithm. Wimer, Koren and Cedernaumn [18], and Chong and Sahni [4] proposed branch-and-bound algorithms. Pan and Liu [11] developed algorithms for general floorplans that are approximately slicing.

Finally, we review other related research. Sarrafzadeh [13] investigated how to transform an arbitrary floorplan into a slicing one by slightly increasing the area. His result makes algorithms for slicing floorplans also applicable to non-slicing floorplans. Dai and Kuh [6], Lengauer and Muller [9], and Zimmermann [19] studied how to combine floorplanning and wiring to achieve good overall performance. For more information on floorplan minimization, see the book by Lengauer [8].

### C. Our Results

Throughout the rest of the paper, let the number of basic blocks be  $m$  and the total number of realizations for the basic blocks be  $n$ , where  $n \geq m$ .

In Section 2, we present a new algorithm for area optimization of slicing floorplans that runs in worst case time  $O(n \log n)$  regardless the number of basic blocks  $m$ , the depth of the slicing tree, or the distribution of  $n$  realizations among  $m$  basic blocks. The space complexity of the new algorithm is  $O(n \log n)$ , or  $O(n)$  if we only produce as output the width and height of the minimum realization instead of the composition. The new algorithm is an improvement of the  $O(n^2)$  time and space algorithm of Stockmeyer [14]. The new algorithm also generalizes Stockmeyer's algorithm to allow an arbitrary number of realizations for each basic block.

In Section 3, we prove  $\Omega(n \log n)$  is the lower bound for the running time of any area minimization algorithm even if there are only two basic blocks and  $n$  realizations for basic blocks. Therefore, our algorithm not only finds the optimal realization, but also has an optimal running time.

In Section 4, we discuss the implementation and simulation results. In Section 5, we conclude the paper. Since Stockmeyer's algorithm has become part of almost all hierarchical floorplanning algorithms [3], [4], [11], [12], [13], [16], [18], the new algorithm can improve the running time of these algorithms.

## II. NEW ALGORITHM

For any two realizations  $\rho_1(F)$  and  $\rho_2(F)$  of floorplan  $F$ , we say  $\rho_1(F)$  *dominates*  $\rho_2(F)$  if  $w(\rho_1(F)) \leq w(\rho_2(F))$  and  $h(\rho_1(F)) \leq h(\rho_2(F))$ . Intuitively,  $\rho_1(F)$  dominates  $\rho_2(F)$  if  $\rho_1(F)$  is no greater than  $\rho_2(F)$  in both height and width. For any floorplan  $F$ , the set of *nonredundant realizations* of  $F$ ,  $R(F)$ , is a set of realizations such that no realization in  $R(F)$  dominates any other realization in  $R(F)$ , and any realization of  $F$  is dominated by some realization in  $R(F)$ . Since our objective is to minimize the floorplan, it is obvious that for any floorplan  $F$ , we only need  $R(F)$ . This is true when  $F$  is the given floorplan, a subfloorplan, or a basic block.

We will generate all nonredundant realizations of the given floorplan  $F$ . The realization  $\rho$  that achieves the minimum area  $h(\rho) \cdot w(\rho)$  or minimum perimeter  $2h(\rho) + 2w(\rho)$  can be found in  $O(|R(F)|)$  time once we have  $R(F)$ . It seems we might reduce the time complexity if we just compute one minimum area realization. Unfortunately, the result in Section 3 says that to find the minimum area realization is as hard as to find all nonredundant realizations.

The following is an outline of the new algorithm. We will use a new data structure called realization trees to store the realizations. Realization trees and details of the algorithm will be explained in the following sections.

**Algorithm** FastAreaMin.

**Input:** A slicing floorplan  $F$ .

**Output:** All nonredundant realizations of  $F$  stored in a realization tree.

**Begin:**

- 1: **if**  $F$  is a basic block **then**  
     Create a realization tree  $T$  to store nonredundant realizations of  $F$ .  
     **return**  $T$ .
- 2: **if**  $F$  consists of two subfloorplans  $F_1$  and  $F_2$  sliced vertically **then**
  - 2.1: Recursively find nonredundant realizations of  $F_1$  and  $F_2$  and let the results be in realization trees  $T_1$  and  $T_2$  respectively.
  - 2.2: Find nonredundant realizations of  $F$  whose height is determined by  $F_2$ ;  
     Store the resulting realizations in list  $L$ .
  - 2.3: Find nonredundant realizations of  $F$  whose height is determined by  $F_1$ ;  
     Change  $T_1$  to store the resulting realizations.
  - 2.4: Insert  $L$  into  $T_1$  and delete redundant realizations from  $T_1$ ;  
     **return**  $T_1$ .
- 3: **if**  $F$  consists of two subfloorplans  $F_1$  and  $F_2$  sliced horizontally **then**  
     (same as 2.1 to 2.4 except replace “height” by “width”.)

**End of Algorithm.**

### A. Data Structure

We assume the readers have some familiarity with balanced search trees, such as AVL trees [15] and red-black

trees [5]. In general, a balanced binary search tree allows time  $O(\log k)$  for search, insertion and deletion of any key, where  $k$  is the number of nodes in the search tree. For our purpose, we need an additional property for the balanced search tree:

Given a balanced search tree of  $n_1$  nodes, the search and insertion of  $n_2$  ( $n_2 \leq n_1$ ) keys in sorted order can be performed in total time  $O\left(n_2 \log\left(1 + \frac{n_1}{n_2}\right)\right)$ .

*Lemma 1:* AVL trees and red-black trees satisfy the above property.

*Proof:* Consider the search and insertion of  $n_2$  keys  $x_1 \leq x_2 \leq \dots \leq x_{n_2}$  in a balanced search tree  $T$  of  $n_1$  nodes,  $n_1 \geq n_2$ . A brute-force solution is to search from the root of  $T$  for every  $x_i$ , resulting in a total time of  $O(n_2 \log n_1)$ . A more efficient strategy is to start from where we found the previous key  $x_{i-1}$ , then move up or down the tree to search for the next key  $x_i$ . For details of such an algorithm see Brown and Tarjan [2]. Their algorithm is for AVL trees but we can easily extend the algorithm for red-black tree. The running time of the algorithms are  $O\left(n_2 \log\left(1 + \frac{n_1}{n_2}\right)\right)$ . ■

For any set  $R$  of nonredundant realizations of a floorplan  $F$ , we use a *realization tree*  $T(R)$  to store  $R$ .  $T(R)$  is organized as a balanced search tree. For every realization  $\rho \in R$ , there is a unique node  $v(\rho) \in T(R)$ .  $T(R)$  has two keys, the height and the width, and is organized in increasing height order and also in decreasing width order. This is possible because the realizations in  $R$  are nonredundant. The search, insertion and deletion can be performed under either key. However, the height and width of a realization  $\rho$  are not necessarily stored explicitly in the corresponding node  $v(\rho)$ . Instead, the information is stored in the path from the root of  $T(R)$  to  $v(\rho)$ .

To see how the height and width are stored, take a look of the following segment of C program that defines the data structure of each realization tree node:

```
typedef struct rtnode { /* realization tree node */
    float h; /* part of height */
    float w; /* part of width */
    float ha; /* add to h of descendents */
    float wa; /* add to w of descendents */
    Comp *c; /* composition */
    Comp *ca; /* combine with c of descendents */
    struct rtnode *left; /* left subtree */
    struct rtnode *right; /* right subtree */
    char color; /* balance information */
    char dirty; /* whether to update */
    int size; /* no of nodes in tree */
} Rtnode;
```

For any realization  $\rho$ , let  $P(v(\rho))$  be the path from the root of the realization tree to  $v(\rho)$ , including the root and  $v(\rho)$ . The height and width of realization  $\rho$  are stored as follows:

$$h(\rho) = h(v(\rho)) + \sum_{u \in P(v(\rho))} ha(u),$$

$$w(\rho) = w(v(\rho)) + \sum_{u \in P(v(\rho))} wa(u).$$

Fig. 4 is a realization tree of five realizations of height and width (5,5), (6,4), (7,3), (8,2) and (9,1) respectively.

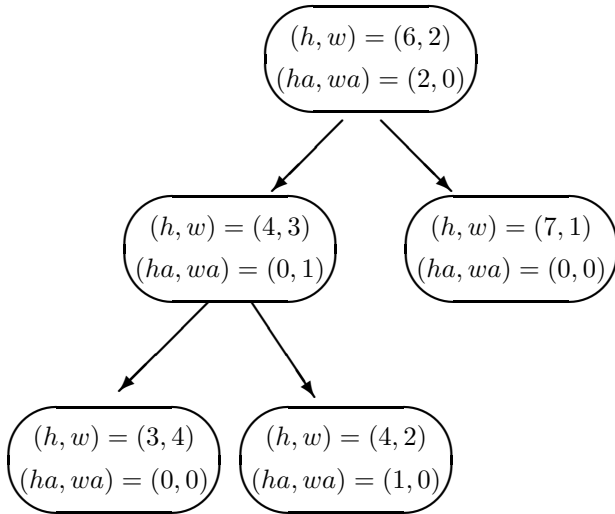


Fig. 4. A realization tree of five realizations (5,5), (6,4), (7,3), (8,2) and (9,1).

To see why we introduce **ha** and **wa**, consider a floorplan  $F$  consisting of two subfloorplans  $F_1$  and  $F_2$  sliced vertically. Assume we want to combine  $k$  realizations  $\sigma_1, \sigma_2, \dots, \sigma_k$  of  $F_1$  with one realization  $\rho$  of  $F_2$  to get  $k$  realizations of  $F$ . Furthermore, assume  $\sigma_1, \sigma_2, \dots, \sigma_k$  form a realization tree  $T_1$ , and the heights of  $\sigma_1, \dots, \sigma_k$  are greater than the height of  $\rho$ . To combine  $\sigma$ 's and  $\rho$ , the traditional algorithm must add the width of  $\rho$  to the widths of  $\sigma_1, \sigma_2, \dots, \sigma_k$  in time  $O(k)$ . However, we can add the width of  $\rho$  to the **wa** field of the root of  $T_1$  in time  $O(1)$ . Now  $T_1$  becomes a realization tree for  $F$ . The value **wa** will be propagated down and added to the width of each realization in  $T_1$  whenever that realization is accessed in the future. We delay the propagation until future access time to avoid repeated unnecessary updates.

There are other fields for each node  $v$  corresponding to a realization  $\rho$ . Field **left** points to a subtree containing nonredundant realizations of heights less than (and widths greater than) the height (width) of  $\rho$ . Field **right** points to a subtree containing nonredundant realizations of heights greater than (and widths less than) the height (width) of  $\rho$ . Field **color** contains the information for re-balancing the tree after insertion and deletion. Field **dirty** indicates whether **ha**, **wa**, and **ca** have been recently changed and have not been updated. Field **size** is an integer representing the number of realizations in the realization subtree with root  $v$ . To remember the composition of  $\rho$ , we use pointers **c** and **ca**. The data structure **Comp** for **c** and **ca** is defined as follows:

```

typedef struct comp { /* composition node */
    struct comp *lot; /* left or top */
    struct comp *rob; /* right or bottom */
}
    
```

} Comp;

Similar to the height and width, the composition is not necessarily stored in  $v(\rho)$ . Instead, the composition is stored in **ca**'s on the path from the root to  $v(\rho)$ , and in **c** of  $v(\rho)$ . For example, consider a subfloorplan  $F$  of three basic blocks  $B_1, B_2$  and  $B_3$  in Fig 1. Assume realization  $\rho$  of  $F$  contain realizations  $r_1, r_2$  and  $r_3$  for  $B_1, B_2$  and  $B_3$  respectively. Fig. 5 shows how pointers **ca** and **c** may represent the composition. Unfortunately, to deal with the composition is much more tedious than to deal with the height and width. Therefore, the reader can skip the details regarding the composition without affecting the understanding of the algorithm.

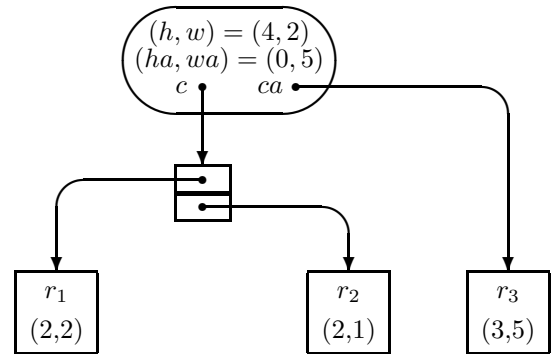


Fig. 5. The composition of a realization.

The search of realization trees is similar to the search of ordinary binary search trees, except when  $v$  is dirty, the values of **ha**, **wa** and **ca** are added to **h**, **w** and **c** and are propagated to the children of  $v$ . The following C program illustrates the search process and the propagation. As the name stands, function **height\_search(r, x)** searches a realization subtree of root **r** for a node  $v(\rho)$  with  $h(\rho) = x$ . For simplicity, we illustrate how to search from the root, though our algorithm will search from wherever we finished the last search. Function **update(r)** updates fields **h**, **w** and **c** of node **r**, and propagates **ha**, **wa** and **ca** to the children of **r**.

```

RTnode *height_search(r, x)
RTnode *r;
float x;
{
    if (r == NIL)
        return NIL;
    if (r->dirty == TRUE)
        update(r);
    if (r->h == x)
        return r;
    if (r->h > x)
        return height_search(r->left, x);
    return height_search(r->right, x);
}

update(r)
    
```

```

RTnode *r;
{
    Comp *p = (Comp *) malloc(sizeof(Comp));

    /* update left subtree */
    r->left->ha = r->left->ha + r->ha;
    r->left->wa = r->left->wa + r->wa;
    ...
    r->left->dirty = TRUE;

    /* update right subtree */
    ...

    /* update node r */
    r->h = r->h + r->ha; /* new height */
    r->w = r->w + r->wa; /* new width */
    r->ha = 0;
    r->wa = 0;
    p->lot = r->c;
    p->rob = r->ca;
    r->c = p; /* new composition */
    r->ca = NIL;
    r->dirty = FALSE;
}

```

### B. Algorithm Details

We now explain each step of Algorithm *FastAreaMin* in detail.

Step 1: If floorplan  $F$  is a basic block  $B_i$ , then let the number of realizations of  $B_i$  be  $k_i$ . We can sort all realizations of  $B_i$  in  $O(k_i \log k_i)$  time, delete redundant ones in  $O(k_i)$  time, and create a realization tree  $T$  for the nonredundant ones in  $O(k_i \log k_i)$  time. For every nonredundant realization  $\rho_j$  of  $B_i$ , we create a node  $v_j \in T$ , such that  $w = w(\rho_j)$ ,  $h = h(\rho_j)$ , and  $\mathbf{wa} = \mathbf{ha} = 0$ .

Step 2: Suppose floorplan  $F$  consists of two subfloorplans  $F_1$  and  $F_2$  sliced vertically as shown in Fig. 6.

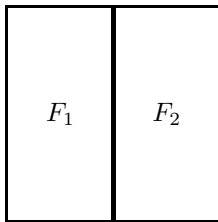


Fig. 6. A floorplan  $F$  with two subfloorplans  $F_1$  and  $F_2$ .

Step 2.1: We first recursively compute  $R(F_1)$  and  $R(F_2)$  and get their nonredundant realizations in realization trees  $T_1$  and  $T_2$ . Let the number of realizations in  $T_1$  and  $T_2$  be  $n_1$  and  $n_2$ . Assume without loss of generality  $n_1 \geq n_2$ , otherwise exchange  $T_1$  and  $T_2$ . Field `size` can help us to identify which tree contains more realizations in  $O(1)$  time.

Step 2.2: Now consider nonredundant realizations of  $F$  such that the heights are decided by  $F_2$ . We also include nonredundant realizations whose heights are de-

termined by both  $F_1$  and  $F_2$  simultaneously. For each realization  $\rho_i(F_2) \in R(F_2)$ , we want to find a realization  $\sigma_j(F_1) \in R(F_1)$  such that the height of  $\sigma_j$  is less than or equal to the height of  $\rho_i$  and the width of  $\sigma_j$  is the minimum among all such  $\sigma_j$ 's. In other words, we want to find index  $j$ :

$$j = \max_{1 \leq k \leq n_1} \{k \mid \sigma_k(F_1) \in R(F_1), h(\sigma_k(F_1)) \leq h(\rho_i(F_2))\}.$$

Given  $\rho_i$ , we can find the corresponding  $\sigma_j \in R(F_1)$  by searching  $T_1$ . Together,  $\rho_i(F_2)$  and  $\sigma_j(F_1)$  give a realization of  $F$  with height  $h(\rho_i(F_2))$  and width  $w(\rho_i(F_2)) + w(\sigma_j(F_1))$ . The composition of this new realization is represented by a pointer to `c` of  $v(\rho_i)$  and `c` of  $v(\sigma_j)$ .

To quickly generate all nonredundant realizations of  $F$  such that the heights are decided by  $F_2$ , we enumerate  $R(F_2)$  in increasing height order, and for every  $\rho_i(F_2) \in R(F_2)$ , search  $T_1$  for the corresponding  $\sigma_j(F_1)$ . Since  $\rho_i$ 's are in increasing height order,  $\sigma_j$ 's must be in nondecreasing height order. Therefore, the total time to enumerate  $R(F_2)$  is  $O(n_2)$ , and the total time to search  $T_1$  is  $O(n_2 \log(1 + \frac{n_1}{n_2}))$ . The newly generated realizations are stored in a temporary list  $L$  in increasing height order for later use. The size of  $L$  is at most  $n_2$ .

Step 2.3: We now consider nonredundant realizations of  $F$  such that the heights are decided by  $F_1$ . For each realization  $\rho_i(F_2) \in R(F_2)$ , we want to find realizations  $\sigma_j(F_1), \sigma_{j+1}(F_1), \dots, \sigma_l(F_1)$  in  $R(F_1)$  such that

$$\begin{aligned}
 j &= \min_{1 \leq k \leq n_1} \{k \mid \sigma_k \in R(F_1), h(\sigma_k(F_1)) > h(\rho_i(F_2))\}, \\
 l &= \max_{1 \leq k \leq n_1} \{k \mid \sigma_k \in R(F_1), h(\sigma_k(F_1)) < h(\rho_{i+1}(F_2))\}.
 \end{aligned}$$

This can be done through two searches of  $T_1$  using  $h(\rho_i(F_2))$  and  $h(\rho_{i+1}(F_2))$ . If no such  $j$  and  $l$  are found, then we repeat the above for the next realization  $\rho_{i+1} \in R(F_2)$ . Otherwise, we can form the following  $l - j + 1$  realizations of  $F$ :

$$\begin{aligned}
 \rho_i \text{ and } \sigma_j &: \text{height} = h(\sigma_j(F_1)) \text{ and} \\
 &\quad \text{width} = w(\sigma_j(F_1)) + w(\rho_i(F_2)), \\
 \rho_i \text{ and } \sigma_{j+1} &: \text{height} = h(\sigma_{j+1}(F_1)) \text{ and} \\
 &\quad \text{width} = w(\sigma_{j+1}(F_1)) + w(\rho_i(F_2)), \\
 &\quad \dots \\
 \rho_i \text{ and } \sigma_l &: \text{height} = h(\sigma_l(F_1)) \text{ and} \\
 &\quad \text{width} = w(\sigma_l(F_1)) + w(\rho_i(F_2)).
 \end{aligned}$$

To store the newly generated realizations, we change the fields of nodes  $v(\sigma_j), \dots, v(\sigma_l)$  in  $T_1$ . Step by step, we will turn  $T_1$  into a realization tree of  $F$ . However, we cannot afford  $O(l - j)$  time to explicitly change the nodes since otherwise we would have the same time complexity as Stockmeyer's. Instead, we change fields `ha`, `wa` and `ca` as we explained earlier. Fig. 7 illustrates the general situation of nodes  $v(\sigma_j), v(\sigma_{j+1}), \dots, v(\sigma_l)$ .

Nodes  $v(\sigma_j), v(\sigma_{j+1}), \dots, v(\sigma_l)$  form a continuous interval in realization tree  $T_1$ . Let  $nca(\sigma_j, \sigma_l)$  be the *nearest common ancestor* of  $v(\sigma_j)$  and  $v(\sigma_l)$ . Let the *left boundary* be

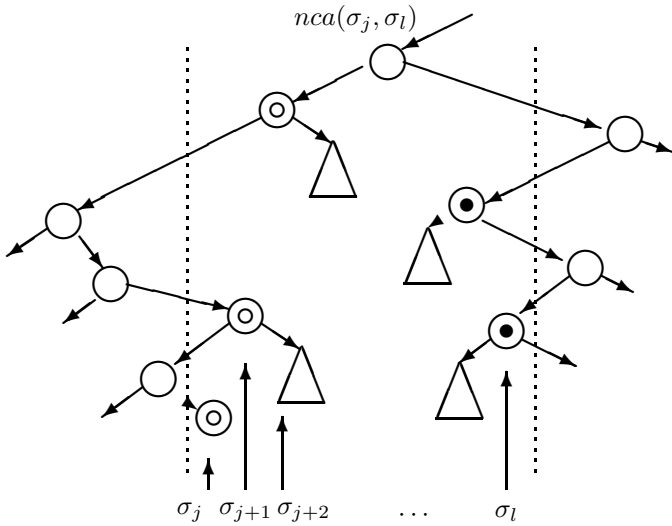


Fig. 7. Nodes  $v(\sigma_j), v(\sigma_{j+1}), \dots, v(\sigma_l)$  in realization tree  $T_1$ .

the set of nodes  $u$  such that  $u$  is on the path from  $v(\sigma_j)$  to  $nca(\sigma_j, \sigma_l)$  and the height of  $u$  is greater than or equal to the height of  $\sigma_j$ . In Fig. 7, nodes with circles inside are the left boundary. Let pointer  $r$  point to the node for  $\rho_i(F_2)$ . For every left boundary node pointed by  $u$  in  $T_1$  including  $v(\sigma_j)$  but not  $nca(\sigma_j, \sigma_l)$ , we make the following changes:

```
u->w = u->w + r->w;
u->right->wa = u->right->wa + r->w;
```

```
p = (Comp *) malloc(sizeof(Comp));
p->lot = u->c;
p->rob = r->c;
u->c = p;
```

```
p = (Comp *) malloc(sizeof(Comp));
p->lot = u->right->ca;
p->rob = r->c;
u->right->ca = p;
```

```
u->right->dirty = TRUE;
```

Similarly, let the *right boundary* be the set of nodes  $u$  such that  $u$  is on the path from  $v(\sigma_l)$  to  $nca(\sigma_j, \sigma_l)$  and the height of  $u$  is less than or equal to the height of  $\sigma_l$ . In Fig. 7, nodes with dots inside are the right boundary. For every right boundary node  $u$ , including  $v(\sigma_l)$  but not  $nca(\sigma_j, \sigma_l)$ , we make the following changes:

```
u->w = u->w + r->w;
u->left->wa = u->left->wa + r->w;
...
u->left->dirty = TRUE;
```

Let pointer  $nca$  point to  $nca(\sigma_j, \sigma_l)$ . We make the following changes:

```
nca->w = nca->w + p->w;
```

```
p = (Comp *) malloc(sizeof(Comp));
```

```
p->lot = nca->c;
p->rob = r->c;
nca->c = p;
```

Among the newly generated realizations, no one dominates another. Since  $\rho_i(F_2)$ 's are in increasing height order, the total search time for  $\sigma_j(F_1)$ 's and  $\sigma_l(F_1)$ 's is  $O(n_2 \log(1 + \frac{n_1}{n_2}))$ . It is easy to see all the nearest common ancestors can be found in the same time. The total number of nodes in the left and right boundaries, for all intervals, is at most the number of nodes visited if we go up and down a balanced search tree to search  $n_2$  keys in sorted order. From Lemma 1, the total time to update fields  $w$ ,  $wa$ ,  $c$  and  $ca$  for all intervals is  $O(n_2 \log(1 + \frac{n_1}{n_2}))$ .

Step 2.4: We insert the list  $L$  of size  $O(n_2)$  generated in Step 2.2 into the realization tree  $T_1$  of size  $O(n_1)$  obtained in Step 2.3. In increasing height order, we search  $T_1$  using the heights of realizations in  $L$ . Consider a realization  $\rho_i(F)$  in  $L$  and realizations  $\rho_j(F), \rho_{j+1}(F)$  in  $T_1$ , where

$$h(\rho_j(F)) \leq h(\rho_i(F)) \leq h(\rho_{j+1}(F)).$$

If  $\rho_i$  in  $L$  is dominated by  $\rho_j$  in  $T_1$ , then we ignore  $\rho_i$ . If  $\rho_{j+1}$  in  $T_1$  is dominated by  $\rho_i$  in  $L$ , then insert  $\rho_i$  to  $T_1$  and delete  $\rho_{j+1}$  from  $T_1$ . If neither of the above happens, then insert  $\rho_i$  into  $T_1$ . When we finish,  $T_1$  becomes the realization tree for  $F$ ,

Since there are  $O(n_2)$  searches and  $O(n_2)$  insertions, both in sorted order, the total time for search and insertion is  $O(n_2 \log(1 + \frac{n_1}{n_2}))$ . In Step 2.2 and 2.3, there can be  $O(n_1)$  deletions.

Step 3: If floorplan  $F$  consists of two subfloorplans sliced horizontally, then exchange the words "height" with "width",  $h$  with  $w$ , etc, to the above discussion and everything should follow.

*Theorem 1:* Algorithm *FastAreaMin* correctly finds all nonredundant realizations of  $F$ .

*Proof:* Due to the slicing property of the floorplan, realizations in subfloorplans  $F_1$  and  $F_2$  can only be treated as basic blocks when we compute the nonredundant realizations of  $F$  [10], [14].

Clearly, for every nonredundant realization  $\rho$  of  $F$ , if  $h(\rho)$  is determined by  $F_2$ , or determined by both  $F_1$  and  $F_2$ , then  $\rho$  will be put in  $L$  during Step 2.2. For every nonredundant realization  $\rho$  of  $F$ , if  $h(\rho)$  is determined by  $F_1$ , then  $\rho$  will be in  $T_1$  at the end of Step 2.3. No nonredundant realization can be deleted during Step 2.3 and Step 2.4. Therefore, all nonredundant realizations of  $F$  will appear in  $T_1$  in Step 2.4. On the other hand, Step 2.4 guarantees all realizations in the final realization tree are nonredundant. Therefore, the algorithm works correctly. ■

### C. Time and Space Complexity

We first prove a fact we need later in the estimation of the time complexity.

*Lemma 2:* Given a floorplan  $F$  with  $n$  realizations for the basic blocks, there are at most  $n$  nonredundant realizations for  $F$ .

*Proof:* By induction on the number of basic blocks. If  $F$  is a basic block, then the lemma is clearly true. Otherwise, let  $F$  contain two subfloorplans  $F_1$  and  $F_2$  sliced vertically, where  $F_1$  and  $F_2$  contain  $n_1$  and  $n_2$  realizations for the basic blocks respectively,  $n = n_1 + n_2$ . From the induction hypothesis,  $F_1$  and  $F_2$  have at most  $n_1$  and  $n_2$  nonredundant realizations respectively. The height of each realization of  $F$  is decided either by  $F_1$  or by  $F_2$ . If the height of a realization of  $F$  is decided by a realization of  $F_1$ , there is at most one choice for the realization of  $F_2$ , and vice versa. Therefore, there are at most  $n_1 + n_2 = n$  nonredundant realization for  $F$ . ■

*Theorem 2:* Algorithm *FastAreaMin* finds all nonredundant realizations in worst case time and space  $O(n \log n)$ , where  $n$  is the number of realizations for the basic blocks.

*Proof:* Let  $\mathcal{T}(n)$  be the worst case time cost of the algorithm on search and insertion operations only, where  $n$  is the number of realizations for the basic blocks. From Lemma 2, there are at most  $n$  nonredundant realizations for the floorplan. Therefore, we have the following recurrence relation:

$$\mathcal{T}(n) \leq \begin{cases} cn \log n & \text{if } F \text{ is a basic block,} \\ \max \left\{ \mathcal{T}(n_1) + \mathcal{T}(n_2) + cn_2 \log \left( 1 + \frac{n_1}{n_2} \right) \right\} & \text{if } F \text{ consists of } F_1 \text{ and } F_2. \end{cases}$$

where  $c$  is a constant,  $n_1$  and  $n_2$  are the number of realizations of basic blocks of  $F_1$  and  $F_2$  respectively, and the maximum is taken over all  $n_1, n_2$  such that  $n_1 + n_2 = n$  and  $n > n_1 \geq n_2 > 0$ . We prove by induction that

$$\mathcal{T}(n) \leq cn \log n. \quad (1)$$

Obviously  $\mathcal{T}(1) = 0 \leq c \cdot 1 \log 1$ . Assume (1) is true for all  $k < n$ , then

$$\begin{aligned} \mathcal{T}(n) &\leq \max_{\substack{n_1 \geq n_2 \\ n_1 + n_2 = n}} \left\{ \mathcal{T}(n_1) + \mathcal{T}(n_2) + cn_2 \log \left( 1 + \frac{n_1}{n_2} \right) \right\} \\ &\leq \max_{\substack{n_1 \geq n_2 \\ n_1 + n_2 = n}} \{ cn_1 \log n_1 + cn_2 \log n_2 + \\ &\quad + cn_2 \log \left( 1 + \frac{n_1}{n_2} \right) \} \\ &= \max_{\substack{n_1 \geq n_2 \\ n_1 + n_2 = n}} \{ cn_1 \log n + cn_2 \log(n_2 + n_1) \} \\ &= cn \log n. \end{aligned}$$

Therefore, the total time used for search and insertion is  $O(n \log n)$ . To show the total time for deletion is also  $O(n \log n)$ , we use an argument known as the amortization. Each deletion uses at most  $O(\log n)$  time for re-balancing. (AVL trees use  $O(\log n)$  time while red-black trees use only  $O(1)$  time.) We claim there are at most  $n$  deletions for the whole algorithm. This is because if  $k$  nodes are deleted in Step 2.3 or 2.4 of the algorithm, then the number of nonredundant realizations of  $F$  is at most  $n - k$ . From Lemma 2 there are at most  $n$  nonredundant realization for floorplan  $F$ . Therefore, there can be at most  $n$  deletions.

The space complexity is bounded by the time complexity which is  $O(n \log n)$ . However, if we just compute as output the minimum area instead of the composition of the realization, then we can reduce the space cost to  $O(n)$  by omitting fields  $c$ ,  $ca$  and related composition pointers. ■

It worths to point out that using realization trees only, we reduce the time complexity from Stockmeyer's  $O(n^2)$  to  $O(n \log^2 n)$ . The efficient search algorithm described in Lemma 1 helps us to further reduce the time complexity to  $O(n \log n)$ .

Once we find the set of all nonredundant realizations  $R(F)$ , the minimum area realization can be found in  $O(|R(F)|) = O(n)$  time. Therefore, the area minimization problem can be solved in  $O(n \log n)$  time.

### III. LOWER BOUND

Ben-Or [1] proved the following problem requires  $\Omega(n \log n)$  algebraic operations, where algebraic operations include  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\sqrt{\quad}$ ,  $=$ ,  $>$ ,  $\geq$ , etc.

*Set Disjointness Problem.* Given two sets of positive real numbers  $X = \{x_1, x_2, \dots, x_n\}$  and  $Y = \{y_1, y_2, \dots, y_n\}$ , determine whether there are indices  $i$  and  $j$  such that  $x_i = y_j$ .

*Theorem 3:* The area minimization problem requires  $\Omega(n \log n)$  algebraic operations, even if there are only two basic blocks, where  $n$  is the number of realizations for the basic blocks.

*Proof:* We show a reduction from the set disjointness problem. Given an instance of the set disjointness problem  $X$  and  $Y$ , construct a floorplan  $F$  with only two basic blocks  $B_1$  and  $B_2$  sliced vertically, and associate the following realizations to  $B_1$  and  $B_2$ :

$$\begin{aligned} R(B_1) &= \{(x_1, 1/x_1), (x_2, 1/x_2), \dots, (x_n, 1/x_n)\}, \\ R(B_2) &= \{(y_1, 1/y_1), (y_2, 1/y_2), \dots, (y_n, 1/y_n)\}. \end{aligned}$$

It is easy to see the minimum area of  $F$  is 2 if and only if there exist indices  $i$  and  $j$  such that  $x_i = y_j$ . Since the set disjointness problem requires  $\Omega(n \log n)$  algebraic operations, the area minimization problem also requires  $\Omega(n \log n)$  algebraic operations. ■

### IV. SIMULATION

Both Stockmeyer's algorithm and a simplified version of the new algorithm are implemented by the author in C on a SPARC 5 running SunOS 4.1.3. Stockmeyer's algorithm is about 300 lines of source code, while the new algorithm is about 800 lines of source code, including all the data structure operations. Red-black trees, instead of AVL trees, are used in the latest implementation. Simulation results indicate for unbalanced floorplans, the new algorithm is much faster than Stockmeyer's algorithm for  $n \geq 30$ . Table 1 shows for unbalanced floorplans of  $n = 100$  to 1000, the new algorithm is 3 to 140 times faster than Stockmeyer's algorithm. Each basic block contains two realizations in order to compare with Stockmeyer's algorithm. The running time does not include the time to read the floorplan from the input file and to print the final realizations to the

output file, and is averaged over 100 executions. For balanced floorplans, however, the new algorithm uses twice as much time as Stockmeyer's algorithm, for all values of  $n$ . This is due to the data structure overhead.

No. of Basic Blocks ( $n$ )	Stockmeyer's Alg. (sec)	New Alg. (sec)
100	0.06	0.02
200	0.34	0.04
300	1.00	0.06
400	2.21	0.07
500	4.06	0.10
600	6.81	0.12
700	10.48	0.14
800	15.30	0.16
900	22.10	0.18
1000	29.41	0.21

TABLE I

SIMULATION RESULTS FOR LARGE UNBALANCED FLOORPLANS.

## V. CONCLUSION

We have presented a new algorithm of worst case time  $O(n \log n)$  and space  $O(n \log n)$ , or  $O(n)$  depending on the output requirement, for area optimization of slicing floorplans, where  $n$  is the total number of realizations for the basic blocks. This is an improvement, both in time and in space, of the widely used Stockmeyer's  $O(n^2)$  time and  $O(n^2)$  space algorithm [14]. Our main idea is a new data structure for storing and updating realizations. We also prove  $\Omega(n \log n)$  is the lower bound on the running time of any area optimization algorithm of slicing floorplans even if there are only two basic blocks.

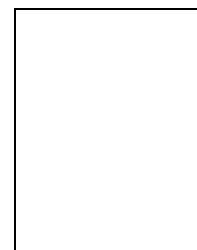
In real applications, the slicing tree may be balanced, but the number of realizations for basic blocks are not necessarily balanced, since the basic blocks are often implemented with different design styles. In addition, the number of nonredundant realizations for subfloorplans are not necessarily balanced. The new algorithm works best for these unbalanced cases. Some algorithms [13] can transform an arbitrary non-slicing floorplan into a slicing one, but the floorplan can be unbalanced. Since most algorithms for non-slicing floorplans use as a subroutine an algorithm for slicing floorplans [3], [4], [11], [12], [13], [16], [18], the new algorithm may be used to speed up these algorithms. The data structure and the delayed update idea can be used by some of the branch-and-bound algorithms [4], [16], [18] to reduce the running time and space as well.

## ACKNOWLEDGMENTS

The author thanks Larry Stockmeyer for constructive comments on an earlier version of the paper, and Steve Tate for discussions.

## REFERENCES

- [1] M. Ben-Or, "Lower bounds for algebraic computation trees," *Proc. 15th Annual ACM Symposium on Theory of Computing*, pp. 80–86, 1983.
- [2] M. R. Brown and R. E. Tarjan, "A fact merging algorithm," *Journal of ACM*, Vol. 26, No. 2, April 1979, pp. 211–226.
- [3] C. H. Chen and I. G. Tollis, "Area optimization of spiral floorplans," *Journal of Circuits, Systems and Computers*, Vol. 3, No. 4, pp. 833–857, 1993.
- [4] K. Chong and S. Sahni, "Optimal realizations of floorplans," *IEEE Trans. on Computer-Aided Design*, Vol. 12, No. 6, pp. 793–801, 1993.
- [5] T. Corman, C. E. Leiserson and R. Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.
- [6] W.-M. Dai and E. S. Kuh, "Simultaneous floor planning and global routing for hierarchical building block layout," *IEEE Trans. on Computer-Aided Design*, Vol. 6, No. 5, pp. 828–837, 1987.
- [7] M. C. Golumbic, "Combinatorial merging," *IEEE Trans. on Computers*, Vol. C-25, pp 1164–1167, 1976.
- [8] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, New York, Wiley, 1990.
- [9] T. Lengauer and R. Muller, "Robust and accurate hierarchical floorplanning with integrated global wiring," *IEEE Trans. on Computer-Aided Design*, Vol. 12, No. 6, pp. 802–809, 1993.
- [10] R. H. J. M. Otten, "Efficient floorplan optimization." In *Proc. International Conference on Computer Design: VLSI in Computers*, pp. 499–502, 1983.
- [11] P. Pan and C. L. Liu, "Area minimization for floorplans," *IEEE Trans. on Computer-Aided Design*, Vol. 14, No. 1, pp. 123–132, 1995.
- [12] P. Pan, W. Shi and C. L. Liu, "Area minimization for hierarchical floorplans," *Algorithmica*, Vol. 15, No. 6, pp. 550–571, 1996.
- [13] M. Sarrafzadeh, "Transforming an arbitrary floorplan into a sliceable one," *Proc. International Conference on Computer-Aided Design*, pp. 386–389, 1993.
- [14] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Information and Control*, Vol. 57, pp. 91–101, 1983.
- [15] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM Press, Philadelphia, PA, 1983.
- [16] T.-C. Wang and D. F. Wong, "Optimal floorplan area optimization," *IEEE Trans. on Computer-Aided Design*, Vol. 11, No. 8, pp. 992–1002, 1992.
- [17] T.-C. Wang and D. F. Wong, "A note on the complexity of Stockmeyer's floorplan optimization technique," *Algorithmic Aspects of VLSI Layout*, ed. M. Sarrafzadeh and D. T. Lee, Lecture Notes Series on Computing, Vol. 2, World Scientific Publishers, 1993, pp. 309–320.
- [18] S. Wimer, I. Koren, and I. Cederbaum, "Optimal aspect ratios of building blocks in VLSI," *IEEE Trans. on Computer-Aided Design*, Vol. 8, No 2, pp. 139–145, 1989.
- [19] G. Zimmermann, "A new area and shape function estimation technique for VLSI layout," *Proc. 25th Design Automation Conference*, pp. 60–95, 1988.



**Weiping Shi** (S'91-M'92) received the B.S. and M.S. degrees in computer science from Xi'an Jiaotong University, China, in 1982 and 1984 respectively, and Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1992. Since 1992, he has been an Assistant Professor in the Department of Computer Science, University of North Texas. His research interests include computer-aided design of VLSI circuits, defect and fault tolerance of VLSI systems, and design and analysis of algorithms.