Optimal Folding Of Standard And Custom Cells *

Venkat Thanvantri and Sartaj Sahni
Department of CIS,
University of Florida,
Gainesville, FL-32611

Abstract

We study the problem of folding an ordered list of standard and custom cells into rows of a chip so as to minimize either the routing area or the total chip area. Nine versions of the folding problem are formulated and fast polynomail time algorithms are obtained for each. Two of our formulations correspond to problems formulated in [PAIK93] for the folding of a stack of bit-slice components. Our algorithms for these two formulations are asymptotically superior to those of [PAIK93].

Keywords and Phrases

standard cell folding, custom cell folding, layout area

*This research was supported in part by the National Science Foundation under the grant MIP 91-03379 and the Army Research Office under grant DAA H04-95-1-0111.

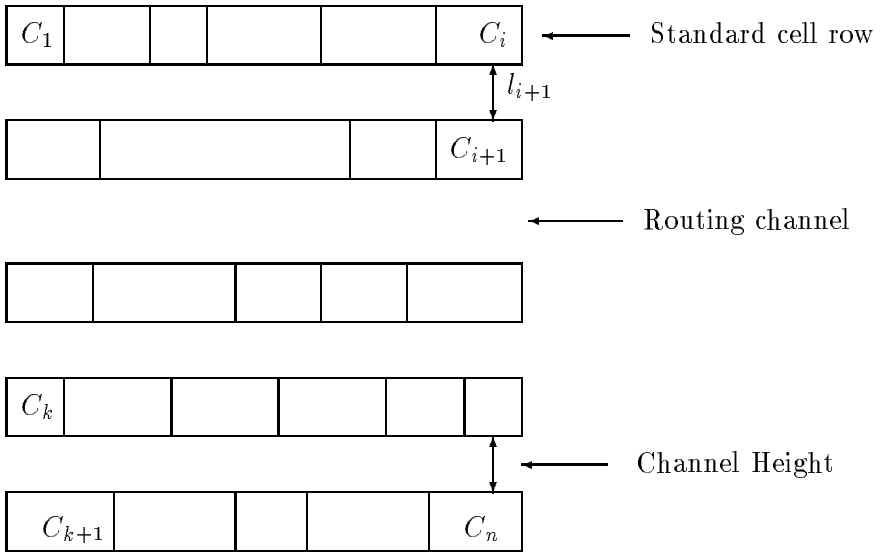

Figure 1: Standard cell Architecture

1 Introduction

Standard cell and gate array design styles are characterized by a row (column) organization of the layout. The layout area is divided into a number of parallel rows separated by routing channels as shown in Figure 1. The layout problem is generally divided into two independent subtasks: placement and routing. In the placement step the appropriate locations and orientations of the standard cells are decided. In the routing step, the required connections are added.

One approach to placement is linear ordering with folding [SHRA88, KANG83, COX80]. In this approach, the placement is divided into two distinct steps. The first is linear ordering in which an order of the modules is determined so as to minimize the connection length or minimize maximal density of connections for modules positioned in one line. The folding step maps the linear order into the row structure of the chip. The linear ordering problem is NP-hard and heuristic strategies are discussed in [SHRA88] to minimize the connection length as well as maximal density of connections. The greedy strategy is adopted in [SHRA88] for folding the ordered modules.

In this paper, we consider only the second step of the placement approach just described. We begin with an ordered component list $C_1, C_2, \cdots, C_n$ and develop algorithms to fold this list into rows. If the list is folded at C_i, then the component C_i is in one row and C_{i+1} is in the next. If the list is folded at C_i and C_j and at no component C_k for $i < k < j$, then components $C_{i+1}, \cdots, C_j$ are in the same row. Suppose the list is folded at C_i. The channel height needed between the rows containing C_i and C_{i+1} may be estimated [HEL77] using the number of nets that have a pin in one of the components $C_1, \cdots, C_i$ as well as in one of the components $C_{i+1}, \cdots, C_n$. Let this height estimate be l_i, $1 \leq i < n$. Let $l_n = 0$.

We study the following folding problems:

1. Standard cell folding to minimize total routing channel area subject to a chip width constraint W. Since each routing channel has the same width, the chip area assigned for routing is minimized when the sum of the channel heights is minimized. This problem is solved in $O(n)$ time using dynamic programming (Section 3.1). Note that whenever we use the term chip area, we could instead use subchip area.

2. Standard cell folding to minimize chip area subject to a chip width constraint W. In this problem both routing area and the area assigned for the components is considered. Since the chip width is fixed at W, area minimization is equivalent to minimizing chip height. In Section 3.1, we use dynamic programming to obtain an $O(n)$ algorithm for this problem.

3. Standard cell folding to minimize total routing area subject to a total routing channel height constraint H. This problem differs from problem 1 only in that the total height of the routing channels is fixed at H, and their width is variable rather than the routing channels having variable total height and fixed width W. In Section 3.2, we show how to solve this problem in $O(n \log n)$ time.

4. Standard cell folding to minimize chip area subject to a chip height constraint H. This problem is solved in $O(n \log n)$ time in Section 3.2.

5. Standard cell folding using equal height channels of width W. We are to find a folding that uses channels of minimum height. Among all such foldings, one that uses the fewest number of routing channels (and hence fewest number of component rows) is to be found. In Section 4.1, we develop an $O(n \log n)$ time algorithm for this problem. However, for most practical instances, the algorithm has run time $O(n)$.

6. Standard cell folding using equal height routing channels of width W. Find a folding that minimizes the total chip area. This can be done in $O(n^2)$ time (see Section 4.2).

7. Standard cell folding using equal height channels and a chip of height H. The folding should minimize the total chip area. Our algorithm for this problem can be found in Section 4.3. Its complexity is $O(n^2)$.

8. Custom cell folding to minimize total chip area subject to a chip width W. Note that in standard cell layout, all cells/components/modules have the same height and may have variable widths. In custom cell layout, the cells may differ in both height and width. We assume that the cell row height is set to be the height of the tallest cell assigned to that row. In Section 5.1, we develop an $O(n \log n)$ algorithm for this problem.

9. Custom cell folding to minimize total chip area subject to a chip height constraint H. We solve this problem in Section 5.2 using an algorithm of complexity $O(n \log^2 n)$.

We note that problem 8 has been studied previously in [PAIK93] in the context of bit slice stack folding. The algorithm developed there has complexity $O(n^2)$ while ours has complexity $O(n \log n)$. Problem 9 has also been studied in [PAIK93]. Our $O(n \log^2 n)$ algorithm is an improvement over the $O(n^2 \log n)$ algorithm developed in [PAIK93].

Since some of our algorithms use the parametric search technique, we begin, in Section 2, with an overview of this.

2 Parametric Search

In this section, we provide an overview of the parametric search method of Frederickson [FRED92] which uses developments by Frederickson and Johnson [FRED83, FRED84] and Frederickson [FRED91]. This overview has been tailored to suit our application here and is not as general as that provided in [FRED83, FRED84, FRED91, FRED92].

Assume that we are given a sorted matrix of $O(n^2)$ candidate values M_{ij}, $1 \leq i, j \leq n$. By sorted, we mean that

$$M_{ij} \leq M_{i,j+1}, 1 \leq i \leq n, 1 \leq j < n$$
$$\text{and } M_{ij} \leq M_{i+1,j}, 1 \leq i < n, 1 \leq j \leq n$$

The matrix is provided implicitly. That is, we are given a way to compute M_{ij}, in constant time, for any value of i and j. We are required to find the least M_{ij} that satisfies some criterion F. The criterion F has the property that if $F(x)$ is not satisfied, then $F(y)$ is not satisfied (i.e., it is infeasible) for all $y \leq x$. Similarly, if $F(x)$ is satisfied (i.e., it is feasible), then $F(y)$ is feasible for all $y \geq x$. In a parametric search, the minimum M_{ij} that satisfies F is found by trying out some of the M_{ij}'s. As different M_{ij}'s are tried, we maintain two values λ_1 and λ_2, $\lambda_1 < \lambda_2$ with the properties:

(a) $F(\lambda_1)$ is infeasible.

(b) $F(\lambda_2)$ is feasible.

Initially, $\lambda_1 = 0$ and $\lambda_2 = \infty$ (we assume F is such that $F(0)$ is infeasible, $F(\infty)$ is feasible, and $M_{ij} > 0$ for all candidate values). To determine the next candidate value to try, we begin with the matrix set $S = \{M\}$. At each iteration, the matrices in S are partitioned into four equal sized matrices (assume, for simplicity, that n is a power of 2). As a result of this, the size of S becomes four times its previous size. Next, a set T comprised of the largest and smallest elements from each of the matrices in S is constructed. The median of T is the candidate value x to try next. The following possiblities exist for x and $F(x)$:

(1) $x \leq \lambda_1$. Since $F(\lambda_1)$ is infeasible, $F(y)$ is infeasible for all $y \leq \lambda_1$. So, $F(x)$ is infeasible.

(2) $x \geq \lambda_2$. Now, $F(x)$ is feasible.

(3) $\lambda_1 < x < \lambda_2$. $F(x)$ may be feasible or infeasible. This is determined by computing $F(x)$. If x is feasible, λ_2 is set to x. Otherwise, λ_1 is set to x.

Following the update (if any) of λ_1 or λ_2 resulting from trying out the candidate value x, all matrices in S that do not contain candidate values y in the range $\lambda_1 < y < \lambda_2$ may be eliminated from S.

A more precise statement of the search process is given by procedure *PSEARCH* (Figure 2). This procedure may be invoked as *PSEARCH*($\{M\}$,0,∞,x,0). *dimension* is the current number of rows or columns in each matrix of S and $finish$ is a stopping rule. The search for the minimum candidate that satisfies F is terminated when the number of remaining candidates is $\leq finish$. If $\lambda_2 = \infty$ when *PSEARCH* terminates, then none of the candidate values is feasible. If λ_2 is finite, then it is the smallest candidate that is feasible.

```
Procedure PSEARCH(S,λ1,λ2,dimension,finish);
   repeat
      if dimension > 1 then [ replace each matrix in S by
                                 four equal sized submatrices;
                                 dimension := dimension/2 ]
      for i := 1 to 3 do
      begin
         if dimension = 1 then
            [ Let T be the multiset of values in all matrices of S; ]
         else
            [ Let T be the multiset obtained by selecting the largest
              and smallest values from each matrix of S; ]
         x := median(T);
         if (λ1 < x < λ2) then
            if F(x) is feasible then λ2 := x
            else λ1 := x;
         Eliminate from S all matrices that have no values
         such that λ1 < x < λ2;
      end;
   until dimension^2 * |S| ≤ finish;
end; {PSEARCH}
```

Figure 2: Procedure for parametric search. (Restricted version of Procedure $MSEARCH$ of [FRED92].)

Since we have assumed n is a power of two, each time a matrix is divided into four, the submatrices produced are square and have dimension that is also a power of 2. Since M is provided implicitly, each of its submatrices can be stored implicitly. For this, we need merely record the matrix coordinates (indices) of the top left and bottom right elements (actually, the latter can be computed from the former using the submatrix dimension). The multiset T required on each iteration of the **for** loop is easy to construct because of the fact that M is sorted. Note that since M is sorted, all of its submatrices are also sorted. Consequently, the largest element of each submatix is in bottom right corner and the smallest is in the top left corner. These elements can therefore be determined in constant time per matrix of S.

Theorem 1 : *[FRED92] The number of feasibility tests F performed by procedure PSEARCH when started with $S = \{M\}$, M an $n \times n$ sorted matrix that is provided implicitly is $O(\log n)$ and the total time spent obtaining the candidates for feasibility test is $O(n)$.* □

Corollary 1 : *Let $t(n)$ be the time needed to determine if $F(x)$ is feasible. The complexity of PSEARCH is $O(n + t(n) \log n)$.* □

For some of the algorithms we describe later, *PSEARCH* will be initiated with $|S| > 1$ (i.e., S will contain more than one M matrix initially; all matrices in S will still be of the same size). To analyze the complexity of these algorithms, we shall use the following theorem and corollary.

Theorem 2 : *[FRED92] If PSEARCH is initiated with S containing m sorted matrices, each of dimension n, then the number of feasibility tests is $O(\log n)$ and the total time spent obtaining the candidate values for these tests is $O(mn)$.* □

Corollary 2 : *Let $t(n)$ be as in Corollary 1. The complexity of PSEARCH under the assumptions of Theorem 4 is $O(mn + t(n) \log n)$.* □

While we have described *PSEARCH* under the assumption that the matrices of candidate values are square and of dimension a power of 2, parametric search easily handles other matrix shapes and sizes. For this, we can add more rows at the top and columns to the left so that the matrices become square and have a dimension that is a power of two. The

entries in the new rows and columns are 0. This does not affect the asymptotic complexity of *PSEARCH*. Alternatively, we can modify the matrix splitting process to partition into four roughly equal submatrices at each step. The details of these generalizations are given in [FRED83, FRED84, FRED91, FRED92].

Procedure *PSEARCH* is a restricted version of procedure *MSEARCH* of [FRED92]. An alternative search algorithm in which the **for** loop is iterated twice, once with T being the multiset of the largest values in S and once with T being the multiset of the smallest values in S is given in [FRED83, FRED84].

3 Standard Cell Folding (Problems 1–4)

Our discussion of problems 1–4 is divided into two parts. In Section 3.1, we consider problems 1 and 2. In both these, the chip width and hence the cell and routing channel widths are fixed at W. In Section 3.2, we consider problems 3 and 4 in both of which the chip height is fixed at H. In all four problems, the routing channels have variable height. Each cell and hence each cell row has height h. The width of cell i is w_i, $1 \leq i \leq n$. Let $w_{ij} = \sum_{k=i}^{j} w_k, 1 \leq i \leq j \leq n$. In case of fixed chip width W, we may assume that $w_i \leq W$, $1 \leq i \leq n$.

3.1 Width Constrained Case (Problems 1 and 2)

We first consider problem 1. In this, we are to minimize the total routing area. Since the channel widths are fixed at W, it is sufficient to minimize the sum of channel heights. Suppose that $C_1, \cdots, C_n$ is folded at C_i in an optimal folding X. Then the folding of $C_1, \cdots, C_i$ in X as well as that of $C_{i+1}, \cdots, C_n$ must be minimum area foldings. Hence, the principle of optimality holds and we can use dynamic programming [HORO78].

Let $f(i,s)$, $i \leq s$, denote the minimum sum of channel heights when the component list $C_i, \cdots, C_n$ is folded such that $C_i, \cdots, C_s$ are in one cell row and the first fold is at C_s (so, C_{s+1} is in the next cell row). It is easy to see that $f(n,n) = l_n = 0$. For $1 \leq i < s \leq n$, we get

$$f(i,s) = \begin{cases} \infty & \text{if } w_{is} > W \\ f(i+1,s) & \text{otherwise} \end{cases} \qquad (1)$$

Also, for $1 \leq i = s \leq n$, we get

$$f(i,i) = \min_{i<q\leq n} \{f(i+1,q) + l_i\} \tag{2}$$

The solution to problem 1 is obtained by first using Equations 1 and 2 to determine $f(i,s)$, $1 \leq i \leq s \leq n$ and then determining the minimum of $f(1,j)$, $1 \leq j \leq n$. The w_{is}'s may be precomputed in $O(n^2)$ time. Each $f(i,s)$, $i < s$ takes $O(1)$ time to compute and $f(i,i)$ takes $O(n-i)$ time. Hence, all the $f(i,s)$'s, $i \leq s$ may be obtained in $O(n^2)$ time. The minimum of the $f(1,j)$'s can be obtained in $O(n)$ time. So the overall time needed to solve problem 1 using Equations 1 and 2 is $O(n^2)$.

A more careful implementation of the dynamic programming algorithm results in a complexity $O(n)$. First we compute the suffix sums

$$Q_i = \sum_{j=i}^{n} w_j, 1 \leq i \leq n$$

in $O(n)$ time. Let $Q_{n+1} = 0$. From the suffix sums, each w_{is} can be computed in $O(1)$ time using

$$w_{is} = Q_i - Q_{s+1}$$

Next, from Equation 1 we see that for $i < s$ and $w_{is} \leq W$:

$$f(i,s) = f(i+1,s) = f(i+2,s) = \cdots = f(s,s) = F(s)$$

So, Equation 1 becomes (for $i < s$)

$$f(i,s) = \begin{cases} \infty & w_{is} > W \\ F(s) & \text{otherwise} \end{cases} \tag{3}$$

Using Equation 3, Equation 2 may be rewritten as:

$$\begin{aligned} F(i) = f(i,i) &= \min_{i<q\leq n} \{f(i+1,q) + l_i\} \\ &= \min_{i<q\leq n \text{ and } w_{i+1,q}\leq W} \{F(q) + l_i\} \\ &= l_i + \min_{i<q\leq n \text{ and } w_{i+1,q}\leq W} \{F(q)\} \end{aligned} \tag{4}$$

The minimum total routing height needed is

$$\min_{1\leq i\leq n \text{ and } w_{1i}\leq W} \{F(i)\} \tag{5}$$

So, problem 1 may be solved by computing the n $F(i)$'s using Equation 4 (rather than the $O(n^2)$ $f(i,s)$'s using Equations 1 and 2) and finding the minimum of $O(n)$ $F(i)$'s in Equation 5. To compute the $F(i)$'s using Equation 4, we begin with $F(n) = 0$ and compute $F(n-1), F(n-2), \cdots, F(1)$, in that order. To compute an $F(i)$ we need to find the minimum of a multiset S_i of previously computed F's. Specifically,

$$S_i = \{F(q) \mid 1 < q \leq n \text{ and } \ w_{i+1,q} \leq W\}$$

Observation 1 : *If $w_{j+1,q} > W$, then $w_{i+1,q} > W$ for $i \leq j$. Hence, if $F(q) \notin S_j$, then $F(q) \notin S_i$ for $i \leq j$.* $\square$

From Observation 1, it follows that S_{i-1} may be computed from S_i, $1 < i \leq n$ by eliminating those $F(q)$'s for which $w_{i,q} > W$ and adding in $F(i)$ (note that, by assumption, $w_i = w_{ii} \leq W$).

Lemma 1 : *If $F(a) \in S_i$, $F(b) \in S_i$, $F(a) \leq F(b)$, $a < b$, then we may eliminate $F(b)$ from S_i and continue to compute $S_{i-1}, S_{i-2}, \cdots, S_1$ as described above. This does not affect the values of $F(i-1), \cdots, F(1)$.*

Proof : Note that $F(j)$, $j < i$ is being computed using the equation

$$F(j) = l_j + \min_{F(q) \in S_j} \{ F(q) \}$$

If $F(b)$ is eliminated from S_i, the value of $F(i)$ is unaffected as $F(a) \leq F(b)$. If $F(a)$ is eliminated from S_j, $j < i$ because $w_{j+1,a} > W$, then $F(b)$ would also be eliminated as $a < b$ and so $w_{j+1,b} > w_{j+1,a} > W$. If $F(a)$ is eliminated because there is a $F(c) \leq F(a)$, $c \leq a$, then so also will be $F(b)$ be eliminated as $F(c) \leq F(a) \leq F(b)$ and $c < a < b$. $\square$

Observation 1 and Lemma 1 motivate us to maintain S as a sequential queue [HORO94] in an array $Result[1..n]$. $Result[i].q$ and $Result[i].F$ together represent an entry of S yielding the value $F(q)$. The elements of S are stored in positions $tail, tail+1, \cdots, head$ of array $Result$. The $F(q)$ values are in descending order left-to-right. Hence, the q values are in ascending order. Procedure *MinimizeHtStandard* (Figure 3) is the resulting algorithm.

Theorem 3 : *The procedure MinimizeHtStandard given in Figure 3 is correct.*

```
Procedure MinimizeHtStandard;
{ Compute the minimum height layout}
      { Initialize S_n = {F(n) = 0 } }
      head := n; tail := n;
      Result[tail].F := 0; Result[tail].q = n;
     { Compute F(i) }
      for i := n − 1 downto 1 do
      begin
         { Compute S_i }
         while (Q[i + 1] − Q[Result[head].q + 1] > W) do
               head := head − 1; {delete from S_{i+1}, Observation 1 }
           temp := l[i] + Result[head].F; {Use min F in S_i to compute F(i)}
         while (temp ≤ Result[tail].F) do  { delete using Lemma 1 }
               tail := tail + 1;
           { Store F(i) }
           tail := tail − 1;
           Result[tail].F := temp;
           Result[tail].q := i;
      end; { of for }
     while (Q[1] − Q[Result[head].q + 1] > W) do
           head := head − 1;
     MinimizeHtStandard := Result[head].F;
  end; { of MinimizeHtStandard }
```

Figure 3: Procedure to obtain a minimum height folding

Proof : There are two parts to the working of procedure *MinimizeHtStandard*. The first one is computing $F(i)$, in which deletions of $F(.)$'s can occur. The second one is inserting the computed $F(i)$ at the appropriate place in the array.

The procedure maintains the following invariant at the start of each iteration of the **for** loop.

Invariant: $Result[tail].F > Result[tail+1].F > \cdots > Result[head].F$

It is clearly true when $i = n-1$ as $head = tail$.

The invariant is true at the start of the iteration and so $Result[head].F$ is the minimum maintained $F(.)$ value. The component number is maintained in $Result[head].q$. We check whether $Q[i+1] - Q[Result[head].q+1] > W$ and if so by virtue of Observation 1, we can eliminate this value. We do so by decrementing the *head* pointer. We keep repeating this until we find a record $k = Result[head].q$ such that $BR[i+1] - BR[k+1] \leq W$. This record pointed to by *head* has the minimum of the maintained $F(.)$ values. We compute $F(i)$ and store it in *temp*. Notice that at the end of the **while** loop, we have deleted a few $F(.)$'s and the invariant property still holds.

The invariant holds before the start of the second **while** loop. Here we start at *tail*. If the inequality is true, then we delete the record and this is justified by Lemma 1. We keep doing so until $temp > Result[tail].F$. Then we decrement the tail pointer and store the *temp* record. So, the invariant holds at the end of the iteration. Consequently, the invariant holds at the start of each iteration of the **for** loop and the F's are correctly computed.

The minimum height layout is the minimum of the maintained $F(.)$ values that satisfy the width constraint, i.e $Q[1] - Q[Result[head].q+1] \leq W$. The last **while** loop of the procedure take care of this fact. The last line of procedure computes $MinimizeHtStandard$, which is the minimum height layout. □

Whenever the pointers *head* or *tail* are advanced in the **while** loops, we delete $F(.)$ values. This cost can be charged towards deletion of $F(.)$ values. The remaining code within the **for** loop takes $O(n)$ amortized time. The complexity of the procedure *MinimizeHtStandard* is clearly $O(n)$ as no more than n deletions can take place. Using standard dynamic programming traceback techniques [HORO78], the fold points can be obtained in additional $O(n)$ time.

Problem 2, i.e, minimize total area rather than just routing area may be done in a

similar way. Let $f(i,s), i \leq s$ now denote the minimum chip height for the component list $C_i, \cdots, C_n$ assuming the first fold is at s. As before $f(n,n) = 0$ and Equation 1 holds for $i < s$. Equation 2 needs to be replaced by

$$f(i,i) = \min_{i<q\leq n} \{f(i+1,q) + l_i + h\} \tag{6}$$

Using Equations 1 and 6 and the development for problem 1, an O(n) time algorithm for problem 2 may be obtained.

3.2 Height Constrained Case (Problems 3–4)

The solutions to problems 3 and 4 are similar. Both use parametric search and we describe only the solution to problem 3. Since the total height of the routing channels is fixed at H, the area assigned for routing is minimized by minimizing the chip width W. To use parametric search to minimize W, we must do the following:

1. Identify a set of candidate values for the minimum W. This set must be provided as a sorted matrix with the property that each matrix entry can be computed in constant time.

2. Provide a way to determine if a candidate width W is feasible, i.e, can the component stack can be folded using total channel height H and width W ?

For the feasibility test of 2, we can use procedure *MinimizeHtStandard* of Figure 3 by setting W to the candidate value being tested and then determine if $MinimumHtStandard \leq H$ following the execution of the procedure.

Next, we provide an $n \times n$ sorted matrix M (n is the total number of components in the component list) of candidate values. To determine the candidate matrix M, we observe that the width of any layout is given by $\sum_{q=i}^{j} w_i$ for some $i, j, 1 \leq i \leq j \leq n$. This formula gives us the width of the segment that contains components C_i throught C_j. M is a sorted matrix that contains all candidate values. The minimum M_{ij} for which a height H folding is possible is the minimum width height-H folding. We now show how the elements of M may be computed efficiently given the index pair (i,j). Let

$$T_i = \sum_{j=i}^{n} w_i, 1 \leq i \leq n$$

and let $T_{n+1} = 0$. Then,

$$M_{ij} = \begin{cases} T_{n-i+1} - T_{j+1}, i+j \geq n+1 \\ 0, i+j < n+1 \end{cases}$$

So, if we precompute the T_i's each M_{ij} can be determined in constant time. The precomputation of the T_i's takes $O(n)$ time. Since feasibility testing takes linear time, from Corollary 1, it follows that the complexity of the described parametric search to find the minimum width folding is $O(n + t(n) \log n) = O(n + n \log n) = O(n \log n)$.

4 Standard Cell Folding (Problems 5–7)

In this section, we deal with layouts which have fixed channel area, e.g, semicustom chips in which each routing channel is of the same height.

4.1 Minimum Channel Height (Problem 5)

We may view the result of any width W folding as the transformation of the component list $C_1, \cdots, C_n$ into a new component list $B_1, \cdots, B_k$, $k \leq n$ where B_i represents the components folded into row i of the layout. The width of each B_i equals the sum of the widths of the components assigned to cell row i and this is $\leq W$. Also, the routing channel between rows i and $i+1$ must have height at least equal to l_{j_i} where C_{j_i} is the last component assigned to cell row i. We see that

$$width(B_i) = \sum_{j=j_{i-1}+1}^{i} w_j$$

$$\text{and height of channel}(i) \geq l_{j_i}$$

where $j_0 = 0$. When channel heights are the same, the height must be at least $\max_{1 \leq i < m}\{l_{j_i}\}$.

With this knowledge, we can develop a greedy algorithm to minimize channel height. In this, we repeatedly combine together pairs of components (this is equivalent to assigning them to the same cell row or B_i) so that no created component has width greater than W. The pairs are chosen in non-increasing order of l_i. The greedy algorithm is given in Figure 4. Each set of combined components is represented by a pointer, *last*, from the first

```
Procedure MinChannelHeight;
    for i := 1 to n do {intialize component blocks }
    begin
       first[i] := i; last[i] := i;
    end;
    Sort p[1..n] = [1, 2, ..., n] so that
     l[p[i]] ≥ l[p[i + 1]], 1 ≤ i < n
    i := 1;
    while( width[first[p[i]]] + width[p[i] + 1] ≤ W ) do
    begin
       width[first[p[i]]] := width[first[p[i]]] + width[p[i] + 1];
       first[last[p[i] + 1]] := first[p[i]];
       last[first[p[i]]] := last[p[i] + 1];
       i := i + 1;
    end;
    MinChannelHeight:= l[p[i]];
end;
```

Figure 4: Procedure to obtain a minimum channel height folding

component to the last and another pointer, *first*, from the last component to the first. The width of the combined component is kept in the first elementary component of the combined component.

In the algorithm of Figure 4, we initialize the combined component blocks to consist of elementary components in the first **for** loop. The sort gives us the order in which the l's are to be "eliminated" so that the maximum of the remaining l's is the minimum. In the **while** loop l's are eliminated by combining blocks. This is done until the next highest l (we assume that $\sum w_i > W$ so it is not possible to eliminate all l's). The highest remaining l is $l[p[i]]$ and this is the smallest channel height needed.

The correctness of the procedure is easily established. For its complexity, we see that except for the sort step, the others take $O(n)$ time. The sort can be done in $O(n \log n)$ time. However, in practice, $\max\{l_i\} - \min\{l_i\} = O(n)$ and the sort can be done in $O(n)$ time using a radix sort with radix $O(n)$ (i.e., a bin sort) [HORO94]. One may also verify

that the minimum number of cell rows needed is obtained by doing a greedy folding on the combined components that remain when procedure *MinChannelHeight* terminates.

4.2 Minimize Chip Area Subject To Width Constraint (Problem 6)

First, consider a modified version of problem 6 in which in addition to the chip width W, we are given the height L of each routing channel. We are to fold the components so as to minimize the total chip area. To solve modified problem 6 in linear time, we first make a pass over all the components and combine components C_i and C_{i+1} if $l_i > L$. If any component that results has width $> W$, L is an infeasible channel height. Following the combining of blocks in this way, the resulting blocks are packed into cell rows in a greedy manner (i.e., a new cell row is started only if the component being placed does not fit in the current cell row). The fact that this minimizes the number of cell rows and hence chip area is easily verified.

Problem 6 can be solved using the solution to modified problem 6 by trying out all $O(n)$ possible values for L (i.e., the distinct l_i's) and seeing which minimizes overall area. (Actually only l_i's that are no less than the minimum feasible L as determined by problem 5 need be tried). The resulting complexity is $O(n^2)$.

4.3 Minimize Chip Area Subject To Height Constraint (Problem 7)

As for problem 6, we define a modified problem 7 in which the channel height L is known. This modified problem is solved using parametric search. The candidate values are described by the same M matrix as used in Section 3.2. The solution to modified problem 6 is used for the feasibility test. This enables us to solve the modified version of problem 7 in $O(n \log n)$ time. Now, by trying out all $O(n)$ possible L values (as in Section 4.2) the minimum area folding can be determined. The overall time complexity is $O(n^2 \log n)$.

5 Custom Cell Folding (Problems 8 and 9)

In this section, we relax the requirement that all components have the same height h. Let h_i be the height of C_i. If $C_i, \cdots, C_j$ are assigned to the same cell row and no other components

Procedure *MinimizeHtCustom*;
{ Compute the minimum height folding}
$head := n; tail := n; left := n; right := n;$
for $i := 1$ **to** n **do**
$Hlist[i].gvalue := \infty;$
$Flist[tail].q := n; Flist[tail].F = 0;$
$Hlist[n].top := tail; Hlist[n].bottom := tail;$
$Hlist[n].hvalue := h[n];$
$Hlist[n].gvalue := Hlist[n].hvalue + Flist[Hlist[n].top].F;$
InitializeWinnerTree(T);
for $i := n - 1$ **downto** 1 **do**
begin
DeleteValue(i);
InsertValue(i);
end;{of **for** }
DeleteValue(0);
$MinimizeHtCustom :=$ Winner of the Tree T;
end; { of *MinimizeHtCustom* }

Figure 5: Procedure to obtain a minimum height folding for custom cells

are assigned to this row, then the cell row height is

$$\max_{i \le q \le j} \{h_q\}$$

The height of the folding is the sum of the heights of the cell rows and routing channels.

5.1 Width Constrained Folding (Problem 8)

Since the chip width is fixed at W, chip area is minimized by minimizing chip height. Let $R_{ij} = \max_{i \le q \le j}\{h_q\}$, $1 \le i \le j \le n$. Let $f(i,s)$, $i \le s$ be the minimum height into which $C_i, \cdots, C_n$ can be folded such that the first fold is at C_s. Following the development of Section 3.1, we see that $f(n,n) = h_n$, and for $i < s$,

$$f(i,s) = \begin{cases} \infty & \text{if } w_{is} > W \\ f(s,s) + R_{is} - h_s, & \text{otherwise} \end{cases} \tag{7}$$

and for $i = s$,

$$f(i,i) = \min_{i<q\leq n} \{ f(i+1,q) + h_i + l_i\} \tag{8}$$

The minimum height into which the folding can be done is $\min_{1\leq q\leq n} \{f(1,q)\}$. As described in Section 3.1, the set of dynamic programming equations can be solved in $O(n^2)$ time. However, the development of Section 3.1, that results in an $O(n)$ time solution does not apply to the new set of equations. Instead, we are able to solve problem 8 in $O(n \log n)$ time.

Define $F(i) = f(i,i) - h_i$. Substituting into Equation 7, we get

$$f(i,s) = \begin{cases} \infty & \text{if } w_{is} > W \\ F(s) + R_{is}, & \text{otherwise} \end{cases} \tag{9}$$

From Equation 8, we get

$$\begin{aligned} F(i) = f(i,i) - h_i &= \min_{i<q\leq n} \{f(i+1,q)\} + l_i \\ &= l_i + \min\{f(i+1,i+1), \min_{i+1<q\leq n} \{f(i+1,q)\}\} \\ &= l_i + \min\{F(i+1) + h_{i+1}, \min_{i+1<q\leq n, w_{i+1,q}\leq W} \{F(q) + R_{i+1,q}\}\} \\ &= l_i + \min_{i<q\leq n, w_{i+1,q}\leq W} \{F(q) + R_{i+1,q}\} \end{aligned} \tag{10}$$

The height of the minimum height folding is

$$\min_{1\leq q\leq n,\ w_{1i}\leq W} \{F(i) + R_{1i}\} \tag{11}$$

Beginning with $F(n) = f(n,n) - h_n = 0$, the remaining F's may be computed, in the order $F(n-1), \cdots, F(1)$, by using Equation 10. To use Equation 10, we keep a multiset S_i of F values as in Section 3.1. We begin with $S_n = \{F(n)\}$ and rewrite Equation 10 as :

$$F(i) = l_i + \min_{F(q)\in S_i} \{F(q) + R_{i+1,q}\} \tag{12}$$

Observation 1 of Section 3.1 applies to Equation 12 and we may eliminate from S_i any $F(q)$ for which $w_{i+1,q} > W$.

Observation 2 : $R_{i,q} \geq R_{i,q-1} \geq \cdots \geq R_{i,i}$, $1 \leq i \leq q \leq n$.

Using Observation 2 and Equation 12 we can show that Lemma 1 applies for the computation of the F's as defined in this section.

```
Procedure DeleteValue(i);
{ Delete F(l) such that w_{i+1,l} > W }
    done:=false; bool:= false;
    while (not done) do
      if (Q[i+1] - Q[Flist[Hlist[right].top].q + 1] > W) then
      {Delete this F(.) value}
          Hlist[right].top = Hlist[right].top - 1;
          head = head - 1;
          bool := true;
          if (Hlist[right].top < Hlist[right].bottom) then
              { Make this record inactive }
              Hlist[right].gvalue := ∞;
              AdjustWinnerTree(T, right);
              right := right - 1; bool := false;
          end;{of if}
      else done := true;
      end;{of if}
    end;{of while}
    if bool then
        Hlist[right].gvalue := Hlist[right].hvalue + Flist[Hlist[right].top].F;
        AdjustWinnerTree(T, right);
    end;{of if}
  end; { of DeleteValue }
```

Figure 6: Procedure to delete $F(.)$ values as in Observation 3

Observation 3 : *If $h_j > h_q$ and $i \leq j \leq q$, then $R_{iq} \neq h_q$. Also, if $h_j \geq h_{j+1}$ and $i \leq j$ then $R_{ij} = R_{i,j+1}$.*

Now, we devise a method to find the minimum in Equation 12 efficiently. We store the $F(.)$ values in an array of records called $Flist$. Each $Flist$ record has two fields, $Flist.q$ and $Flist.F$. $Flist.F = F(Flist.q)$, ie, say $F(8) = 50$ then there is a record which has $Flist.q = 8$ and $Flist.F = 50$. There are two pointers, *head* and *tail* that are used. Initially, $head = tail = n$. At any point, the head and tail have values such that $head \geq tail$ and $F(tail) > F(tail + 1) > \cdots > F(head)$. This data structure is same as the one used in Section 3.1.

When computing $F(i)$ we need to associate $F(q)$ values with $R_{i+1,q}$ values and then generate values $F(q) + R_{i+1,q}$, and find the minimum of these values. Suppose $h_q > h_{q+1}$

then $R_{iq} = R_{i,q+1}$ from Observation 3. Associate the values $F(q)$ and $F(q+1)$ with $R_{i,q}$ in this case. In general, if $R_{i,q} = R_{i,q+1} = R_{i,q+2} = \cdots = R_{i,l}$, then we have a single $Hlist$ record with h_q value and associated with it the values $F(q), F(q+1), \cdots, F(l)$. Note that the $F(.)$ values must satisfy the condition : $F(q) > F(q+1) \cdots > F(l)$. Otherwise the $F(.)$ values which violate the condition can be removed as in Lemma 1 by doing a left to right scan. We use an array of records $Hlist$ of size n with fields $Hlist.hvalue$ representing the height, $Hlist.top$ and $Hlist.bottom$ the two pointers which keep track of the $F(.)$ values associated with this record. The *top* and *bottom* pointers point to the $F(.)$ values satisfying the condition: $Flist[Hlist.top].F < Flist[Hlist.top-1].F < \cdots < Flist[Hlist.bottom].F$. I.e, $Flist[Hlist.top].F$ is the minimum $F(.)$ value associated with this record. Note that every $F(.)$ value is associated with a unique $Hlist$ record. We generate the value $Flist[Hlist.top].F + Hlist.hvalue$ (which is $F(q) + R_{i,q}$) and store it in $Hlist.gvalue$ (generated value). These generated values are used to construct a winner tree T (see [HORO94]).

The winner of the tree T is the minimum we are looking for when computing $F(i)$. Let a $Hlist$ record be active if $Hlist.gvalue \neq \infty$. The pointers $left$ and $right$, $left \leq right$, are used to point to the currently active list of $Hlist$ records. $Hlist[left]$ is the leftmost active record and $Hlist[right]$ is the rightmost active record.

The procedure *MinimizeHtCustom* is given in Figure 5. The pointers are initialized and the winner tree T initialized. In the procedure *DeleteValue(i)*, the $F(l)$ values that satisfy the conditions in Observation 1, i.e., $F(l)$ values such that $w_{i+1,l} > W$ are deleted. Let $Q[j] = w_j + w_{j+1} + \cdots + w_n$.

The procedure *DeleteValue* is given in Figure 6. The boolean *bool* keeps track of whether a $Hlist$ record has been made inactive. If so, it moves the pointer *right* to left to point to an active $Hlist$ record. Also, the winner tree T is adjusted to update the current minimum. The call to function *AdjustWinnerTree* takes $O(\log n)$ time [HORO94]. Note that the winner tree T is adjusted a maximum of two times whenever an $F(.)$ value is deleted. Let the number of deletes when *DeleteValue* is invoked be x. Then, the time complexity of *DeleteValue* is $O(x \log n)$.

The procedure *InsertValue(i)* first finds the winner of the tree T. This is added with $l[i]$ to get $F(i)$ as in Equation 12. Once we find $F(i)$, we then insert a $Hlist$ record with $Hlist.hvalue = h[i]$ and the $F(.)$ value is inserted in the array of $Flist$ records. The winner

```
Procedure InsertValue(i);
    left := left − 1; tail := tail − 1;
    Flist[tail].q := i;
    Flist[tail].F := Winner of the Min Tree T + l[i];
    Hlist[left].hvalue := h[i];
    Hlist[left].top := tail; Hlist[left].bottom := tail;
    Hlist[left].gvalue := Hlist[left].hvalue + Flist[Hlist[left].top].F;
    AdjustWinnerTree(T, left);
    while (head ≠ tail and Flist[tail].F ≤ Flist[tail + 1].F) do
        Hlist[left + 1].bottom := Hlist[left + 1].bottom + 1;
        if (Hlist[left + 1].bottom > Hlist[left + 1].top) then
            Hlist[left + 1] := Hlist[left]; { Move the record }
            Hlist[left].gvalue := ∞;
            AdjustWinnerTree(T, left);
            left := left + 1;
        end;{of if}
        Flist[tail + 1] = Flist[tail]; { Move the record }
        tail := tail + 1;
        Hlist[left].top := tail; Hlist[left].bottom := tail;
    end;{of while}
    while (left ≠ right and Hlist[left].hvalue ≥ Hlist[left + 1].hvalue) do
    {Conditions of Observation 3 apply}
        Hlist[left].top := Hlist[left + 1].top;
        Hlist[left + 1] := Hlist[left]; { Move the record }
        Hlist[left].gvalue := ∞;
        AdjustWinnerTree(T, left);
        left := left + 1;
        Hlist[left].gvalue := Hlist[left].hvalue + Flist[Hlist[left].top].F;
        AdjustWinnerTree(T, left);
    end;{of while}
end; { of InsertValue }
```

Figure 7: Procedure to Insert $F(.)$ values

tree T is then adjusted. In the first **while** loop of the *InsertValue*, conditions of Lemma 1 are checked. If the conditions apply then the $F(.)$ values are deleted and the winner tree adjusted. Let the number of $F(.)$ value deletions be y. In the second **while** loop of the *InsertValue*, it is checked to see whether the conditions of Observation 3 apply. If so, the $F(.)$ records of the adjacent $Hlist$ record is added to the current $Hlist$ record and the record moved. The winner tree is then adjusted. Every time, the conditions of Observation 3 apply in the **while** loop, we spend $O(\log n)$ time. I.e., every time the conditions apply we merge two adjacent $Hlist$ records. Let the number of merges in a single invocation of *InsertValue* be z. The total time taken by a single invocation of *InsertValue*, assuming y $F(.)$ values are deleted and z $Hlist$ merges take place is $O((y+z+1)\log n)$ time.

Note that not more than n $F(.)$ values can be deleted in total, and not more than n $Hlist$ records can be merged in total. This implies that the total time taken by the procedure *MinimizeHtCustom* is $O(n \log n)$. In contrast, the algorithm of [PAIK93], for the same problem takes $O(n^2)$ time.

5.2 Height Constrained Folding (Problem 9)

To obtain the minimum height folding, given the width of the folding W, we use parametric search in conjunction with the procedure *MinimizeHtCustom* developed in Section 5.1. The procedure *MinimizeHtCustom* is used for the feasibility testing. In feasibility testing, we are given the width, x, of the layout and we test whether it is possible to obtain a folding such that the height of the folding is $\leq H$. The set of candidate values is the same as the ones described in Section 3.2. The feasibility testing takes $O(n \log n)$ time, and from Corollary 1, the total time taken to obtain the minimum height folding is $O(n+n \log n * \log n)$ = $O(n \log^2 n)$. The same problem is solved in $O(n^2 \log n)$ time in [PAIK93].

6 Experimental Results

The procedure *MinimizeHtStandard* (Figure 3) was programmed in C and run on a SUN 4 workstation. The solution produced by *MinimizeHtStandard* was compared with the one obtained using the greedy heuristic of [SHRA88]. The data for these programs were produced by having a linearly ordered list of modules and making interconnections between the modules using a random number generator. The connections were prioritized

n	Greedy	Ours
100	624.4	609.8
400	1979	1961.2
1000	3813.7	3721.2

Table 1: Heights produced by width-constrained standard cell folding algorithms

n	Ours	[PAIK93]
64	2.56	23.80
250	10.9	350.3
1000	39.23	6125.5

Times are in milliseconds

Table 2: Run times of width-constrained folding algorithms for custom cells

so that there is a large number of connections between modules which are close together. Our algorithm always produces better solutions than the greedy heuristic and the results are depicted in Table 1. The results shown are the average of 10 runs for each n. Our algorithm, on the average, took 2 to 3 times more time to arrive at the solution than taken by the greedy heurisitic.

The algorithm *MinimizeHtCustom* was programmed and the run times compared with the algorithm of [PAIK93]. The results of the experiments are shown in Table 2. Both the programs were written in C. It is evident that our algorithm is considerably superior to that of [PAIK93]. Since both algorithms generate optimal solutions, the chip area is the same using either.

7 Conclusions

We have developed optimal algorithms to fold a linearly ordered list of standard and custom cells. Several optimization constraints were considered. These resulted in a total of nine problem formulations. Two of these correspond to problem formulations for the bit-slice stack folding problem studied in [PAIK93]. The algorithms we have developed for these two cases are asymptotically superior to those developed in [PAIK93]. Experimentation with

one of these shows that the asymptotic superiority of our algorithms translates into a much reduced execution time. For the other formulations, heuristics were proposed in [SHRA90]. Our algorithms have acceptible asymptotic complexity and guarantee optimal solutions. In fact, experiments conducted with one yielded foldings with smaller chip area on all tested instances.

References

[BREU79] Breuer, M. A., "Min-Cut Placement", Journal of Design Automation and Fault Tolerant Computing, 1:4, pp:346-362, 1977.

[COX80] Cox, G. W. and B. D. Carroll, "The Standard Transistor Array (STAR) Part II Automatic Cell Placement Techniques", 18th Design Automation Conference, pp. 451-457, 1980.

[FRED83] G. N. Frederickson, and D. B. Johnson, "Finding kth paths and p-centers by generating and searching good data structures", *Journal of Algorithms*, 4:61-80, 1983.

[FRED84] G. N. Frederickson, and D. B. Johnson, "Generalized selection and ranking: sorted matrices", *SIAM Journal on computing*, 13:14-30, 1984.

[FRED91] G. N. Frederickson, "Optimal algorithms for tree partitioning", *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California (Jan. 1991), pp. 168-177

[FRED92] G. N. Frederickson, "Optimal parametric search algorithms in trees I: tree partitioning", Purdue University, Technical Report CSD-TR-1029, 1992.

[HEL77] Heller, W. R., W. F. Mikhail, and W. E. Donath, " Prediction of Wiring Space Requirements for LSI", 14th Design Automation Conference, pp. 32-42, 1977.

[HORO78] E. Horowitz, and S. Sahni, "Fundamentals of Computer Algorithms", Computer Science Press, Maryland, 1978.

[HORO94] E. Horowitz, and S. Sahni, "Fundamentals of Data Structures in Pascal", Computer Science Press, New York, 1994.

[KANG83] Kang, S., "Linear Ordering and Application to Placement", 20th Design Automation Conference, pp. 457-464, 1983.

[PAIK93] D. Paik, S. Sahni, "Optimal folding of bit sliced stacks", *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol 12(11), Nov. 1993, 1679-1685.

[SHRA88] E. Shragowitz, L. Lin, S. Sahni, "Models and algorithms for structured layout", *Computer Aided Design*, Butterworth & Co, London, 20, 5, 1988, 263-271.

[SHRA90] E. Shragowitz, J. Lee, and S. Sahni, "Placer-router for sea-of-gates design style", in *Progress in Computer Aided VLSI Design*, Ed. G.Zobrist, Ablex Publishing, Vol 2, 1990, 43-92.