Folding A Stack Of Equal Width Components *

Venkat Thanvantri

Department of CIS
University of Florida
Gainesville, FL-32611

Sartaj Sahni

Department of CIS
University of Florida
Gainesville, FL-32611

Abstract

We consider two versions of the problem of folding a stack of equal width components. In both versions, when a stack is folded, a routing penalty is incurred at the fold. In one version, the height of the folded layout is given and we are to minimize width. In the other, the width of the folded layout is given and its height is to be minimized.

1 Introduction

Component stack folding, in the context of bit sliced architectures, was introduced by Larmore, Gajski, and Wu [6]. In this paper, they used this model to compile layout for cmos technology. Further applications of the model were considered by Wu and Gajski [11]. In the model of [6] and [11] the component stack can be folded at only one point. In addition, it is possible to reorder the components on the stack. A related, yet different, folding model was considered by Paik and Sahni [7]. In this, no limit is placed on the number of points at which the stack may be folded. Also, component reordering is forbidden. They point out that this model may also be used in the application cited by [6] and [11]. Furthermore, it accurately models the placement step of the standard cell and sea-of-gates layout algorithms of Shragowitz et al. [8, 9].

Formally, a component stack is comprised of variable height and variable width components $C_1, C_2, \ldots, C_n$ stacked one on top of the other. C_1 is at the top of the stack and C_n at the bottom. If the component stack is folded at C_i we obtain two adjacent stacks $C_1, C_2, \ldots, C_i$ and $C_n, C_{n-1}, \ldots, C_{i+1}$. The folding also inverts the left to right orientation of the components $C_n, \ldots, C_{i+1}$. Notice that folding results in a snake-like rearrangement. Each fold flips the left-to-right orientation of a component. Pairs of folded stacks may have nested components, components in odd stacks are left aligned; and components in even stacks are right aligned. The area of the folded stack is the area of the smallest rectangle that bounds the layout. To determine this, depending on the model, we may need to add additional space at the stack ends to allow for routing between components C_i and C_{i+1} where C_i is a folding point. If so, let $r_i \geq 0$, $2 \leq i \leq n$, denote the height of the routing space needed if the stack is folded at C_{i-1}.

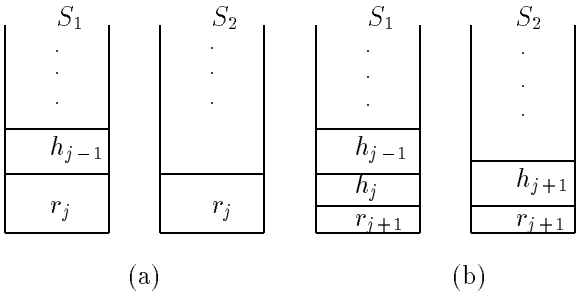

Figure 1: Case when $h_j + r_{j+1} \leq r_j$

In this paper we consider two of the problems considered in [7]:

(1) *Equal-width, height-constrained with routing area at stack ends.*

(2) *Equal-width, width-constrained with routing area at stack ends.*

Our algorithms employ two techniques. The first is normalization in which an input instance is transformed into an equivalent normalized instance that is relatively easy to solve. The second technique is *parameterized searching.*

2 Normalization

Let h_i be the height of the component C_i, $1 \leq i \leq n$. Let r_i be the routing height needed between C_{i-1} and C_i if the component stack is folded at C_{i-1}, $2 \leq i \leq n$; and let $r_1 = r_{n+1} = 0$. The defined component stack is *normalized iff* the conditions C1 and C2 given below are satisfied for every i, $1 \leq i \leq n$.

C1 : $h_i + r_{i+1} > r_i$

C2 : $h_i + r_i > r_{i+1}$

An unnormalized instance I may be transformed into a normalized instance $\hat{I}$ with the property that from a minimum height or minimum width folding of $\hat{I}$, one can easily construct a similar folding for I. To

*This research was supported in part by the National Science Foundation under the grant MIP 91-03379

```
Procedure Normalize(C,n)
{ Normalize the component stack C[1]...C[n]}
   i := 1; next := 2;
   while next ≤ n + 1 do
     case
       : C[i].h + C[next].r ≤ C[i].r :
           {Combine with C[i − 1]}
           C[i − 1].h := C[i − 1].h + C[i].h;
           C[i − 1].l := C[i].l;
           i := i − 1;
       : C[i].h + C[i].r ≤ C[next].r :
           {Combine with C[next]}
           C[i].h := C[i].h + C[next].h;
           C[i].l := C[next].l;
           next := next + 1;
       :else: C[i + 1] := C[next];
           i := i + 1; next := next + 1;
   end;
   n := i − 1;
end; {Normalize}
```

Figure 2: Normalizing a stack

obtain $\hat{I}$, we identify the least value of i at which either C1 or C2 is violated. Let this value of i be j. By choice of j, either

$$h_j + r_{j+1} \leq r_j, \text{ or}$$
$$h_j + r_j \leq r_{j+1}.$$

We note that it is not possible for both of these inequalities to hold simultaneously. Suppose that $h_j + r_{j+1} \leq r_j$. Now $j > 1$ as $h_1 + r_2 > 0$ while $r_1 = 0$. Also, $h_j + r_j > r_{j+1}$. Consider any folding of I in which C_{j-1} is a fold point (Figure 1(a)). Let the height of the stack S_1 be $h(S_1)$ and that of $S_2, h(S_2)$. Consider the folding obtained from Figure 1(a) by moving C_j from S_2 to S_1. Let the height of the stacks now be $h'(S_1)$ and $h'(S_2)$. We see that $h'(S_1) = h(S_1) - r_j + h_j + r_{j+1} \leq h(S_1)$ and $h'(S_2) = h(S_2) - r_j - h_j + r_{j+1} < h(S_2)$.

So, the height and width of the folding of Figure 1(b) is no more than that of Figure 1(a). Hence, the instance I' obtained from I by replacing the component pair $((h_{j-1}, r_{j-1}), (h_j, r_j))$ by the single component $(h_{j-1} + h_j, r_{j-1})$ has the same minimum width/height folding as does I. From a minimum width/height folding for I' one can obtain one for I by replacing the component $(h_{j-1} + h_j, r_{j-1})$ by the two components of I it is composed of.

If $h_j + r_j \leq r_{j+1}$, then $h_j + r_{j+1} > r_j$ and $j < n$ (as $r_{n+1} = 0$ and $h_n + r_n > 0$). This time, I' is obtained by replacing the component pair $((h_j, r_j), (h_{j+1}, r_{j+1}))$ by the single component $(h_j + h_{j+1}, r_j)$.

The component pair replacement scheme just described may be repeated as often as needed to obtain a normalized instance $\hat{I}$. Note that the scheme terminates as each replacement reduces the number of components by one and every one instance component is normalized.

The preceding discussion leads to the normalization procedure *Normalize* of Figure 2. $C[i].h, C[i].r, C[i].f$, and $C[i].l$, respectively, give the height, routing height needed if the stack is folded at $C[i-1]$, index of first input component represented by $C[i]$, and index of the last input component represented by $C[i]$. At input, we have: $C[i].h = h_i$, $C[i].r = r_i$, $C[i].f = C[i].l = i$, $1 \leq i \leq n$, and $C[n+1].r = 0$. Note that, by definition, $C[1].r = r_1 = 0$. On output, component $C[i]$ is the result of combining together the input components $f, f+1, \ldots, l$. The heights and the r values are appropriately set. The correctness of procedure *Normalize* is established in Theorem 1. Its complexity is $O(n)$ as each iteration of the **while** loop takes constant time; the first two *case* clauses can be entered atmost a total of $n-1$ times as on each entry the number of components is reduced by 1. The **else** clause can be entered atmost $n-1$ times as on each entry *next* increases by 1 and this variable is never decreased in the procedure.

Theorem 1 : *Procedure Normalize produces an equivalent normalized component stack.*

Proof : Refer to [10]. □

Theorem 2 establishes an important property of a normalized stack. This property enables one to obtain efficient algorithms for the two folding problems considered in this paper.

Theorem 2 : *Let (h_i, r_i), $1 \leq i \leq n$ define a normalized component stack. Assume that $r_0 = r_{n+1} = 0$. The following are true:*
P1 : $r_k + \sum_{j=k}^{l} h_j + r_{l+1} < r_{k-1} + \sum_{j=k-1}^{l} h_j + r_{l+1}$, $1 < k \leq l \leq n$
P2 : $r_k + \sum_{j=k}^{l} h_j + r_{l+1} < r_k + \sum_{j=k}^{l+1} h_j + r_{l+2}$, $1 \leq k \leq l < n$

Proof : Direct consequence of C2 and C1, respectively. □

Intuitively, Theorem 2 states that the height needed by a contiguous segment of components from a normalized stack increases when the segment is expanded by adding components at either end.

3 Equal-Width Height-Constrained

The height of the layout is limited to h and we are to fold the component stack so as to minimize its width. This can be accomplished in linear time by first normalizing the stack and then using a greedy strategy to fold only when the next component cannot be accomodated in the current stack segment without exceeding the height bound h. The algorithm is given in Figure 3.

From the correctness of procedure *Normalize*, it follows that a minimum width folding of the normalized instance is also a minimum width folding of the initial instance. So, we need only to show that the **for** loop generates a minimum width folding of the normalized instance generated by the procedure *Normalize*. This follows from properties *P1* and *P2* (Theorem 2) of a normalized instance. Since a segment size cannot decrease by adding more components at either end, the

```
Procedure MinimizeWidth(C, n, h, width)
{ Obtain a minimum width folding whose height is
atmost h}
    Normalize(C, n);
    used := h; width := 1;
    for i := 1 to n do
    case
      : used − C[i].r + C[i].h + C[i + 1].r ≤ h :
      { assign C[i] to current segment }
        used := used − C[i].r + C[i].h + C[i + 1].r;
      : C[i].r + C[i].h + C[i + 1].r > h :
      {infeasible instance }
        output error message; terminate;
      :else:{start next segment, fold at C[i − 1] }
        width := width + 1;
        used := C[i].r + C[i].h + C[i + 1].r
    end;
end; {MinimizeWidth}
```

Figure 3: Procedure to obtain a minimum width folding

n	[7]	Figure 3
16	0.11	0.05
64	1.80	0.14
256	24.85	0.52

Times are in milliseconds

Table 1: Comparison of equal-width height-constrained algorithms

infeasibility test is correct. Also, there can be no advantage to postponing the layout of a component to the next segment if it fits in the current one.

Note that while we are able to solve the equal-width height-constrained problem in linear time using a combination of normalizing and the greedy method, the algorithm of [7] uses dynamic programming on the unnormalized instance and takes $O(n^2)$ time. In Table 1, we give the observed run times of the two algorithms. These were obtained by running C programs on a SUN 4 workstation. As is evident, our algorithm is considerably superior to that of [7] even on small instances.

4 Equal-Width Width-Constrained

To use parametric search [1, 2, 3, 4] to determine the minimum height folding when the layout width is constrained to be $\leq w$, we must do the following:

(1) Identify a set of candidate values for the minimum height folding. This set must be provided implicitly 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 height h is feasible, i.e., can the component stack be folded into a rectangle of height h and width w ?

In this section, for (1), we shall provide an $n \times n$ sorted matrix M (n is the number of components in the stack) of candidate values. For the feasibility test of (2), we can use procedure *MinimizeWidth* of Figure 3 by setting h equal to the candidate height value being tested and then determining if $width \leq w$ following execution of the procedure. Since the component stack needs to be normalized only once and since *MinimizeWidth* will be invoked several times, the call to *Normalize* should be removed from the procedure *MinimizeWidth* and normalization done before the first invocation of this procedure. Also, the remaining code may be modified to terminate as soon as w folds are made.

To determine the candidate matrix M, we observe that the height of any layout is given by

$$r_i + \sum_{q=i}^{j} h_q + r_{j+1}$$

for some $i, j, 1 \leq i \leq j \leq n$. This formula just gives us the height of the segment that contains components C_i through C_j. Define Q to be the $n \times n$ matrix with the elements

$$Q_{ij} = \begin{cases} r_i + \sum_{q=i}^{j} h_q + r_{j+1}, 1 \leq i \leq j \leq n \\ 0, i > j \end{cases}$$

Then for every value of w, Q contains a value that is the height of a minimum height folding of the component stack such that the folding has $width \leq w$. From Theorem 2, it follows that

$$Q_{ij} \leq Q_{i,j+1}, 1 \leq i \leq n, 1 \leq j < n$$
$$Q_{ij} \geq Q_{i+1,j}, 1 \leq i < n, 1 \leq j \leq n$$

Let $M_{ij} = Q_{n-i+1,j}, 1 \leq i \leq j \leq n$. So, M is a sorted matrix that contains all candidate values. The minimum M_{ij} for which a width w folding is possible is the minimum height width-w folding. We now need to show how the elements of M may be computed efficiently given the index pair (i, j). Let

$$H_i = \sum_{j=1}^{i} h_j, 1 \leq i \leq n$$

and let $H_0 = 0$. We see that

$$Q_{ij} = \begin{cases} r_i + H_j - H_{i-1} + r_{j+1}, i \leq j \\ 0, i > j \end{cases}$$

and so,

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

So, if we precompute the H_i's each M_{ij} can be determined in constant time. The precomputation of the H_i's takes $O(n)$ time. Now, we can use the $O(n \log n)$

n	[7]	$O(n \log n)$	$O(n \log\log n)$	$O(n \log^* n)$	$O(n)$
16	4.9	1.47	2.28	1.49	1.52
64	314.7	8.84	15.75	27.14	26.71
256	23255	45.96	76.55	169.58	169.42
4096	-	1041.90	2148.60	2597.75	2760.25

Times are in milliseconds

Table 2: Run times of equal-width width-constrained algorithms

parametric search algorithm of [4] to solve the equal-width width-constrained problem in $O(n \log n)$ time.

[1, 2, 3, 4] present several refinements of the basic parametric search technique. These refinements apply to the equal-width width-constrained problem just as well as to the path partitioning problem provided we start with a normalized instance and use the candidate matrix M defined above. These refinements result in algorithms of complexity $O(n \log\log n)$, $O(n \log^* n)$, and $O(n)$ for our component stack problem.

5 Experimental Results

The four parametric search algorithms for the equal-width height-constrained problem were programmed in C and run on a SUN 4 workstation. For comparison purposes, the $O(n^3)$ dynamic programming algorithm of [7] was also programmed. The run time performance of these five algorithms is given in Table 2. These times represent the average time for ten instances of each size. The algorithm of [7] takes much more time than each of the parametric search algorithms. However, within the class of parametric search algorithms, the $O(n \log n)$ one is fastest in the tested problem size range. This may be attributed to the increased overhead associated with the remaining algorithms. The $O(n \log n)$ algorithm is recommended for use in practice unless the number of components in a stack is very much larger than 4096.

6 Conclusions

We have shown that while the equal-width height-constrained and equal-width width-constrained stack folding problems cannot be solved by applying the greedy method and parametric search, respectively, these methods can be successfully applied if the input is first normalized. Normalization can be done in linear time. Hence the overall complexity is determined by that of applying the greedy method or parametric search to the normalized data.

We have developed a linear time algorithm for the equal-width height-constrained problem. This compares very favorably (both analytically and experimentally) with the $O(n^2)$ dynamic programming algorithm of [7].

For the equal-width width-constrained problem we have developed four algorithms of complexity $O(n \log n)$, $O(n \log\log n)$, $O(n \log^* n)$, and $O(n)$, respectively. All compare very favorably with the $O(n^3)$ dynamic programming algorithm of [7]. Experimental results indicate that the $O(n \log n)$ algorithm performs best on practical size instances.

References

[1] 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.

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

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

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

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

[6] L. Larmore, D. Gajski and A. Wu, "Layout Placement for Sliced Architecture", University of California, Irvine, Technical Report, 1990.

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

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

[9] 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.

[10] V. Thanvantri, and S. Sahni, "Folding a stack of equal width components", University of Florida, Technical Report 94-011, 1994.

[11] A. Wu, and D. Gajski, "Partitioning Algorithms for Layout Synthesis from Register-Transfer Netlists", *Proc. of International Conference on Computer Aided Design*, November 1990, pp. 144-147.