Folding A Stack Of Equal Width Components [*]

Venkat Thanvantri and 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. We develop a normalization technique that permits the first version to be solved in linear time by a greedy algorithm. The second version can be solved efficiently using normalization and parametric search. Experimental results are presented.

Keywords and Phrases

stacked bit slice architectures, folding, layout area

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

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 [10]. In the model of [6] and [10] 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 [10]. 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 stack of equal width components is comprised of variable height 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 (Figure 1(a)). If the stack is realized, physically, in this way, the area needed is $\Sigma h_i * w$ where $h_i > 0$ is the height of C_i and $w > 0$ is the width of each component. 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 $C_n, \ldots, C_{i+1}$. Figure 1(b) shows the stack of Figure 1(a) after folding at C_{i_1}, C_{i_2}. Notice that folding results in a snake-like rearrangement. While not apparent from the figure, each fold flips the left-to-right orientation of a component. As can be seen from Figure 1(b), 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_k} and $C_{i_{k+1}}$ where C_{i_k} 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 2).

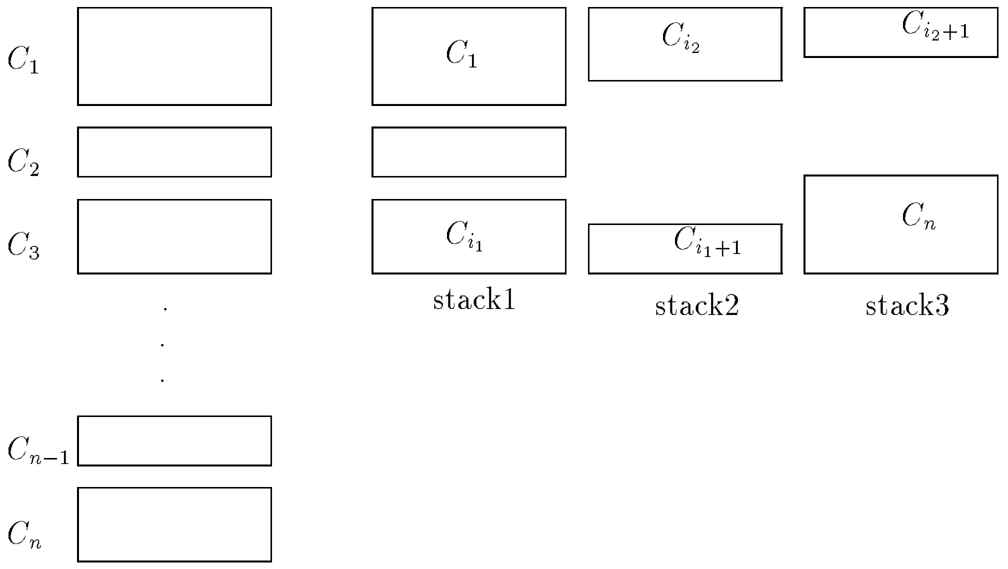
In practical situations, the height (width) of the rectangle into which the stack is to be folded may be limited (and known in advance) and we are to minimize the width (height). Several versions of folding into height (width) constrained rectangles were considered by Paik and Sahni [7]. Their results are summarized in Table 1.

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

(1) *Equal-width, height-constrained with routing area at stack ends.* In this problem, we are to fold a stack of equal height components into a rectangle of given height so as to minimize the width (and hence area) of the rectangle. For this problem, the algorithm of [7] runs in $O(n^2)$ time. We develop an $O(n)$ algorithm.

(2) *Equal-width, width-constrained with routing area at stack ends.* Here the width of the rectangle into which the folding occurs is given and we are to minimize its height (and hence area). Four algorithms with complexity $O(n \log n)$, $O(n \log \log n)$, $O(n \log^* n)$, and $O(n)$ respectively are obtained. Experimental results indicate that the $O(n \log n)$ algorithm is fastest in practice. This is due to the fact that this algorithm has least overhead.

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.* In Section 2 we describe our normalization technique and then in Section 3, we show how this results in a linear time algorithm for the equal-width height-constrained problem. Parameterized



(a) Component Stack (b) Folded into three stacks

Figure 1: Stack of equal width components

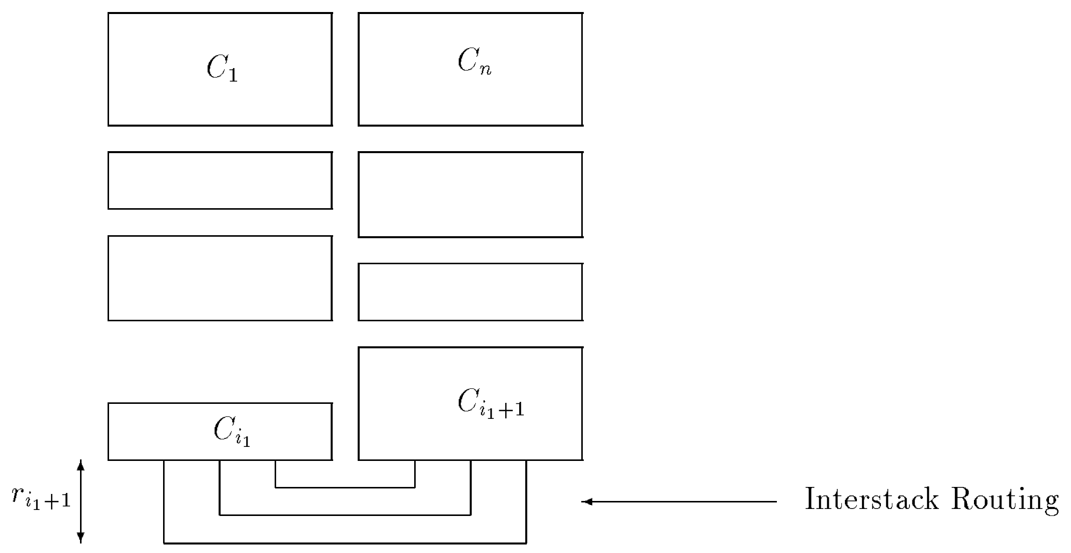


Figure 2: Routing space reserved

	Routing area at stack ends	
	No	Yes
Equal width, height constrained	$O(n)$	$O(n^2)$
Equal width, width constrained	$O(n)$	$O(n^3)$
Equal height, height constrained	$O(n^4 \log n)$	$O(n^4 \log n)$
Equal height, width constrained	$O(n^4 \log^2 n)$	$O(n^4 \log^2 n)$
Variable heights and widths, height constrained	$O(n^5 \log n)$	$O(n^5 \log n)$
Variable heights and widths, width constrained	$O(n^5 \log^2 n)$	$O(n^5 \log^2 n)$

Table 1: Summary of results of [7]

(reproduced from [7])

searching is described in Section 4 and then used in Sections 5 to obtain the algorithms for the equal-width width-constrained problem. Experimental results comparing the relative performance of the various algorithms for the equal-width width-constrained problem are given in Section 6.

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 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 first note that (since $h_j > 0$) 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

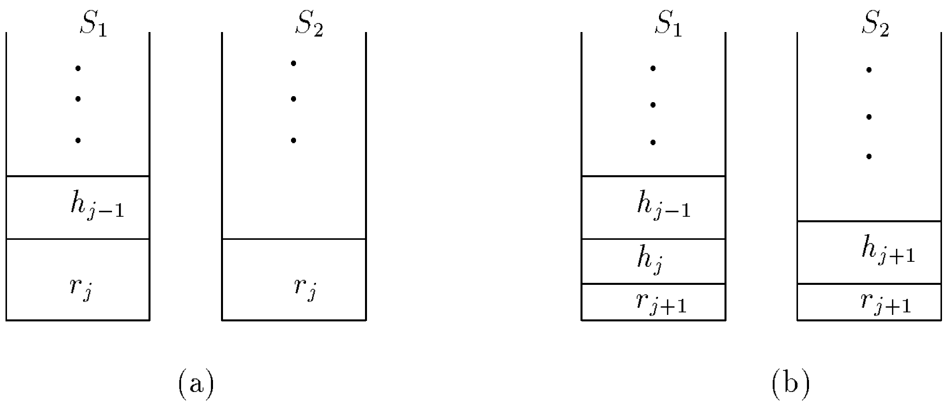


Figure 3: Case when $h_j + r_{j+1} \leq r_j$

(Figure 3(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 3(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 3(b) is no more than that of Figure 3(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 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 proof is similar to the previous case.

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 4. The input to this procedure is a component stack $C[1] \ldots C[n]$ and the output is a normalized stack $C[1] \ldots C[n]$ (the input n (say n'') will be generally larger that the output n (say n')).

```
**Procedure** *Normalize*($C$,$n$)
{ Normalize the component stack $C[1] \ldots C[n]$}
    $i := 1; next := 2;$
    **while** $next \leq n + 1$ **do**
      **case**
        : $C[i].h + C[next].r \leq C[i].r$ :
            {Combine with $C[i-1]$}
            $C[i-1].h := C[i-1].h + C[i].h;$
            $C[i-1].l := C[i].l;$
            $i := i - 1;$
        : $C[i].h + C[i].r \leq C[next].r$ :
            {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 4: Normalizing a stack

$C[i].h, C[i].r, C[i].f$, 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 : The procedure maintains the following invariant at the start of each iteration of the **while** loop:

Invariant: Normalizing conditions C1 and C2 are satisfied by all components $C[j], j < 1$.

This is clearly true when $i = 1$ as there is no component $C[j]$ with $j < 1$. If the invariant is true at the start of some iteration, then it is true at the end of that iteration. To see this, note that if we enter the first clause of the **case** then following the execution of this clause, $C[j].h, C[j].r, C[j+1].r, j < i'$, where i' is the value of

i following execution of the clause, are unchanged. So, the execution does not affect C1 and C2 for $j < i'$. If the second **case**-clause is entered, then again C1 and C2 are unaffected by the execution for $j < i$ as $C[j].h, C[j].r$, and $C[j+1].r$, $j < i$ are unchanged. When the third clause is entered the validity of C1 and C2 for $j < i'$ follows from the fact that the conditions for the first two clauses are false.

On termination, $next = n + 2$. The last iteration of the **while** loop could not have entered the first clause of the **case** statement as in this clause, $next$ is not increased. While in the second clause, $next$ is increased, the condition $C[i].h + C[i].r < C[next].r$ cannot be true in the last iteration as now $next = n'' + 1$ (n'' is the input value of n), $C[i].h + C[i].r \geq 0, C[n''].r = 0$. So, the last iteration caused execution of the third clause of the **case** statement. As a result, $C[n'']$ is moved to position $n'' + 1$ of C. From the invariant, it follows that C1 and C2 are satisfied for $j < i' = n'' + 1$ (note i' is the final value of i). Hence the output component stack $C[1] \ldots C[n']$ is normalized. $\square$

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:*

$$P1: \qquad r_k + \sum_{j=k}^{l} h_j + r_{l+1} < r_{k-1} + \sum_{j=k-1}^{l} h_j + r_{l+1}, 1 < k \leq l \leq n$$

$$P2: \qquad r_k + \sum_{j=k}^{l} h_j + r_{l+1} < r_k + \sum_{j=k}^{l+1} h_j + r_{l+2}, 1 \leq k \leq l < n$$

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 5.

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 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 2, we give the observed run times of the two algorithms. These were obtained

```
**Procedure** *MinimizeWidth(C, n, h, width)*
{ Obtain a minimum width folding whoose height is atmost $h$}
    $Normalize(C, n)$;
    $used := h; width := 1$;
    **for** $i := 1$ **to** $n$ **do**
    **case**
      : $used - C[i].r + C[i].h + C[i+1].r \leq h$ :
      { 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 5: Procedure to obtain a minimum width folding

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 Parametric Search

In this section, we provide an overview of the parametric search method of Frederickson [4], which uses developments by Frederickson and Johnson [1, 2] and Frederickson [3]. This overview has, however, been tailored to suit our application here and is not as general as that provided in [1, 2, 3, 4].

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

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

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.

n	[7]	Figure 5
16	0.11	0.05
64	1.80	0.14
256	24.85	0.52

Times are in milliseconds

Table 2: Comparison of equal-width height-constrained algorithms

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 6). 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 $PSEARCH(S,\lambda_1,\lambda_2,dimension,finish)$;
  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 $(\lambda_1 < x < \lambda_2)$ then
        if $F(x)$ is feasible then $\lambda_2 := x$
        else $\lambda_1 := x$;
      Eliminate from $S$ all matrices that have no values
      such that $\lambda_1 < x < \lambda_2$;
    end;
  until $dimension^2 * |S| \leq finish$;
end; {$PSEARCH$}
```

Figure 6: Procedure for parametric search. (Restricted version of Procedure *MSEARCH* of [4].)

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 3 : *[4] 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 $O(n)$.* $\square$

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

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 4 : *[4] 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 $O(mn)$.* $\square$

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

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 [1, 2, 3, 4].

Procedure *PSEARCH* is a restricted version of procedure *MSEARCH* of [4]. 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 [1, 2]. We experimented with both the formulations and found that for our stack folding application, the three iteration formulation of Figure 6 is faster by approximately 43%.

5 Equal-Width Width-Constrained

To use parametric search 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 5 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 for $O(\log n)$ candidate values, 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.

Since feasibility testing and normalization each take linear time, from Corollary 1, it follows that the complexity of the described parametric search to find the minimum height folding is $O(n + t(n) \log n) = O(n + n \log n) = O(n \log n)$.

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

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

$$Q_{ij} \ge Q_{i+1,j}, 1 \le i < n, 1 \le j \le n$$

Let $M_{ij} = Q_{n-i+1,j}, 1 \le i \le j \le n$. 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

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

and let $H_0 = 0$. We see that

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

and so,

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

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. Hence, the overall complexity of the parametric search algorithm to find the minimum height folding remains $O(n \log n)$.

We note that our $O(n \log n)$ algorithm is very similar to the $O(n \log n)$ algorithm of [1] to partition a path into k subpaths such that the length of the shortest subpath is maximized. The differences are that

(1) We need to normalize the component stack before parametric search can be used. and,

(2) The definition of M_{ij} needs to be adjusted to account for the routing heights r_i and r_{j+1} needed at either end of the stack.

[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.

6 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 3. These times represent the average time for ten instances of each size.The component heights were obtained using a random number generator. The four parametric search algorithms did not exhibit much run time variation among instances with the same number of components. The algorithm of [7] takes much more time than each of the parametric search algorithms. 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.

n	[7]	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log \log n)$	$\mathrm{O}(n \log^* n)$	$\mathrm{O}(n)$
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 3: Run times of equal-width width-constrained algorithms

7 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 $\mathrm{O}(n^2)$ dynamic programming algorithm of [7].

For the equal-width width-constrained problem we have developed four algorithms of complexity $\mathrm{O}(n \log n)$, $\mathrm{O}(n \log\log n)$, $\mathrm{O}(n \log^* n)$, and $\mathrm{O}(n)$, respectively. All compare very favorably with the $\mathrm{O}(n^3)$ dynamic programming algorithm of [7]. Experimental results indicate that the $\mathrm{O}(n \log n)$ 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", *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 11(1), Jan. 1992, 102-114.

[7] 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.

[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] A. Wu, and D. Gajski, "Partitioning Algorithms for Layout Synthesis from Register-Transfer Netlists", *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol 11(4), Apr. 1992, 453-463.