*VLSI DESIGN*
1998, Vol. 7, No. 4, pp. 425–436
Reprints available directly from the publisher
Photocopying permitted by license only



# CLOTH MEASURE: A Software Tool for Estimating the Memory Requirements of Corner Stitching Data Structures

DINESH P. MEHTA

*Department of Computer Science, University of Tennessee, Space Institute, Tullahoma, TN 37388-9700*

*(Received 15 April 1995; In final form 20 October 1995)*

In a previous paper [1], we derived formulae for estimating the storage requirements of the Rectangular and L-shaped Corner Stitching data structures [2, 3] for a given layout. These formulae require the computation of quantities called violations, which are geometric properties of the layout. In this paper, we present optimal $\Theta(n \log n)$ algorithms for computing violations, where $n$ is the number of rectangles in the layout. These algorithms are incorporated into a software tool called CLOTH MEASURE. Experiments conducted with CLOTH MEASURE show that it is a viable tool for estimating the memory requirements of a layout without having to implement the corner stitching data structures, which is a tedious and time-consuming task.

*Keywords:* Corner stitching, data structures, rectangles, computational geometry, memory

## 1. INTRODUCTION

When a CAD systems designer is considering the adoption of an existing software tool or algorithmic technique for his/her CAD system, it is useful if additional data regarding the performance characteristics and resource requirements of that tool/technique are available. Examples of such quantities include computation time on established benchmarks, memory and hard disk requirements, OS requirements, etc. In this paper, we provide a tool for estimating the storage requirements of the Rectangular Corner Stitching (RCS) and L-shaped Corner Stitching (LCS) data structures on a given layout. Memory requirements are particularly relevant in the context of VLSI layouts because of the large number of components contained in a single layout. If the amount of memory required by a layout using a particular data structure exceeds the memory available on the system, it may not be possible to process the layout on that system. Even if there is sufficient memory, the amount of memory required by the layout could have some bearing on the runtime of the software. (For example, a data structure which uses less memory might require less disk swaps than a data structure which requires more memory). Thus, the estimator proposed in this paper would be used

up-front by a CAD systems designer on sample layouts available to the designer. The designer will choose a data structuring technique for his/her application based, in part, on the information provided by the memory estimator and on the available hardware.

Next, we mention why a memory estimator is needed for corner stitching: unlike simpler data structures such as arrays and linked lists, it is not trivial to manually estimate the storage requirements of RCS and LCS. For example, if $n$ items are inserted into a linked list, then the amount of storage required is $n$ multiplied by the number of bytes required by a single list node. Corner stitching has the unique property that the insertion of $n$ rectangles does not imply that there are $n$ nodes in the data structure. The exact total number of nodes in the data structure is considerably more than $n$ and depends on the relative positions of the $n$ rectangles, and in LCS, on the order in which they are inserted!

We begin by reviewing the corner stitching data structure. Corner stitching is a data structuring technique proposed by Outsterhout [2] for representing rectangular tiles in interactive VLSI layout editing systems and was used to implement the Magic system [4]. It was preferred over other data structures for VLSI layouts such as linked lists [5], bins [6, 7] and neighbor pointers [8] because it allows fast, localized algorithms for a variety of interactive and batch operations [2]. Corner stitching was subsequently extended to trapezoidal tiles by Marple *et al.* and was used to implement the Tailor system [9]. Marple *et al.* compared corner stitching with linked lists, K-D trees, and quad trees. Although there were certain advantages and disadvantages associated with each data structure, corner stitching was, on balance, found to be the most appropriate. See [9] for a detailed discussion on the pros and cons of each data structure. Corner stitching was later extended to curved tiles, by Séquin and Facanha [10]. Both extensions are similar to the original data structure in that "the topological and conceptual issues remain virtually unchanged" [10]. The difference is that, in order to accommodate tiles of more complex shapes than rectangles, "the low-level geometric operations are replaced with procedures that are more sophisticated-and more difficult to implement" [10]. Blust and Mehta [3] extended the corner stitching data structure of Ousterhout so that, in addition to representing rectangular tiles, it also represented $L$-shaped tiles. This extension required a modification of the topology of the corner stitching data structure. It was seen that the LCS data structure, while retaining the advantages of corner stitching over simpler data structuring methods, required less memory than the alternative approach of partitioning each L-shaped object into two rectangular objects and then using RCS. Implementing the corner stitching data structure is a rather tedious task as indicated by the following quote [10] attributed to John Ousterhout, the inventor of the Corner Stitching data structure:

"*Corner-Stitching is pretty straightforward at a high level, but it can become much more complicated when you actually sit down to implement things, particularly if you want the implementation to run fast...*"

Our own experience supports this opinion. Further, we note that the LCS data structure is even more complex than the RCS data structure, which only exacerbates the problem.

In view of this, we have provided, in a previous paper [1], a general formula for the memory requirements of each of the RCS and LCS data structures on a given layout. These formulae require knowledge about certain geometric properties of the layout called *violations* of the **CV** property [1]. However, [1] does not describe how to compute violations. Therefore, in this paper, we present optimal algorithms to compute violations of the **CV** property in a layout. This results in a software tool CLOTH MEASURE for estimating the storage requirements of a layout *without having to implement or run the Corner Stitching data structure*. Thus, these techniques are expected to be useful to a CAD System designer in choosing

which data structure to use (without having to purchase, adapt, or implement it locally) for his/her layout.

In Section 2, we review definitions of the **CV** property and its violations. Section 3 describes the algorithms for computing violations of the **CV** property. Finally, Section 4 provides experimental performance data of CLOTH MEASURE.

## 2. REVIEW OF THE CV PROPERTY AND ITS VIOLATIONS

In this section, we review definitions and concepts presented in [1] that are relevant to our discussion.

### 2.1. The CV Property

A layout consisting of solid, rectangular tiles is said to have the **CV** property iff

1. No two solid tiles touch each other and
2. No two horizontal edges of these tiles are collinear and mutually visible; i.e., it is not possible to draw a horizontal line segment through two horizontal edges without the segment intersecting a third solid tile.

### 2.2. Violations of the CV Property

We first classify the violations of the **CV** property as follows:

1. A *horizontal* violation is said to occur between two solid tiles $A$ and $B$ iff there exists a horizontal segment $h$ such that
   (a) A segment of a horizontal side of $A$ and a segment of a horizontal side of $B$ are contained in $h$, and
   (b) No portion of any other tile is contained in $h$.

   There are four types of horizontal violations as shown in Figure 1.

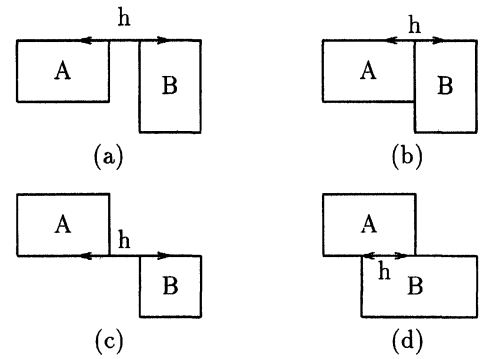


FIGURE 1 Horizontal violations.

(a) Same side, no touching (*sn*) horizontal violation: Both $A$ and $B$ are on the same side of the horizontal segment, and $A$ and $B$ do not touch each other. Let $h_{sn}$ denote the number of such violations in the layout.
(b) Same side, touching (*st*) horizontal violation: Both $A$ and $B$ are on the same side of the horizontal segment, and $A$ and $B$ touch each other. Let $h_{st}$ denote the number of such violations in the layout.
(c) Opposite side, no touching (*on*) horizontal violation: $A$ and $B$ are on opposite sides of the horizontal segment, and $A$ and $B$ do not touch each other. Let $h_{on}$ denote the number of such violations in the layout.
(d) Opposite side, touching (*ot*) horizontal violation: $A$ and $B$ are on opposite sides of the horizontal segment, and $A$ and $B$ touch each other. Let $h_{ot}$ denote the number of such violations in the layout.

Figure 2 shows three cases where there is no horizontal violation between $A$ and $B$. Note

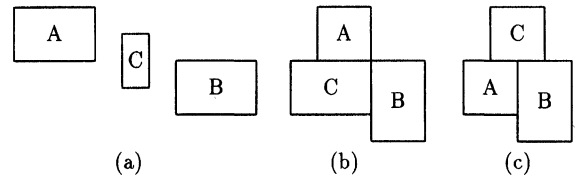


FIGURE 2 There are no horizontal violations between $A$ and $B$.

that, in each case, there is a violation between $A$ and $B$ if $C$ is removed.

2. *Vertical* violations are of two types:

(a) A same side (*ss*) vertical violation is said to occur between two solid tiles $A$ and $B$ iff there exists a vertical segment $v$ such that

(i) $v$ is the union of a vertical side of $A$ and a vertical side of $B$,
(ii) $A$ and $B$ are on the same side of $v$.

Let $v_s$ denote the number of such violations in the layout. Figure 3(a) shows a same-side vertical violation between $A$ and $B$, while, in Figure 3(b), there is no violation between $A$ and $B$. This is because $v$ is not the union of vertical sides of $A$ and $B$.

(b) A *vertical group* of tiles is a maximal list of solid tiles $(T_1, T_2, \ldots, T_m)$ such that there exists a vertical segment $v$ where

(i) $v$ is the union of vertical sides of $T_i$ for $1 \leq i \leq m$, and
(ii) $T_i$, for $1 \leq i \leq m$, are on the same side of $v$.

$v$ is known as the *common side* of the vertical group.

An opposite side (*os*) vertical violation is said to occur between two vertical groups $G$ and $H$ iff there exists a vertical segment $v$ such that

(i). $v$ is the intersection of the common side of $G$ and the common side of $H$,
(ii). $G$ and $H$ are on opposite sides of $v$, and
(iii). $v$ consists of more than one point.

Let $v_0$ denote the number of such violations in the layout. Figure 4(a) shows three vertical groups (I), (II), and (III). There is an opposite-side, vertical violation between (I) and (II) and (I) and (III). Figure 4(b) also shows three vertical groups (each of which consists of exactly one rectangle). There is no opposite-side vertical violation between any pair of groups.

LEMMA 1 *A layout satisfies the* **CV** *property if and only if it does not contain any of the violations described above.*

*Proof* Proved in [1]. □

Figure 5 shows a layout, consisting of 19 solid tiles and 26 vacant tiles, that violates the **CV** property. We first enumerate the horizontal violations:

1. *sn* violations: (C, G), (G, J), (J, Q), (D, H), (O, R).
2. *st* violations: (B, C).
3. *on* violations: (G, I), (E, I), (R, S).
4. *ot* violations: (A, B), (C, D), (G, H), (I, L), (L, N), (J, K), (K, M), (O, P), (Q, R)

Next, we enumerate the vertical violations:

1. Same side violations: (A, B), (A, B), (G, H), (I, L), (L, N), (J, K), (J, K), (K, M), (O, P),

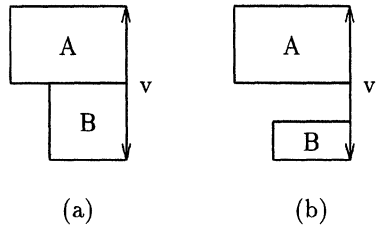


FIGURE 3 (a) Same-side, vertical violation. (b) No same-side vertical violation between $A$ and $B$.

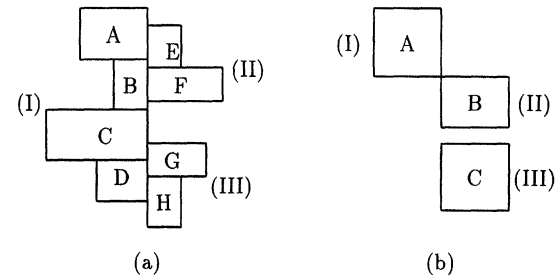


FIGURE 4 (a) Opposite-side, vertical violations. (b) No opposite-side vertical violations.

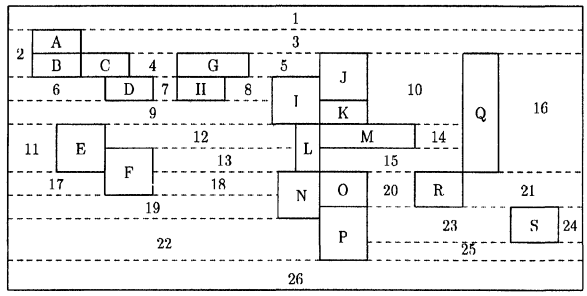


FIGURE 5 Example layout with violations

(O, P). Note that there are *two* violations (A, B), (J, K), and (O, P) because, in each pair, both the left vertical sides and right vertical sides of each rectangle are involved in a same side violation.

2. Opposite side violations: (AB, C), (E, F), (ILN, JKM), (ILN, OP).

So, $h_{sn} = 5$, $h_{st} = 1$, $h_{on} = 3$, $h_{ot} = 9$, $v_s = 10$, and $v_0 = 4$; i.e., the total number of violations in the layout is 32.

### 2.3. Memory Requirements of RCS and LCS for Arbitrary Circuits

The following theorems which are proved in [1] provide formulas for the memory requirements of the RCS and LCS data structures. These formulas require knowledge of the number of each type of violation in the layout.

THEOREM 1 *The RCS data structure representing a set of n solid, rectangular tiles with k violations of the* ***CV*** *property requires* $28(4n+1-k)$ *bytes.*

THEOREM 2 *The LCS data structure requires between* $82n-18k$ *and* $103n+28-(25-3\beta)k$ *bytes for a set of n solid tiles with k violations of the* ***CV*** *property, where*

$$\beta = \frac{h_{ot} + v_0 + h_{on} + h_{sn}}{k}. \qquad (1)$$

## 3. ALGORITHMS FOR COMPUTING VIOLATIONS

The input to our algorithms for computing violations is a list of $n$ non-overlapping rectangles. The algorithms for computing vertical and horizontal violations require lists of vertical and horizontal sides of the input rectangles, respectively. These are trivially obtained from the list of input rectangles. It is assumed that each vertical side is stored as a structure with four data items ($x$, $y_1$, $y_2$, *left*) where $x$ refers to the position of the side and $(y_1, y_2)$ represents the vertical range of the side. The Boolean variable *left* is 1 if the side is the left side of a rectangle and 0 if it is the right side of a rectangle. Similarly, each horizontal side is stored as a structure with four data items $(y, x_1, x_2, top)$. The Boolean variable *top* is 1 if the side represents an upper side of a rectangle and 0, if it represents a bottom side of a rectangle. Since each rectangle has two vertical and two horizontal sides, the total number of vertical or horizontal sides is $2n$.

### 3.1. Computing Horizontal Violations

We begin with a simple observation:

LEMMA 2 *Horizontal violations only take place between horizontal sides with the same y-coordinates.*

*Proof* Follows from the definition of horizontal violations. □

DEFINITION A *chain* is a maximal set of horizontal sides such that the horizontal segment joining the leftmost left endpoint to the rightmost right endpoint of all sides in the set does not *cut* a vertical side (though it may *touch* a vertical side). We will say that an object is *cut* by a line, if that line splits it into two objects, one on each side of the line. An object is *touched* by a line if they intersect, but no portion of the object lies on one of the sides of the line.

Figure 6 shows three chains. Chain 1 consists of the lower horizontal side of 1 and the upper sides of 2 and 3; chain 2 consists of the lower side of 5 and the upper sides of 4 and 6; chain 3 consists of the upper side of 7. Note that each chain is maximal; if more horizontal sides are added to any chain, then the horizontal segment mentioned in the definition of a chain cuts either $A$ or $B$ or both.

LEMMA 3 *Horizontal violations only take place among horizontal sides belonging to the same chain.*

*Proof* Suppose there is a horizontal violation between two sides in different chains. Then, these sides must either touch each other or be visible to each other. However, since they must be separated by a vertical side from the definition of a chain, and since rectangles do not overlap, both these scenarios are impossible. □

*Algorithm ComputeHorizViols* outlines a planesweep algorithm for computing horizontal violations. The algorithm consists of three steps. Details of Steps 2(b) and 2(c) are provided later:

*Algorithm ComputeHorizViols*

1. Initialize $T$, a balanced search tree to the empty tree. $T$ represents, in left-to-right order, the horizontal cross-section of rectangles that are cut by a horizontal sweepline. Rectangles that are touched (but not cut) by the sweepline are not included in $T$. Since rectangles do not overlap, nor do their horizontal cross-sections in $T$. Hence, the $x$-coordinate of the left endpoint of the cross section may be used as the key on which $T$ is maintained.
2. Next, a horizontal sweepline moves from the bottom of the plane towards the top of the plane. In order to facilitate the planesweep algorithm, we first sort the array of horizontal sides so that side $h_1$ precedes side $h_2$ if and only if one of the following is true:

$$h_1.y < h_2.y.$$
$$h_1.y = h_2.y \quad \text{and} \quad h_1.x_1 < h_2.x_1$$

This results in all horizontal sides with the same $y$-coordinate being stored contiguously as a single group in the array. Moving the sweepline is achieved by moving from one group of horizontal sides with the same $y$-coordinate to the next. For each $y$-coordinate $y_i$ at which the top or bottom horizontal side of a rectangle is encountered, do the following:

(a) Delete all horizontal sides at $y_i$ that are the top sides of their rectangles from $T$. These rectangles are only touched, but not cut by the sweepline at $y_i$.
(b) Identify all chains at $y_i$. Details are discussed separately.
(c) Compute violations in each chain at $y_i$. Details are discussed separately.
(d) Insert all horizontal sides at $y_i$ that are bottom sides of their rectangles into $T$. These rectangles may be cut later when the sweepline is moved up.

3. Output $h_{ot}$, $h_{on}$, $h_{st}$ and $h_{nt}$.

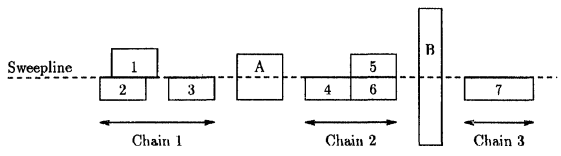


FIGURE 6 Chains.

**Step 2(b) Identify Chains:** We identify chains by traversing a set of all horizontal sides with the same $y$-coordinate from left to right. This is facilitated by the manner in which horizontal sides are sorted. Each side is searched for in the search tree. Since rectangles do not overlap, the element will not be found; and the search will lead to an external node (represented by a null pointer) in the tree. A chain consists of all sides in this set that lead to the same external node; that is all sides that are between the same pair of rectangle cross-sections formed by the sweepline. All sides in a chain occur contiguously because horizontal sides are sorted in left to right order of their left endpoints. To determine whether two sides are in

the same chain, we associate a Boolean variable *mark* with each tree pointer. *mark* is initialized to 0. When the first side in a chain is encountered, the search leads to a null tree pointer. The variable *mark* associated with this pointer, which was 0, is set to 1. Each subsequent side in the same chain encounters the same pointer, and therefore, the associated variable *mark*, which is now 1. This indicates that the new side belongs to the current chain. The first side in the next chain encounters another null pointer whose variable *mark* is 0. This signals that the current chain is complete. The variable *mark* associated with this chain is reset to 0; and the algorithm proceeds to identify the next chain in the group.

Figure 7 shows the balanced search tree corresponding to the sweepline and rectangles of Figure 6. The sweepline cuts rectangles $A$ and $B$; so their $x_1$ coordinates are stored in the tree. The other rectangles are touched, but not cut, by the sweepline and therefore not included in the tree. Any attempt to search for the $x_1$ coordinates of the horizontal sides contained in the sweepline in the tree will lead to one of the three external nodes of the tree. Sides of rectangles 1, 2, and 3 lead to external node $E_0$; sides of rectangles 4, 5, and 6 lead to external node $E_1$; the single side of rectangle 7 leads to external node $E_2$. Each set of sides forms a chain.

**Step 2(c) Identify violations within a chain:** To identify violations among sides in a chain, we move a set of three sides of the chain from left to right. Let $s_1$, $s_2$, and $s_3$ represent these three sides from left to right. Note that $s_2$ does not always immediately follow $s_1$, but $s_3$ always immediately follows $s_2$. Since the array of horizontal sides is sorted so that all sides with the same $y$ field are sorted in increasing order of $x_1$, we have $s_1.x_1 \leq s_2.x_1 \leq s_3.x_1$. Figure 8 identifies all possible cases (Excluding symmetric cases). Table I describes the action to be taken in each case. We interpret the action "move $s_1$ to position $i$" to mean that $s_1$ is now the side in position $i$ of the array, and that $s_2$ and $s_3$ are the sides in positions $i+1$ and $i+2$, if these sides are part of the current chain. Similarly, the action "move $s_2$ to position $i$" (Case 1) means that $s_2$ is now the side in position $i$ and $s_3$ is the side in position $i+1$, if these sides are part of the current chain. In this case, $s_1$ is not moved. In

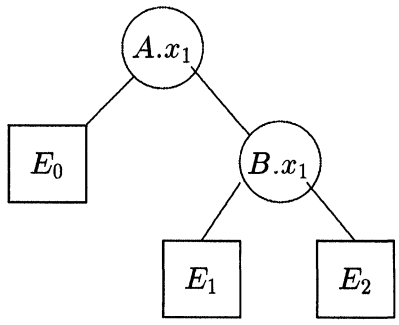


FIGURE 7 Snapshot of the search tree.

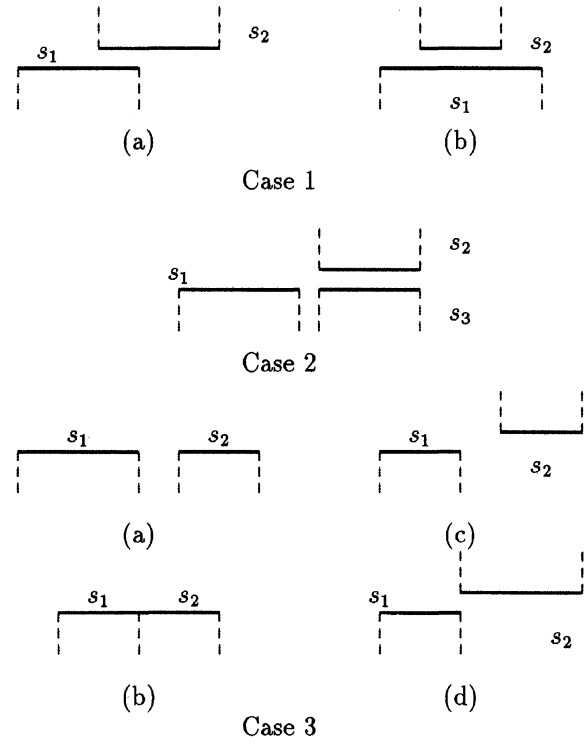


FIGURE 8 Three cases in Step 2(c) of Algorithm Compute-HorizViols. In each case, the thick horizontal lines represent horizontal sides of rectangles are drawn at below those corresponding to the bottoms of rectangles for clarity. In case, $s_1$, $s_2$, and $s_3$ are at the same height.

some cases, either $s_3$ or both $s_2$ and $s_3$ are not defined (Cases 3 and 4). This happens when the current chain has run out of sides. Side $s_3$ is not relevant in Case 1, and is not shown. It can be verified that the correct action is taken for each case.

LEMMA 4 *The computational complexity of Algorithm Compute Horiz Viols is* $\Theta(n \log n)$, *where n is the number of rectangles in the layout.*

*Proof* Steps 1 and 3 of the algorithm both take $\Theta(1)$ time. Step 2 requires the $2n$ horizontal sides to be sorted. This is accomplished in $\Theta(n \log n)$ time. Excluding insertion into, deletion from, or searching of $T$, $\Theta(1)$ time is spent on each side in Steps 2(a), 2(b), and 2(d). At most $n$ insertions to, $n$ deletions from, and $2n$ searches of $T$ are made during the course of the algorithm. Each of these operations takes $\Theta(\log n)$ time. Hence, the complexity of the algorithm excluding step 2(c) is $\Theta(n \log n)$.

Step 2(c) computes the number of violations for each chain. Each case of Step 2(c) (listed in Tab. I) takes $\Theta(1)$ time. Furthermore, each case increments the count of one of the four horizontal violations, except when the end of the chain is reached. An exception to this is Case 2, which is encountered only when there is a horizontal violation between $s_2$ and $s_3$. The number of times Case 2 is encountered is bounded by the number of horizontal violations in the layout. The total complexity of Step 2(c) is therefore $\Theta(h + c)$, where $h$ is the total number of horizontal violations in the layout and $c$ is the total number of chains in the layout. If $h$ exceeds $3n+1$, then, from Theorem 1, RCS requires negative memory, which is not possible. So, $h$ is $O(n)$. $c$ cannot exceed the number of horizontal sides and is $O(n)$. Step 2(c) consumes $O(n)$ time and the entire algorithm requires $\Theta(n \log n)$ time. □

### 3.2. Computing Vertical Violations

We begin with two simple observations:

LEMMA 5 *Vertical violations only take place between vertical sides with the same x coordinates.*

*Proof* Follows from definition of vertical violations. □

LEMMA 6 *The common side of a vertical group is composed of a sequence of vertical sides such that each pair of adjacent vertical sides has a same-side vertical violation.*

*Proof* Follows from definition of vertical violations. □

TABLE I Table showing different cases for computing horizontal violations within a chain

| Cases | | Conditions | Actions |
|---|---|---|---|
| 1 | | $s_1 \cdot x_2 > s_2 \cdot x_1$ | $h_{ot}++;$ |
| | *a* | $s_1 \cdot x_2 \leq s_2 \cdot x_2$ | **if** $(s_1 \cdot x_2 == s_2 \cdot x_2)$ move $s_1$ to posn immediately after $s_2$ **else** move $s_1$ to $s_2$'s position |
| | *b* | $s_1 \cdot x_2 > s_2 \cdot x_2$ | move $s_2$ 1 position to the right |
| 2 | | $s_2 \cdot x_1 == s_3 \cdot x_1$ | No viols between $s_1$ and $s_2$ /$s_3$ ` Move $s_1$ to $s_2$'s position |
| 3 | | $s_2 \cdot x_1 < s_3 \cdot x_1$ or $s_3$ does not exist | |
| | *a* | $s_1 \cdot top == s_2 \cdot top$ and $s_1 \cdot x_2 < s_2 \cdot x_1$ | $h_{sn}++$ · **if** $s_3$ exists, move $s_1$ to $s_2$'s position **else** go to next chain |
| | *b* | $s_1 \cdot top == s_2 \cdot top$ and $s_1 \cdot x_2 == s_2 \cdot x_1$ | $h_{st}++$ · **if** $s_3$ exists, move $s_1$ to $s_2$'s position **else** go to next chain |
| | *c* | $s_1 \cdot top \neq s_2 \cdot top$ and $s_1 \cdot x_2 < s_2 \cdot x_1$ | $h_{on}++$ · **if** $s_3$ exists, move $s_1$ to $s_2$'s position **else** go to next chain |
| | *d* | $s_1 \cdot top \neq s_2 \cdot top$ and $s_1 \cdot x_2 == s_2 \cdot x_1$ | $h_{ot}++$ · **if** $s_3$ exists, move $s_1$ to $s_2$'s position **else** go to next chain |
| 4 | | $s_2$ and $s_3$ do not exist | Go to next chain |

We now outline an algorithm for computing all vertical violations in the layout. There are two steps in the algorithm:

1. Compute same-side violations and simultaneously obtain common sides for each vertical group.
2. Compute opposite-side violations using the common sides computed in Step 1.

**Step 1:** First, sort the array of vertical sides so that side $v_1$ precedes side $v_2$ if and only if one of the following is true:

$$v_1 \cdot x < v_2 \cdot x.$$
$$v_1 \cdot x = v_2 \cdot x \quad \text{and} \quad v_1 \cdot left > v_2 \cdot left$$
$$v_1 \cdot x = v_2 \cdot x \quad \text{and} \quad v_1 \cdot left = v_2 \cdot left \quad \text{and}$$
$$v_1 \cdot y_1 < v_2 \cdot y_1$$

Notice that this results in all vertical sides making up a common side being stored contiguously in bottom-top order. Same-side violations can now be computed by traversing a sequence of left (or right) sides at each $x$-coordinate from bottom to top and checking to see if adjacent sides touch. If so, the count of same-side vertical violations is incremented. All vertical sides making up a common side are merged to form the common side. The common side is then stored in the next available location in array *commonsides*, which will be used in Step 2.

**Step 2:** Array *commonsides* is already sorted so that side $v_1$ precedes $v_2$ if and only if one of the three conditions listed in Step 1 is satisfied. The list of right common sides and the list of left common sides of rectangles at the same $x$-coordinate are stored in two adjacent groups of contiguous locations. All opposite side vertical violations occur between pairs of common sides, one from each list. These may be obtained by traversing both lists in left to right order. At each stage, the current sides from each list are compared to see if there is a violation between them. The algorithm then moves to the next side in one or both lists, as appropriate. This is repeated for each $x$-coordinate at which vertical sides exist.

**Complexity:** Sorting $2n$ sides requires $\Theta(n \log n)$ time. Steps 1 and 2, both involve traversing the array once, with constant time spent at each element. So, Steps 1 and 2 take $\Theta(n)$ time. The overall time for the algorithm is $\Theta(n \log n)$.

### 3.3. Proof of Optimality

THEOREM 3 $\Theta(n \log n)$ *is a lower bound for the computation of violations.*

*Proof* We will prove the theorem by reducing the problem of sorting $n$ integers to the problem of computing violations. The reduction itself takes $\Theta(n)$. Therefore, if it is possible to compute violations in $o(n \log n)$, then it is also possible to sort $n$ integers in $o(n \log n)$, which is a contradiction.

1. Let $l_i$, $1 \leq i \leq n$, be $n$ distinct integers that are to be sorted. Form $n$ rectangles $R_i$, $1 \leq i \leq n$, such that $R_i \cdot x_1 = l_i - 0.4$, $R_i \cdot y_1 = 0$, $R_i \cdot x_2 = l_i + 0.4$, and $R_i \cdot y_2 = i$. Since the $l_i$ are distinct, the transformation results in $n$ non-overlapping rectangles that have $n-1$ horizontal same-side, no-touching violations Figure 9. This transformation takes $\Theta(n)$ time.
2. We now run our algorithm to compute violations on the set of rectangles. Assume that the algorithm prints a violation between $R_i$ and $R_j$, $R_i.x_1 < R_j.x_1$, as $(i, j)$. Note that $(i, j)$ gets printed if and only if $l_i$ is the immediate predecessor of $l_j$ in the sorted order of $l_i$s.
3. We establish a pointer from $i$ to $j$ for each such ordered pair reported and mark $j$. The first element in the sorted is $l_k$, $1 \leq k \leq n$, such that $k$ is the only unmarked integer. The remaining elements in the sorted list may be obtained by following the pointers established earlier in this step. This reverse transformation also takes $\Theta(n)$ time. □

## 4. EXPERIMENTAL RESULTS AND CONCLUSIONS

For CLOTH MEASURE to be a viable tool for the prediction of the memory requirements of the

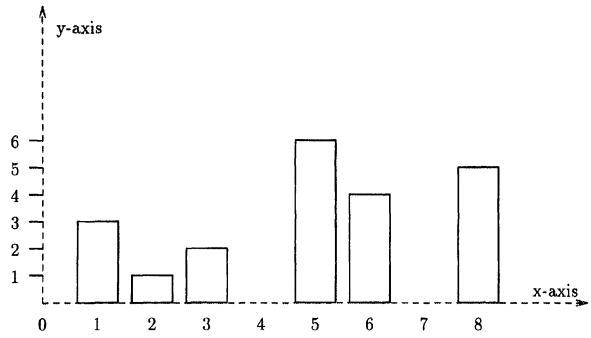


FIGURE 9 Set of rectangles corresponding to the integers $[l_1, l_2, l_3, l_4, l_5, l_6] = [2, 3, 1, 6, 8, 5]$. The violation computation algorithm outputs the ordered pairs: (3, 1), (1, 2), (2, 6), (6, 4), (4, 8).

corner stitching data structure, the following should all be true:

1. *Simplicity*: CLOTH MEASURE should be simpler to implement than the Corner Stitching data structure. (Otherwise it is easier to determine the memory requirements of a layout by implementing corner stitching on that layout). CLOTH MEASURE required approximately 1100 lines of C++ code. Of this, approximately 600 lines of code were required to implement the sorting algorithm (mergesort) and the balanced search tree (2–3–4 tree). Since these would typically be available in a C++ class library, only about 500 lines of code are required. In contrast, Rectangular Corner Stitching required 3000 lines of C code, while L-shaped Corner Stitching required approximately 8500 lines of C code.
2. *Accuracy*: CLOTH MEASURE computes exactly the memory requirements of the RCS data strcuture, from the theoretical results of [1]. CLOTH MEASURE computes a range of values for the memory requirements of the LCS data structure. The reason that a range is obtained rather than exact value is due to the property of the *LCS* data structure that the same set of solid tiles can have many representations. The actual *LCS* representation for a given set of solid tiles is determined by factors such as the order in which tiles are inserted into the data structures. Different representations of the same set of tiles have different memory requirements. Hence the inability of our estimation process to produce an exact value for the memory requirements of a layout is a consequence of the representation method (and not of the estimation process).

   Columns 6, 7, and 8 of Table II suggest that the actual memory requirements of layouts in *LCS* lie nearer the best case prediction than the worst case prediction. (Data for the actual memory requirements of layouts using LCS were obtained from [3]. This was done by inserting additional statements in the LCS code for the insertion and deletion operations that kept a global count of tiles currently in the data structure).
3. *Computation time*: CLOTH MEASURE should execute faster for a given set of solid tiles than the corner stitching algorithm for inserting rectangles into the corner stitching data structure. (Otherwise, it is quicker to use corner stitching than CLOTH MEASURE if both are available). Columns 2–5 of Table II compare the insertion times for Magic [4], RCS and LCS with the execution time for CLOTH MEASURE. (Data for the run times for LCS, Magic, and RCS are obtained from [3]). Data for run times for CLOTH MEASURE were obtained by running it for the same layout on the same computer – a SUN SPARCstation 10. The table shows that CLOTH MEASURE is significantly faster.
4. *Memory requirements*: It is also desirable that the memory requirements of CLOTH MEASURE be less than those of *LCS* and *RCS*. (Note that our table does not include the memory requirements for *RCS* as these are greater than those for *LCS*.) The last column of Table II shows the maximum memory required by CLOTH MEASURE. Our algorithm for computing vertical violations requires memory to store two arrays consisting of $2n$ sides each (one contains vertical sides, the other all vertical

TABLE II Experimental comparison between LCS/RCS/Magic and CLOTH MEASURE (CM)

| No. of Tiles | Execution times | | | | LCS Memory Reqmts (KB) | | | WC CM Memory Reqmts (KB) |
|---|---|---|---|---|---|---|---|---|
| | RCS | Magic | LCS | CM | actual | best case | worst case | |
| 5001 | 4.50 | 3.18 | 8.86 | 2.14 | 349.1 | 324.9 | 404.2 | 260.1 |
| 10000 | 7.61 | 6.12 | 18.20 | 5.14 | 700.5 | 651.4 | 810.6 | 520.0 |
| 15000 | 11.36 | 9.30 | 27.86 | 8.37 | 1058.4 | 983.8 | 1224.6 | 780.0 |
| 20000 | 16.63 | 12.73 | 38.10 | 11.71 | 1421.3 | 1320.2 | 1644.1 | 1040.0 |
| 25000 | 21.24 | 16.59 | 49.23 | 15.28 | 1782.1 | 1655.1 | 2061.8 | 1300.0 |
| 30000 | 26.26 | 20.33 | 59.70 | 19.41 | 2154.5 | 2000.1 | 2492.4 | 1560.0 |
| 35000 | 31.98 | 25.45 | 70.27 | 23.67 | 2531.4 | 2347.9 | 2926.8 | 1820.0 |
| 40000 | 38.52 | 29.74 | 83.40 | 27.72 | 2905.6 | 2692.3 | 3356.9 | 2080.0 |
| 45000 | 45.72 | 35.37 | 97.23 | 32.11 | 3296.6 | 3053.4 | 3808.9 | 2340.0 |
| 50000 | 48.24 | 41.32 | 109.78 | 35.46 | 3688.8 | 3416.3 | 4263.3 | 2600.0 |
| 55001 | 59.10 | 47.90 | 121.89 | 40.09 | 4074.3 | 3771.1 | 4707.2 | 2860.1 |
| 60000 | 69.25 | 54.45 | 138.49 | 43.95 | 4470.4 | 4138.1 | 5167.5 | 3120.0 |

common sides). Assuming that 3 integers and one char are used to store a side, where an integer requires 4 bytes and a character requires 1 byte, this quantity $= 52n$ bytes[1]. Algorithm *ComputeHorizViols* requires memory to store one array consisting of $2n$ sides and a search tree which contains a maximum of $n$ sides. Assume that each tree node requires one integer (to store the index of the horizontal side in the array), two pointers (left and right children), and one character associated with each pointer (to determine whether a horizontal side belongs to the same chain as its predecessor in the array). This represents the worst case scenario if the tree is a 2–3–4 tree. Assuming that a pointer requires 4 bytes, *ComputeHorizViols* requires $40n$ bytes. The total space required by the algorithm is the maximum of the space required by the two algorithms (because the memory occupied by each is released when it terminates). The table shows that CLOTH MEASURE requires less memory. Note that the quantity tabulated represents the worst case memory requirements. The actual memory requirements are expected to be less because the number of common sides will typically be less than the number of vertical sides.

We have demonstrated that CLOTH MEASURE is an effective tool for measuring the memory requirements of the *RCS* and *LCS* data structures. We expect that the techniques of this paper can be extended to the trapezoidal and curved versions of the Corner Stitching data structure described in [9] and [10].

## *References*


[1] Dinesh P. Mehta (1994). "Estimating the storage requirements for rectangular and L-shaped corner stitching data structures", in *Proceedings of the Fourth Great Lakes Symposium on VLSI*, pp. 34–37.

[2] John K. Ousterhout (1984). "Corner Stitching: A data structring technique for VLSI layout tools", *IEEE Transactions on Computer-aided Design*, **3**(1), pp. 87–100.

[3] Blust, G. and Mehta, D. P. (1993). "Corner Stitching for L-Shaped Tiles", in *Proceedings of the Third Great Lakes Symposium on VLSI*, pp. 67–68.

[4] Ousterhout, J., Hamachi, G., Mayo, R., Scott, W. and Taylor, G. (1984). "Magic: A VLSI layout system", in *Proc. of 21st Design Automation Conf.*, pp. 152–159.

[5] John, K. Ousterhout (1981). "Caesar: An interactive editor for VLSI", *VLSI Design*, **II**(4), pp. 34–38.

[6] Bentley, J. L. and Friedman, J. H. (1979). "A survey of algorithms and data structures for range searching", *ACM Computing Surveys*, **11**(4).

[7] Kedem, G. (1982). "The quad-CIF tree: A data structure for hierarchical on-line algorithms", in *Proceedings of the 19th Design Automation Conference*, pp. 352–357.

[8] Hsueh, M. Y. (1979). "Symbolic layout and compaction of integrated circuits", Tech. Rep. UCB/ERL/M79/80, University of California, Berkeley.


[1]Our assumptions about the memory requirements of integers, characters, and pointers are consistent with those made in computing the memory requirements of the two corner stitching data structures.

[9] David Marple, Michiel Smulders and Henk Hegen (1990). "Tailor: A Layout System Based on Trapezoidal Corner Stitching", *IEEE Transactions on Computer-aided Design*, **9**(1), pp. 66–90.
[10] Carlo, H. Séquin and Heloisa da Silva Façanha (1993). "Corner Stitched Tiles with Curved Boundaries", *IEEE Transactions on Computer-aided Design*, **12**(1), pp. 47–58.

## Authors' Biography

**Dinesh Mehta** received the B.Tech., degree in Computer Science and Engineering from the Indian Institute of Technology at Bombay in 1987, the M.S., degree in Computer Science from the University of Minnesota in 1990 and the Ph.D., degree in Computer Science from the University of Florida in 1992. Since then, he has been an Assistant Professor of Computer Science at the University of Tennessee Space Institute. His research interests are in pattern discovery, VLSI CAD, and parallel computing.

Dr. Mehta has co-authored the text *Fundamentals of Data Structures in C++*with Ellis Horowitz and Sartaj Sahni.