Nearly Balanced Quad List Quad Tree - A Data Structure for VLSI Layout Systems*

PEI-YUNG HSIAO
Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan 300, R.O.C., Phone: 886-35-712121ext56630, FAX: 886-35-721490, E-mail: pyhsiao@cis.nctu.edu.tw

(Received October 4, 1993, Revised December 24, 1994)

In the past ten years, many researchers have focused attention on developing better data structures for storing graphical information. Among the proposed data structures, the quad tree data structure provides a good way to organize objects on a 2-D plane. Region searches proceed at logarithmic speeds a desirable characteristic, but no previously proposed VLSI quad tree data structure distributed objects to subdivide the spatial area. This has been a major drawback for operations such as tree searching and window query. In this paper, we present a new division method to reconstruct those quad trees including the multiple storage quad tree (MSQT) and the quad list quad tree (QLQT) into nearly balanced quad tree data structures. Nearly balanced quad trees based on our new spatial division method are constructed by dynamically translating unbalanced multiple storage quad trees or unbalanced quad list quad trees into balanced structures. All benefits of the original quad tree data structures are completely retained. In addition, this method is simple and balanced quad trees memory require less than the original quad trees. Experimental results illustrate that the improvement in region queries of the presented nearly balanced quad trees to both of the QLQT and the MSQT is better than the improvement of the QLQT to the MSQT.

Key Words: VLSI Layout Systems, Region Query, Quad Trees, Corner Stitching, Spatial Data Structure

1 INTRODUCTION

In interactive CAD applications, a suitable data structure for the storage of graphical information is the most important factor in determining the overall performance of the application. We have to take the speed of key operations, such as windowing, tree traversal, and neighborhood searches, as well as the available memory resources into consideration. Thus, many storage methods ranging from simple linear lists to more sophisticated data structures, such as corner-stitching [1], large k-d tree families [2]-[6], and quad trees [7]-[13], have been introduced to enhance the application of VLSI-CAD tools [14]-[20].

In 1974, Finkel and Bentley proposed a quad tree data structure for organizing points on a plane so that one can quickly find the points in any specified query region [8]. Brown's multiple storage quad tree (MSQT) [11], was proposed in 1986; in 1989, a more efficient data structure, the quad list quad tree (QLQT) [12], was introduced. Both of these quad tree data structures are constructed by regularly dividing a quadrant recursively into four subquadrants of equal area. Division procedures used in the MSQT and QLQT do not consider the distribution of objects in the 2D plane. The QLQT improves the MSQT in the leaf quadrants by transforming its single long list of object reference nodes into four exclusive lists[12].

The QLQT features single-access to any object while region query is running. In other words, it does not need the marking and unmarking operations that multiple-storage quad trees use for any visited object. As a result, their experimental results (only about 7000 objects in the retriev...
showed that the speed of region queries for large windows in QLQT is 2 to 5 times faster than those in the MSQT. However, it is not actually necessary to unmark each MSQT object description node’s flag for every region query. The marking and unmarking of visited objects proposed by Brown merely avoids reporting any object more than once while querying in the MSQT. Therefore, we have modified Brown’s method to speed up region queries in the MSQT by the small improvement of discarding the unmarking operation and replacing local flags with global ones. Our experimental results, including those from three kinds of problem size, 1000, 10000 and 70000 objects in the layout plane, show that improving MSQT marking and unmarking operations did indeed speed up the region query function.

However, when we examine the division method, as shown in Figure 1, we see that both types of tree have unbalanced structures. This shortcoming significantly reduces search and window query efficiency, especially in the worst cases. Generally speaking, quad tree data structures must be balanced if the speed of the region search is to be increased. VLSI layout tools based on balanced data structures will certainly benefit from this advantage since they perform a lot of window query operations.

It should be mentioned here that Su et al. [29] and De Pauw [30] had proposed ways of improving the performance of MSQT and QLQT. Su et al. [29] suggested an adaptive threshold number and a new quad subdivision criterion to take care of nonuniform distributions of object positions and object sizes. De Pauw [30] proposed a multtree with internal storage (MTIS) structure which basically stores the object reference pointers at the internal nodes rather than the leaf nodes of the tree structure. By moving the object reference pointers nearer to the tree root, this method speeds up the region search operation. Nevertheless, neither of these two data structures considers tree balance.

In the practical VLSI tool design applications, the number of tree-building operation is much smaller than the number of region queries (the ratio may be 1 to 100 or 10 to 1000). The overhead of reconstructing the original quad trees into nearly balanced quad trees can therefore be ignored, especially since the proposed binary division method illustrated in Sec. 3 is so simple.

This paper presents two nearly balanced quad trees called the binary balanced multiple storage quad tree (BBMSQT) and binary balanced quad list quad tree (BBQLQT). According to our experience, the BBMSQT and the BBQLQT are absolutely balanced for all general cases of layout distribution except for the worst cases discussed in Sec.5. Even for the worst cases, our nearly balanced quad trees still preserve the advantages of requiring less memory and conducting region query operations quickly.

2 QUAD TREES FOR LAYOUT REPRESENTATIONS

2.1 The Basic Quad Trees

A quad tree, as shown in Figure 1, is constructed by repeatedly dividing all quadrants containing more than \( T_N \) objects (\( T_N \) is a threshold number introduced by Brown [11]) into four subquads. This is done by associating a tree node with each quadrant and drawing four links from each tree node to its four child nodes [8][11]. \( T_N \) thus varies according to the total number of objects.

2.2 Bisector List Quad Trees (BLQT)

In Kedem’s bisector list quad trees [9][11], each non-leaf quadrant is associated with a pair of horizontal and vertical bisector lists. Each object that is not entirely contained in a single subquadrant of a non-leaf quadrant is put in either its horizontal or vertical bisector list. Window-finding operations can be very inefficient when using the bisector list quad tree data structure especially when the window is small and the mask is large. This is because the window-finding operations have to examine objects organized into linear lists. This linear list search is complicated and greatly reduces the efficiency of the window-finding operations.

2.3 Multiple Storage Quad Trees (MSQT)

In 1986, Brown [11] proposed another way of organizing objects that intersect one or more than one quadrant boundary. Such objects are stored in every leaf quadrant they intersect, which means that some objects will be stored in more than one quadrant. This approach wastes some space storing duplicated pointers, since some objects are referenced in more than one quadrant. In order to avoid reporting objects more than once, other authors presented further improvements [12][14]. Objects are marked the first time they are reported, and once marked, are not reported again. This data structure requires marking and unmarking operations to maintain its validity whenever the objects are sought.

2.4 Quad List Quad Trees (QLQT)

In response to the problem raised by multiple-object enumeration in an MSQT, the quad list quad tree data
structure was presented in 1989 [12]. In a QLQT, the single long linked list of object reference nodes is made into four distinct shorter lists. If any object intersects the leaf quadrant, a reference to this object will be included in one of the four lists according to the relative position of the object with respect to the leaf quadrant it intersects. The assignment procedure [12] is illustrated in Figure 2 and Table 1, where four different lists (0 to 3) are associated with each leaf node. If the object's smallest enclosing rectangle overlaps the subregion (Figure 1) corresponding to this leaf node, a reference to this object will be included in one of the four lists depending on its relative position with respect to the subregions shown in Table 1. When objects are searched for or found...
in the QLQT data structure, each object will be accessed only once from one of the four distinct lists instead of from the longer list in the MSQT. Even so, both the MSQT and the QLQT form skewed (heavily unbalanced) trees and require a linear time but not in $O(\log N)$ to perform search operations, where $N$ is the number of objects in the layout plane.

3 THE NEW BALANCED QUAD TREES (BQT)

3.1 The Original Division Method

In Brown’s MSQT [11] and Weyten’s QLQT [12], the division method used to construct the entire quad tree data structure involves simply splitting each quadrant that contains more than $T_N$ objects into four smaller subquadrants of equal size, as shown in Figure 3. Each parent quadrant is represented by a tree node that points to the four subquadrants. This process recursively subdivides any of new subquadrants that contain more than $T_N$ objects.

This division method is suitable for a uniformly distributed layout, but unfortunately layout objects in the real world are usually not distributed in a completely uniform manner. Because, in most physical layout systems, the system global layout plane, say from, $(-2^{31}, -2^{31})$ to $(+2^{31}, +2^{31})$ is usually much larger than the realistic mask layout. Moreover, in the worst cases, these data structures lead to a linear time complexity rather than the expected time order of $O(\log N)$ for the tree search operations ($N$ is the total number of objects in the layout plane).

3.2 The New Division Method to Guarantee a Nearly Balanced Tree

In our approach, we improve the division method by taking the distribution of layout objects into consideration so that each of the four subquadrants contains about the same number of objects. Hence the constructed quad tree data structure will be a nearly balanced tree except for the worst cases discussed in Sec. 5.

In this method, the quadrant to be split is divided by two orthogonal dividing lines. However, each of the resulting four subquadrants may have a different area. In other words, we manipulate the final position of the cross point of the two orthogonal dividing lines to give the four subquadrants about the same number of objects. This is illustrated in Figure 4.

In adjusting the position of the cross point, we can separate the whole division procedure into two independent subprocedures, which means we have to locate the vertical line and horizontal line individually in order to locate the new cross point, as illustrated in Figure 5. Often this division method does not produce a fully balanced quad tree which guarantees the order of tree height in $\log_2 N$. However, in the worst case, the quad trees produced by our division method will guarantee the tree height limited at $\log_2 N$. The independent consideration in each orthogonal direction to locate the two

<table>
<thead>
<tr>
<th>rectangle crossing lower boundary</th>
<th>rectangle crossing left boundary</th>
<th>list type</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>0</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>2</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>3</td>
</tr>
</tbody>
</table>
adjust the lines in both the vertical and horizontal directions to obtain the near balanced position repetitively. We refer to the original unbalanced quad tree (the MSQT or the QLQT) and repeatedly examine the differences between the number of objects contained in each candidate subregion to determine the new candidate dividing position; this process continues until an exact line location is determined. In counting the number of objects, we use the region query function from the original quad tree.

This division process recursively subdivides any new subquads containing more than $T_v$ objects. Finally, the balanced quad tree is constructed by translating the original one. Whenever the quad tree becomes somewhat unbalanced, which frequently occurs after a sequence of insertions or deletions, the tree will be dynamically rebalanced by the above procedures. Figure 7 shows balanced quad tree constructed by translating the original quad tree shown in Figure 1 using the proposed division method.

4 THE TRANSLATING ALGORITHMS

In this section we introduce the detailed algorithms for translating the MSQT into a balanced quad tree. These algorithms can easily be modified to fit any quad tree data structure that employs the region query function.
The following procedures written in C pseudo code are to translate the original MSQT into a balanced MSQT (BMSQT).

The function Translate() initiates balancing of the original quad tree. Balanced_split() coordinates the splitting of subregions that contain more than \( T_N \) objects. Generate_subquad() is the function that makes use of the Cut_region_vertically() and Cut_region_horizontally() procedures to determine the positions of the vertical and horizontal division lines.

Translate(QT_Root)

BEGIN
define the whole “BQT_Root” boundary to be a “Window”;

Object_reference_list = Region_query(Window, QT_Root);
{The “Object_reference_list” points to a list of object reference nodes. The “Region_query()” is a fundamental function of the MSQT[11] used to find those objects intersected by the specified “Window”.)

Unmark(Object_reference_list); {Unmark the reported objects.}
Balanced_split(BQT_Root, Object_reference_list);
{Balanced_split() is the main procedure to construct a BMS quad tree. It is executed recursively.}
END {end of Translate()}

Balanced_split(BQT_node, Object_reference_list)

BEGIN
int i;
if (the region size of “BQT_node” is not splittable) return;

FIGURE 6 Using the binary search method to obtain the correct position (2) from the initial position (1) for the vertical dividing line. The last position is located so that the dividing line will divide the specified region into two subregions containing the same number of objects.

FIGURE 7 The data structure of the BMSQT for the same mask layout as shown in Figure 1(a).
{Avoid indefinitely splitting quadrants that contain group of overlapping nested objects.}

Generate_subquad(BQT_node);
{Generate four subquadrants pointed to by the current quadrant, “BQT_node”.}

while (Object_reference_list != NULL)
DO

for (i = 0; i < 4; i++)
BEGIN
if (Object_reference_list->object intersects BQT_node->subquad[i])
BEGIN
Add this specified object to the object reference list of “BQT_node->subquad[i]”;
(BQT_node->subquad[i])->use ++;
END {end of if}
END {end of for}

Delete_node = Object_reference_list;
{The Delete_node is a pointer pointing to the node to be deleted.}
Object_reference_list = Object_reference_list->next_node;
Free(Delete_node); {Free and reallocate memory}
END {end of while}

for (i = 0; i < 4; i++)
if (BQT_node->subquad[i]->use > Try)
BEGIN
BQT_node->subquad[i]->use 1;
Object_reference_list = BQT_node->subquad[i]->object_reference_list;
BQT_node->subquad[i]->object_reference_list = NULL;
Balanced_split(BQT_node->subquad[i], Object_reference_list);
{recursion}
END {end of for}
END {end of Balanced_split()}

Generate_subquad(BQT_node)
{Generate four subquadrants pointed to by the current quadrant, “BQT_node”.}
BEGIN
int X, Y, i, j;

X = Cut_region_horizontally(the region defined by “BQT_node”);
{Apply the Cut_region_horizontally() function to determine the horizontal dividing line, Y = the coordinate returned by the Cut_region_horizontally() function.}

for (i = 0; i < 4; i++)
BEGIN

(avoid indefinitely splitting quadrants that contain group of overlapping nested objects.)

Generate_subquad(BQT_node);
{Generate four subquadrants pointed to by the current quadrant, “BQT_node”.}

while (Object_reference_list != NULL)
DO

for (i = 0; i < 4; i++)
BEGIN
if (Object_reference_list->object intersects BQT_node->subquad[i])
BEGIN
Add this specified object to the object reference list of “BQT_node->subquad[i]”;
(BQT_node->subquad[i])->use ++;
END {end of if}
END {end of for}

Delete_node = Object_reference_list;
{The Delete_node is a pointer pointing to the node to be deleted.}
Object_reference_list = Object_reference_list->next_node;
Free(Delete_node); {Free and reallocate memory}
END {end of while}

for (i = 0; i < 4; i++)
if (BQT_node->subquad[i]->use > Try)
BEGIN
BQT_node->subquad[i]->use 1;
Object_reference_list = BQT_node->subquad[i]->object_reference_list;
BQT_node->subquad[i]->object_reference_list = NULL;
Balanced_split(BQT_node->subquad[i], Object_reference_list);
{recursion}
END {end of for}
END {end of Balanced_split()}

Generate_subquad(BQT_node)
{Generate four subquadrants pointed to by the current quadrant, “BQT_node”.}
BEGIN
int X, Y, i, j;

X = Cut_region_horizontally(the region defined by “BQT_node”);
{Apply the Cut_region_horizontally() function to determine the vertical dividing line, X = the coordinate returned by the Cut_region_horizontally() function.}

Y = Cut_region_horizontally(the region defined by “BQT_node”);
{Apply the Cut_region_horizontally() function to determine the horizontal dividing line, Y = the coordinate returned by the Cut_region_horizontally() function.}

for (i = 0; i < 4; i++)
BEGIN

Allocate a location for “BQT_node->subquad[i]”;
BQT_node->subquad[i]->use = 0;
BQT_node->subquad[i]->object_reference_list = NULL;
for (j = 0; j < 4; j++) BQT_node->subquad[i]->subquad[j] = NULL;
Define the “BQT_node->subquad[i]” boundary;
{The assigning work is shown in Figure 8.}
END {end of for}
END {end of Generate_subquad()}

Cut_region_vertically(region)
{This function determines the vertical dividing line in the specified region according to whether the subregions contain approximately the same number of objects.}
BEGIN
int Lower_bound, Upper_bound, Middle,
Count1, Count2;
BOOL Balanced;

Lower_bound = left boundary of “region”;
Upper_bound = right boundary of “region”;

FIGURE 8 The numbered rectangular regions stand for the four subquadrants whose areas are defined by the two dividing lines and the boundaries of the specified quadrant.
Balanced = FALSE;
{Use the binary search method to obtain where the dividing line should be placed.}

**REPEAT**
Middle = Lower_bound + (Upper_bound - Lower_bound + 1) / 2;

Count1 = Region_objects(the left subregion, QT_Root);
Count2 = Region_objects(the right subregion, QT_Root);
(The Region_objects() issues a region query[12] to the MS quad tree pointed to by “QT_Root”. With these two subregions generated by the dividing line, X = Middle, we will get the number of objects contained in each subregion. Here, Count1 is the number of objects contained in the left subregion and Count2 is the number of objects contained in the right subregion.)

if (Count1 == Count2) Balanced = TRUE;
else if (Count1 > Count2) Upper_bound = Middle;
{Adjust and shift the dividing line to the left.}
else if (Count1 < Count2) Lower_bound = Middle;
{Adjust and shift the dividing line to the right.}
if ((Upper_bound — Lower_bound) < 2) balanced = TRUE;
(This conditional statement acknowledge that we can not always divide the specified region into two subregions containing exactly the same number of objects in the vertical direction. Sometimes, the number of objects will differ by one.)
**UNTIL** (Balanced == TRUE);

return (Middle);
**END** {end of Cut_region_vertically()}

Cut_region_horizontally(region)
{Similar to Cut_region_vertically(), this function determines placement of the horizontal dividing line in a specified region. Please refer to Cut_region_vertically() for details.}

5 COMPLEXITY ANALYSIS
5.1 Descriptive Algorithms with Complexity Analysis

Before describing the generic algorithm for translating the Original Quad Tree (OQT) into our Balanced Quad Tree (BQT), we first present the algorithm, Cut-region-vertically(), for generating vertical(horizontal) division lines. It partitions any given rectangular region containing overlapping or non-overlapping layout rectangles into regionLeft and regionRight, such that each contains exactly or approximately the same number of layout rectangles as the other.

Cut-region-vertically(OQT, region)
{
Step 1
Heuristic binary search for generating regionLeft and regionRight such that
\[ |L_l - L_r| = |R_d| \]
where
\[ R_d = \lceil T_N/10 \rceil + 3 ; T_N = \text{threshold value of OQT} \]
L_l = Query(OQT, regionLeft);
L_r = Query(OQT, regionRight);
and region = vertically disjoint union of regionLeft and regionRight;
Step 2
vertical-division-line = vertical boundary line between regionLeft and regionRight;
Step 3
Return (vertical-division-line);
}

Note that the time complexity for the heuristic binary search shown above depends on the horizontal length (or vertical width) of the region, the distribution of layout rectangles, and our heuristics. Without heuristics and considering the layout distribution, this search needs a time of O(log M), where M is the horizontal length (or vertical width) of the region. M is independent of N, the total number of layout rectangles. With our heuristics in considering the distribution of a given layout, this search can be done in constant time. As a result, the time complexity of Cut-region-vertically() is only influenced by the time complexity of Region Query in OQT. For the worst case, the OQT may be a skewed tree. Therefore, the time complexity of Cut-region-vertically() as well as that of Region Query in OQT is O(N)

Generally, the translation algorithm, Translate(), from OQT to BQT recursively splits each partitioned region into four balanced subregions, region1, region2, region3 and region4, according to the vertical-division-line and horizontal-division-line obtained from Cut-region-vertically() and Cut-region-horizontally(). These four subregions should contain nearly equal numbers of rectangles. During the split operation, BOT tree nodes are built from OQT by referring to the calculated vertical-division-line and horizontal-division-line. For those regions requiring no further splitting, the corresponding object reference nodes in OQT are transferred to constructing BQT one by one. BOT’s tree-building operation as illustrated below, is separately and recursively completed by the
functions of Build-tree-node-of-BQT() and Build-object-reference-node-of-BQT().

**Translate(OQT)**

```
{ 
region = Find-whole-layout-region(OQT); 
BQT = Balanced-split(OQT, region); 
Return = BQT; }
```

**Balanced-split(OQT, region)**

```
{ 
  dividing = Check-whether-further-division(OQT, region); 
if (dividing == YES) 
{ 
  vertical-division-line = Cut-region-vertically(OQT, region); 
  horizontal-division-line = Cut-region-horizontally(OQT, region); 
  BQT = Build-tree-node-of-BQT(BQT, vertical-division-line, horizontal-division-line); 
  region1, region2, region3, region4 = Generate-subquad(region, vertical-division-line, horizontal-division-line); 
  Balanced-split(OQT, region1); 
  Balanced-split(OQT, region2); 
  Balanced-split(OQT, region3); 
  Balanced-split(OQT, region4); 
} else /*dividing = NO */ 
{ 
  BQT = Build-object-reference-node-of-BQT(OQT, region); 
} 
Return(BQT); }
```

In the **Translate()** algorithm above, the final time complexity depends on **Find-whole-layout-region()** and **Balanced-split()**. The **Find-whole-layout-region()** function can easily be done by **Region Query** in OQT, and its time complexity should be in O(N), for the same reason discussed above.

In viewing the algorithm of **Balanced-split()**, because that the functions of Build-tree-node-of-BQT() and Generate-subquad() can be completed by quadrant operation in constant time, hence actually, the time complexity of Balanced-split() should be determined by Build-object-reference-node-of-BQT(), Cut-region-vertically(horizontally), and the recursive operations of Balanced-split() itself. Here, we have already known that the time complexity of the function Cut-region-vertically(horizontally)() is in O(N). In addition, the time complexity of Build-object-reference-node-of-BQT() is linearly dependent of the number of object reference nodes for each partitioned region, which usually is much less than N. From this, we obtain that the time complexity of Balanced-split() should be in O(N * no. of recursive operations). Due to the contribution of balanced partition for each Cut-region-vertically(horizontally)(), the final tree height of BQT as well as the number of recursive Balanced-split() operations is in O(log N). Consequently, the time complexity of Balanced-split() is in O(N log N).

This concludes that the time complexity of our proposed translating algorithm from OQT to BQT, Translate(), is in O(N log N). The reconstructed balanced quad tree, BQT, guarantees the time complexity of **Region Query** in O(log N) which is much better than the original time complexity of **Region Query** in O(N) for OQT.

### 5.2 Worst Case for Tree Height Analysis

In the following analysis, we will show that the height of the proposed balanced quad tree is limited to log₂N, where N is the total number of objects in the plane. In the construction of an MSQT or QLQT, any quadrant containing more than Tₖ objects is recursively split into four subquadrants. Consider the root quadrant and four subquadrants shown in Figure 9, and let a, b, c and d be the number of objects in the subquadrants. The dividing line in either the vertical or horizontal direction divides the entire region into two parts, each of which has an equal number of objects. Therefore, we have

\[
a + b = c + d \quad \text{(in the vertical division),} \quad (4.1)
\]

and

![Figure 9](image)

**FIGURE 9** The variables a, b, c, and d representing the number of objects in the corresponding subquadrants.
a + d = b + c (in the horizontal division). (4.2)

By combining these two equations, we find that a is equal to c and that b is equal to d. For the worst case, shown in Figure 10,

\[ a = c = 0 \text{ and } b = d = \frac{N}{2}. \] (4.3)

(or \( a = c = \frac{N}{2} \) and \( b = d = 0 \)). (4.4)

Next, the quadrants with \( \frac{N}{2} \) objects are recursively subdivided. The worst case may occur in every recursive subdivision as the example shown in Figure 11. However, in the best case, the two orthogonal divisions will divide the entire region into four parts each containing exactly the same number of objects, which implies the following equation:

\[ a = b = c = d = \frac{N}{4}. \] (4.5)

And every subquadrant, in the best case, will always be as described above. We then can conclude that the height of the balanced MSQT is limited to \( \log_2 N \), and not \( \log_4 N \). This is still preferable to the original MSQT, which can become skewed and suffer a linear time complexity in the worst case.

6 EXPERIMENTAL RESULTS

Region query is a basic operation needed for VLSI-CAD tool design. Therefore, region query speed is a valid measure for evaluating the performance of VLSI-CAD application data structure. To validate our proposed data structures, we conducted a series of experiments. The proposed balanced quad trees, the BMSQT and the BQLQT, were compared with the original MSQT and the QLQT using layouts generated by a pseudo random number generator and real circuits. Our program was written in C language and run on a SUN SPARC station SLC.

To compare the performance of the balanced quad trees with the original quad trees, we generated a large number of random windows in size ranging from \( 0 \times 0 \) (a point window) to \( 1 \times 1 \) (a window covering the whole layout), as shown in Table 2. We then used them to query the four quad trees.

The real execution time was measured in seconds for region queries. The following terms are defined to facilitate presentation of the experimental results.

\[ h_1(h_2): \] the average height for all the leaf quadrants in the original quad tree (the balanced quad tree).

<table>
<thead>
<tr>
<th>window size</th>
<th>amount of queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 \times 1</td>
<td>1</td>
</tr>
<tr>
<td>2/3 \times 2/3</td>
<td>10</td>
</tr>
<tr>
<td>1/2 \times 1/2</td>
<td>10</td>
</tr>
<tr>
<td>1/4 \times 1/4</td>
<td>100</td>
</tr>
<tr>
<td>1/10 \times 1/10</td>
<td>200</td>
</tr>
<tr>
<td>1/40 \times 1/40</td>
<td>400</td>
</tr>
<tr>
<td>0 \times 0</td>
<td>500</td>
</tr>
</tbody>
</table>

FIGURE 10 The worst case implying that \( a = c = 0 \) and \( b = d = \frac{N}{2} \).
tl(t2): the average execution time for region queries in the original quad tree (the balanced quad tree).

\[ h\% = \frac{(h_1 - h_2)}{h_1} \times 100\% \]

\[ a\% = \frac{(t_1 (\text{MSQT}) - t_1 (\text{QLQT}))}{t_1 (\text{MSQT})} \times 100\% \]

\[ b\% = \frac{(t_1 (\text{MSQT}) - t_2 (\text{BMSQT}))}{t_1 (\text{MSQT})} \times 100\% \]

\[ g\% = \frac{(t_1 (\text{MSQT}) - t_2 (\text{BQLQT}))}{t_1 (\text{MSQT})} \times 100\% \]

Experiments for random layouts with 1000, 10000 and 70000 objects for the MSQT, the QLQT, the BMSQT, and the BQLQT are tested and the results were shown in Tables 3, 4, 5, 6, 7, 8, 9, 10 and 11. These experimental results confirm that both the search speed and the space requirements of the balanced quad trees are superior to those of the original quad trees, because the height of a balanced quad tree is always shorter than that of the original quad tree.

Furthermore, by observing the experimental results presented by Weyten & Pauw[12] (and shown in Table 12) for the comparison of region queries in the MSQT and the QLQT, we can conclude that those excellent results were obtained that the time exhaustive method of the marking and unmarking operations in Brown’s paper, which was out of date[24][14], was compared in their experiment. In fact, the marking and unmarking operations used in the original MSQT can be improved upon by increasing the values of the flags in the object description nodes associated with each visit instead of maintaining validity by marking them and unmarking them before the next query. It is then not necessary to unmark the visited objects for every region query operation. Hence, our results obtained from using the new

---

**TABLE 3**

Memory Requirements for the Data Structures of the Four Distinct Quad Trees Based on the Same Layout of 1000 Objects.

<table>
<thead>
<tr>
<th>TREE TYPE</th>
<th>Tree nodes</th>
<th>Object ref. nodes</th>
<th>Object nodes</th>
<th>Space Total (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSQT</td>
<td>721</td>
<td>133</td>
<td>1300</td>
<td>46968</td>
</tr>
<tr>
<td>QLQT</td>
<td>721</td>
<td>133</td>
<td>1300</td>
<td>46968</td>
</tr>
<tr>
<td>BMSQT</td>
<td>537</td>
<td>105</td>
<td>1123</td>
<td>45588</td>
</tr>
<tr>
<td>BQLQT</td>
<td>537</td>
<td>105</td>
<td>1123</td>
<td>45588</td>
</tr>
</tbody>
</table>

**TABLE 4**

Comparison of Tree Heights for the Original and the Balanced Quad Tree Based on the Same Layout of 1000 Objects.

<table>
<thead>
<tr>
<th>Tn</th>
<th>Original quad trees (MSQT or QLQT)</th>
<th>Balanced quad trees (BMSQT or BQLQT)</th>
<th>h%</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4.741220</td>
<td>4.486352</td>
<td>5.4%</td>
</tr>
<tr>
<td>10</td>
<td>4.007905</td>
<td>3.987854</td>
<td>0.5%</td>
</tr>
<tr>
<td>20</td>
<td>3.480000</td>
<td>3.253165</td>
<td>6.5%</td>
</tr>
</tbody>
</table>

**TABLE 5**

Memory Requirements for the Data Structures of the Four Distinct Quad Trees Based on the Same Layout of 10000 Objects.

<table>
<thead>
<tr>
<th>TREE TYPE</th>
<th>Tree nodes</th>
<th>Object ref. nodes</th>
<th>Object nodes</th>
<th>Space Total (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSQT</td>
<td>3781</td>
<td>1329</td>
<td>11631</td>
<td>438544</td>
</tr>
<tr>
<td>QLQT</td>
<td>3781</td>
<td>1329</td>
<td>11631</td>
<td>438544</td>
</tr>
<tr>
<td>BMSQT</td>
<td>3369</td>
<td>1125</td>
<td>11123</td>
<td>341848</td>
</tr>
<tr>
<td>BQLQT</td>
<td>3369</td>
<td>1125</td>
<td>11123</td>
<td>341848</td>
</tr>
</tbody>
</table>

**TABLE 6**

Comparison of Tree Heights for the Original and the Balanced Quad Tree Based on the Same Layout of 10000 Objects.

<table>
<thead>
<tr>
<th>Tn</th>
<th>Original quad trees (MSQT or QLQT)</th>
<th>Balanced quad trees (BMSQT or BQLQT)</th>
<th>h%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5.856135</td>
<td>5.798971</td>
<td>1.0%</td>
</tr>
<tr>
<td>20</td>
<td>5.034253</td>
<td>5.000000</td>
<td>0.7%</td>
</tr>
<tr>
<td>30</td>
<td>4.990973</td>
<td>4.928910</td>
<td>1.3%</td>
</tr>
</tbody>
</table>
method in marking and unmarking operations do speed up region queries in the MSQT.

Our results show that the improvement is more striking for large windows. Analyzing the value of \( b/h \) we find it grows larger as window size increases, which means larger windows further enhance improvement and make it more apparent to us. On the other hand, because smaller windows cover fewer objects, a small window then visits a small portion of the objects in the entire layout. As a result, less improvement in query operations between the balanced quad tree and the original one is apparent. By the way, the speed of the

TABE 8

Comparison of Tree Heights for the Original and the Balanced Quad Tree Based on the Same Layout of 70000 Objects.

<table>
<thead>
<tr>
<th>TN</th>
<th>Original quad trees (MSQT or QLQT)</th>
<th>Balanced quad trees (BMSQT or BQLQT)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>h1 (average heights for leafquads)</td>
<td>h2 (average heights for leafquads)</td>
</tr>
<tr>
<td>20</td>
<td>6.561413</td>
<td>6.330518</td>
</tr>
<tr>
<td>40</td>
<td>6.000000</td>
<td>5.992553</td>
</tr>
<tr>
<td>60</td>
<td>5.964623</td>
<td>5.881086</td>
</tr>
</tbody>
</table>

TABE 9


(a) TN = 5

<table>
<thead>
<tr>
<th>Window size</th>
<th>MSQT</th>
<th>QLQT</th>
<th>BMSQT</th>
<th>BQLQT</th>
<th>α%</th>
<th>β%</th>
<th>γ%</th>
<th>β%/h%</th>
<th>γ%/h%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 × 1</td>
<td>0.018500</td>
<td>0.017833</td>
<td>0.015067</td>
<td>0.014267</td>
<td>3.6%</td>
<td>18.6%</td>
<td>22.9%</td>
<td>3.452</td>
<td>4.256</td>
</tr>
<tr>
<td>2/3 × 2/3</td>
<td>0.009323</td>
<td>0.009090</td>
<td>0.007493</td>
<td>0.007127</td>
<td>2.5%</td>
<td>19.6%</td>
<td>23.6%</td>
<td>3.651</td>
<td>4.382</td>
</tr>
<tr>
<td>1/2 × 1/2</td>
<td>0.005553</td>
<td>0.005393</td>
<td>0.004647</td>
<td>0.004390</td>
<td>2.9%</td>
<td>16.3%</td>
<td>20.9%</td>
<td>3.037</td>
<td>3.896</td>
</tr>
<tr>
<td>1/4 × 1/4</td>
<td>0.001673</td>
<td>0.001673</td>
<td>0.001393</td>
<td>0.001348</td>
<td>0%</td>
<td>16.8%</td>
<td>19.4%</td>
<td>3.117</td>
<td>3.614</td>
</tr>
<tr>
<td>1/10 × 1/10</td>
<td>0.000537</td>
<td>0.000537</td>
<td>0.000469</td>
<td>0.000463</td>
<td>-3.0%</td>
<td>12.7%</td>
<td>13.8%</td>
<td>2.355</td>
<td>2.563</td>
</tr>
<tr>
<td>1/40 × 1/40</td>
<td>0.000225</td>
<td>0.000226</td>
<td>0.000212</td>
<td>0.000213</td>
<td>-4.9%</td>
<td>6.0%</td>
<td>5.3%</td>
<td>1.124</td>
<td>0.992</td>
</tr>
<tr>
<td>0 × 0</td>
<td>0.000173</td>
<td>0.000178</td>
<td>0.000167</td>
<td>0.000165</td>
<td>-2.9%</td>
<td>3.3%</td>
<td>4.6%</td>
<td>0.614</td>
<td>0.860</td>
</tr>
</tbody>
</table>

(b) TN = 10

<table>
<thead>
<tr>
<th>Window size</th>
<th>MSQT</th>
<th>QLQT</th>
<th>BMSQT</th>
<th>BQLQT</th>
<th>α%</th>
<th>β%</th>
<th>γ%</th>
<th>β%/h%</th>
<th>γ%/h%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 × 1</td>
<td>0.012033</td>
<td>0.011033</td>
<td>0.012000</td>
<td>0.011033</td>
<td>8.3%</td>
<td>9.0%</td>
<td>10.4%</td>
<td>12.470</td>
<td>12.403</td>
</tr>
<tr>
<td>2/3 × 2/3</td>
<td>0.006463</td>
<td>0.005967</td>
<td>0.006190</td>
<td>0.005793</td>
<td>7.7%</td>
<td>12.6%</td>
<td>13.8%</td>
<td>12.611</td>
<td>12.470</td>
</tr>
<tr>
<td>1/2 × 1/2</td>
<td>0.003973</td>
<td>0.003680</td>
<td>0.003730</td>
<td>0.003523</td>
<td>7.4%</td>
<td>9.3%</td>
<td>10.8%</td>
<td>12.470</td>
<td>12.403</td>
</tr>
<tr>
<td>1/4 × 1/4</td>
<td>0.001277</td>
<td>0.001229</td>
<td>0.001197</td>
<td>0.001159</td>
<td>3.8%</td>
<td>6.9%</td>
<td>8.2%</td>
<td>12.470</td>
<td>12.403</td>
</tr>
<tr>
<td>1/10 × 1/10</td>
<td>0.000413</td>
<td>0.000424</td>
<td>0.000387</td>
<td>0.000392</td>
<td>-2.7%</td>
<td>6.7%</td>
<td>8.6%</td>
<td>12.470</td>
<td>12.403</td>
</tr>
<tr>
<td>1/40 × 1/40</td>
<td>0.000199</td>
<td>0.000210</td>
<td>0.000199</td>
<td>0.000203</td>
<td>-5.5%</td>
<td>6.7%</td>
<td>8.0%</td>
<td>12.470</td>
<td>12.403</td>
</tr>
<tr>
<td>0 × 0</td>
<td>0.000160</td>
<td>0.000167</td>
<td>0.000159</td>
<td>0.000160</td>
<td>-4.4%</td>
<td>6.7%</td>
<td>8.0%</td>
<td>12.470</td>
<td>12.403</td>
</tr>
</tbody>
</table>

(c) TN = 20

<table>
<thead>
<tr>
<th>Window size</th>
<th>MSQT</th>
<th>QLQT</th>
<th>BMSQT</th>
<th>BQLQT</th>
<th>α%</th>
<th>β%</th>
<th>γ%</th>
<th>β%/h%</th>
<th>γ%/h%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 × 1</td>
<td>0.009300</td>
<td>0.008033</td>
<td>0.008800</td>
<td>0.007767</td>
<td>13.6%</td>
<td>5.4%</td>
<td>16.5%</td>
<td>0.825</td>
<td>2.529</td>
</tr>
<tr>
<td>2/3 × 2/3</td>
<td>0.005083</td>
<td>0.004553</td>
<td>0.004633</td>
<td>0.004240</td>
<td>10.4%</td>
<td>8.9%</td>
<td>16.6%</td>
<td>1.358</td>
<td>2.544</td>
</tr>
<tr>
<td>1/2 × 1/2</td>
<td>0.002277</td>
<td>0.002293</td>
<td>0.002960</td>
<td>0.002760</td>
<td>9.4%</td>
<td>8.3%</td>
<td>14.5%</td>
<td>1.268</td>
<td>2.220</td>
</tr>
<tr>
<td>1/4 × 1/4</td>
<td>0.001047</td>
<td>0.001010</td>
<td>0.000953</td>
<td>0.000919</td>
<td>3.5%</td>
<td>7.0%</td>
<td>12.2%</td>
<td>1.378</td>
<td>1.876</td>
</tr>
<tr>
<td>1/10 × 1/10</td>
<td>0.000368</td>
<td>0.000384</td>
<td>0.000350</td>
<td>0.000356</td>
<td>-4.3%</td>
<td>5.0%</td>
<td>8.0%</td>
<td>0.768</td>
<td>0.500</td>
</tr>
<tr>
<td>1/40 × 1/40</td>
<td>0.000184</td>
<td>0.000196</td>
<td>0.000178</td>
<td>0.000187</td>
<td>-6.5%</td>
<td>2.9%</td>
<td>1.6%</td>
<td>0.449</td>
<td>-0.250</td>
</tr>
<tr>
<td>0 × 0</td>
<td>0.000148</td>
<td>0.000160</td>
<td>0.000144</td>
<td>0.000153</td>
<td>-8.1%</td>
<td>2.2%</td>
<td>3.4%</td>
<td>0.343</td>
<td>-0.518</td>
</tr>
</tbody>
</table>
TABLE 10

<table>
<thead>
<tr>
<th>Window size</th>
<th>MSQT</th>
<th>QLQT</th>
<th>BMSQT</th>
<th>BQLQT</th>
<th>α%</th>
<th>β%</th>
<th>γ%</th>
<th>β%h/h%</th>
<th>γ%h/h%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 × 1</td>
<td>0.151667</td>
<td>0.136000</td>
<td>0.137000</td>
<td>0.124000</td>
<td>10.3%</td>
<td>9.7%</td>
<td>18.2%</td>
<td>9.907</td>
<td>18.689</td>
</tr>
<tr>
<td>2/3 × 2/3</td>
<td>0.073117</td>
<td>0.065017</td>
<td>0.066100</td>
<td>0.059317</td>
<td>11.1%</td>
<td>9.6%</td>
<td>18.9%</td>
<td>9.831</td>
<td>19.336</td>
</tr>
<tr>
<td>1/2 × 1/2</td>
<td>0.042650</td>
<td>0.038000</td>
<td>0.038100</td>
<td>0.034250</td>
<td>10.9%</td>
<td>10.7%</td>
<td>19.7%</td>
<td>10.929</td>
<td>20.177</td>
</tr>
<tr>
<td>1/4 × 1/4</td>
<td>0.010764</td>
<td>0.009736</td>
<td>0.009687</td>
<td>0.008762</td>
<td>9.6%</td>
<td>10.0%</td>
<td>18.6%</td>
<td>10.247</td>
<td>19.054</td>
</tr>
<tr>
<td>1/10 × 1/10</td>
<td>0.002105</td>
<td>0.001963</td>
<td>0.001935</td>
<td>0.001810</td>
<td>6.7%</td>
<td>8.1%</td>
<td>14.0%</td>
<td>8.275</td>
<td>14.357</td>
</tr>
<tr>
<td>1/40 × 1/40</td>
<td>0.000443</td>
<td>0.000444</td>
<td>0.000431</td>
<td>0.000423</td>
<td>-0.2%</td>
<td>2.7%</td>
<td>4.5%</td>
<td>2.730</td>
<td>4.625</td>
</tr>
<tr>
<td>0 × 0</td>
<td>0.000235</td>
<td>0.000238</td>
<td>0.000235</td>
<td>0.000235</td>
<td>-1.3%</td>
<td>0.1%</td>
<td>0%</td>
<td>0.116</td>
<td>0</td>
</tr>
</tbody>
</table>

QLQT with small windows is worse than that of the MSQT [12][13] because it involves too many extra tests concerning the types of quad lists.

Our results are encouraging, even though these experiments were based on random layouts with a nearly uniform distribution. When uneven layouts were considered, the performance advantage resulting from our method is considerably greater. Table 10 shows that the query performance of BMSQT and BQLQT is better than that of the QLQT and of the MSQT. Our method is a general technique for improving all 2D spatial data structures that use region query operations. From this perspective, the improvement to all of the 2D spatial data structures provided by our technique appears much better than the contribution of the QLQT[12] to the MSQT[11].

Considering the real circuit shown in Figure 12-13, this mask layout uses 1006 rectangles. It takes about 0.4 seconds to build up the original QLQT, and 0.6 seconds to translate the original QLQT into our balanced QLQT. Additional processing, such as compaction, design-rule checking and circuit extraction, usually requires hundreds or thousands of Region Queries but only uses one period of tree building and translation. The proposed translation algorithm and the balanced quad trees themselves were proven to be more worthwhile. Figure 12-13 shows that memory usage was reduced from 64 Kbytes to 46 Kbytes (or about 30%), and the average tree-height was very much improved: from 18.4 to 3.7. Note that from the tree height distribution illustrated in Figure 12, there are 88 leaf nodes with tree height = 4 and 42 leaf nodes with tree height = 3. This means, in this case, the balanced QLQT was proven to be a fully balanced quad tree.

7 CONCLUSIONS
The proposals in his paper contribute to reduction in memory usage and better query efficiency for the proposed balanced quad trees over original quad trees. The key to building the balanced quad trees described here is determining the locations of dividing lines in a specified region; that is, to adjust the cross point to a suitable position, as shown in Figure 4. The time complexity of the presented translation algorithm and that of the original quad tree building algorithm are in the same time order of O(N log N). However, the reconstructed balanced quad trees guarantee the time complexity of Region Query in O(log N), which is much better than the original time complexity of Region Query in O(N) for OQT.
The experimental results shown in Figure 12-13 and Table 9-12 prove that the balanced quad trees improve all 2D spatial data structures that use region query operations even more than the QLQT improves on the MSQT. For a heavily non-uniformly distributed layout, the improvement in performance provided by our method will be especially significant. From the real circuit shown in Figure 12-13, we obtained about 30% reduction in memory usage, and an average tree height reduction from 18.4 to 3.7. Moreover, for most of real circuits, our balanced QLQT was proven to be a fully balanced quad tree as indicated by the tree height distribution on the message panel of Figure 12-13.

This method also preserves all the advantages of the original quad trees. The proposed data structures speed up search and window query operations significantly and hence should be of great value in the design of VLSI layout tools.

In our research work, we have developed a layout editor[25], a complementary DRC[26], a global router with minimum tile partition[27], and two different layout compactors[28] based on the presented BQLQT.

Acknowledgement

This work was supported in part by NSC82-0404-E004-129. In preparing this manuscript, many thanks to Mr. L. D. Jang from National Chiao Tung University of Taiwan, ROC, and Mr. P. W. Shew from National University of Singapore.

References

FIGURE 12 The Balanced QLQT (for real circuit shown in Figure 13) with average tree height = 3.7 and memory usage = 46 Kbytes.

FIGURE 13 Real layout circuit with 1006 rectangle. The Original QLQT with average tree height = 18.4 and memory usage = 64 Kbytes.


Biographies

PEI-YUNG HSIAO received the B.S. degree in Chemical Engineering from Tung Hai University in 1980, and the M.S. and Ph.D. degrees in Electrical Engineering form the National Taiwan University in 1987 and 1990, respectively. Since August 1990, he has been an Associate Professor in the Department of Computer and Information Science at the National Chiao Tung University, Hsinchu, Taiwan, ROC. His main research interests are VLSI-CAD, Neural Network, and Expert System Applications.
Submit your manuscripts at http://www.hindawi.com