Multicell Quad Trees

S. J. Su* and Y. S. Kuo**
National Chiao Tung Univ.* and Academia Sinica** R. O. C.

Abstract

The multicell quad tree is a two-level tree structure for region queries. At the upper level is a multiple storage quad tree (MSQT). At the lower level, each leaf quad of the MSQT is further subdivided into equal-sized cells. Basically, large-window queries examine structures at the coarse-grained leaf quad level while small-window queries examine structures at the fine-grained cell level. With such two-level structures fitting each kind of queries separately, both large-window and small-window queries can achieve high execution speeds.

1. Introduction

Many applications in computer-aided design (CAD) of integrated circuits require region queries of the objects in the 2-dimensional space (2-space). Region queries find all objects intersecting a specified query window. With the advent of VLSI circuits, a layout design may contain tens of thousands of objects to be region-searched. Therefore designing an efficient data structure for region queries is very crucial.

Several types of data structures have been proposed for representing objects in VLSI layout. Among these data structures, the quad tree has received the most attention. Brown [1] developed the multiple storage quad tree (MSQT) by storing some objects in more than one quad. The quad list quad tree proposed by Weyten and De Pauw [4] is especially efficient for large-window queries. Generalization of the quad tree to the multi-branch tree structure has aimed at improving the search speed of small-window queries [2]. The two-layer quad tree [3] stores large objects in tree nodes at higher levels (closer to the root) and small objects at lower levels.

We have designed a data structure called the multicell quad tree for region queries. The multicell quad tree has a two-level structure. At the upper level is a MSQT. At the lower level, each leaf quad of the MSQT is further subdivided into equal-sized cells. Basically, large-window queries examine structures at the coarse-grained leaf quad level while small-window queries examine structures at the fine-grained cell level. The virtue of this two-level structure is the reduction of object examinations in the query algorithm. Since the 2-space is subdivided into coarse-grained quads, duplications of objects (actually object references) in leaf quads are reduced which leads to the reduction of object (object reference) examinations in large-window queries. For small-window queries, the reduction of object examinations is due to the small number of objects a fine-grained cell intersects. Since the multicell quad tree supports fine subdivisions of the 2-space into cells, the memory utilization problem is very crucial. We have used some techniques to reduce the memory utilization.

In addition to its two-level structure, the multicell quad tree supports some implementation techniques to speed up the query algorithm. The conventional query algorithm executes many rectangle intersection tests to determine which objects intersect a query window. We have introduced rectangle (partial) containment tests. The execution of rectangle (partial) containment tests can eliminate or speed up the execution of a large number of rectangle intersection tests in large-window queries. On the other hand, the conventional quad tree search algorithm performs four rectangle intersection tests for each nonleaf node it visits. Each intersection test consists of four comparison operations. By careful analysis of these intersection tests, many redundant comparison operations are removed which results in a faster implementation of the search algorithm. Incidentally, this new implementation requires less memory to store the nonleaf nodes of the quad tree.

2. The multicell quad tree

The multicell quad tree is a refinement of the MSQT. Let us first recall the structure of a MSQT. Consider a 2-space where rectangular objects are placed. As objects are added to the 2-space, we subdivide the 2-space into different-sized quads. Initially, the entire 2-space is treated as a single quad. Once the number of objects intersecting a quad exceeds a threshold value $T$, the quad is subdivided into 4 equal-sized subquads. These subquads can then be treated as quads for further subdivision. We obtain a multiple storage quad tree by associating a tree node with a quad. Leaf
quads are those quads which are not subdivided. Associated with a leaf quad is a leaf node. A leaf node of the tree has no subnode. A nonleaf node of the tree has exactly 4 links to its 4 subnodes reflecting the quad/subquad relationship. A leaf node has a linked list containing references (i.e. pointers) to those objects intersecting its associated leaf quad. Since an object may intersect more than one leaf quad, there may be multiple references to the same object in a MSQT.

A MSQT typically has much more object references than tree nodes. Thus its query speed is mainly dependent on the number of object references the query algorithm examines. The number of examined object references is usually greater than the number of objects the query window really intersects. This results from the boundary and the duplication effects. Leaf quads intersecting the boundary of the query window may intersect some objects which are examined by the query algorithm, but do not intersect the query window. The unnecessary examinations of these objects around the boundary of the query window constitute the boundary effect. Due to duplications of object references, the query algorithm may examine multiple copies of an object reference in different leaf quads. These multiple examinations of object references constitute the duplication effect. The combination of the boundary and the duplication effects determines the ratio between the number of object references the query algorithm examines and the number of objects the algorithm reports which then determines the query performance. For small windows, the query algorithm examines a small number of leaf quads. Thus the duplication effect is insignificant, and the boundary effect dominates. (If the query algorithm only examines a single leaf quad, the duplication effect is nonexistent.) Since the boundary effect depends on the sizes of leaf quads, threshold $T$ of a MSQT is the determining factor to the performance of small-window queries. $T$ should be set to a small number in order to maximize the performance of small-window queries. On the other hand, a large query window intersects a large number of leaf quads. Among these leaf quads, only a small fraction are around the boundary of the query window. Thus the boundary effect plays a less significant role than the duplication effect in large-window queries. For better large-window performance, threshold $T$ of a MSQT should be set to a relatively large number to alleviate the duplication effect, i.e. to reduce the duplications of object references.

We have pointed out that large-window and small-window queries of a MSQT dictate two opposite conditions to be efficient. We will present a two-level structure, the multicell quad tree, which can satisfy both conditions in its single structure thus making both large-window and small-window queries efficient. A multicell quad tree has the same tree structure as a MSQT. It differs from a MSQT in its leaf quad (leaf node) structure. Instead of using a linked list to represent a leaf quad, we arrange the objects intersecting a leaf quad into a mesh structure. Each leaf quad is subdivided into $M*M$ equal-sized cells. Each cell has its own list for recording all objects intersecting the cell.

The mesh structure for a leaf quad consists of one object reference list and $M*M$ object index lists. Each list is realized as a pointer to a dynamic array. Thus a leaf node of a multicell quad tree has a static structure of $1+M*M$ pointers and $1+M*M$ arrays which are dynamic in size. One of the pointers is to the object reference array, and the others, organized into an $M*M$ cell array, are to object index arrays. The object reference array contains pointers to those objects intersecting the leaf quad. For each cell of a leaf quad, there is an object index array in which each entry is an index to the object reference array. Here we use an index as an identifier of an object. This object index array contains indices for those objects intersecting the cell.

For a multicell quad tree, leaf quads are further subdivided into cells. Consider a leaf quad as shown in Fig. 1. It is subdivided into $4*4$ equal-sized cells and its node structure is shown in Fig. 2. In Fig. 2, the object reference array contains references to those objects intersecting this leaf quad, namely a, b, c and d. Entries of the cell array are pointers to object index arrays which then contain indices for objects intersecting the individual cells. For example, cell (1,1) in Fig. 1 intersects objects a and b. Thus entry (1,1) of the cell array in Fig. 2 points to an array containing indices 0 and 1 which indicate the 0th and the 1st entries in the object reference array, i.e. objects a and b.

Using a conventional MSQT, the 2-space is typically subdivided into relatively coarse quads, say $T = 30$, to avoid the excessive duplications of object references. In contrast, using a multicell quad tree, the 2-space can be subdivided into much finer cells, typically $T = 256$ and $M = 8$. Thus the memory consumption problem must be addressed in a multicell quad tree. We have taken two actions. First, we use a relatively large threshold $T$ for a multicell quad tree to reduce the number of duplicates in object references. Second, we select $T$ to be less than 256. Then, indices to the object reference array can be represented by bytes, which results in a considerable amount of memory savings in object index arrays. With these techniques, a multicell quad tree can support finer subdivisions of the 2-space than a MSQT but with less memory consumption.
The presence of object reference lists as well as object index lists in a multicell quad tree is crucial to the efficiency of region queries. For large-window queries, the query algorithm makes a tree search on the upper-level MSQT and then inspects the appropriate object reference lists for objects intersecting the query window. Since it does not access the object index lists, the query algorithm works like that for a conventional MSQT. The difference is that the upper-level MSQT of a multicell quad tree has a relatively large threshold $T$. From our earlier arguments on the MSQT, large-window queries are efficient when $T$ is relatively large. The same arguments apply to the multicell quad tree.

For small-window queries, the query algorithm first makes a tree search on the upper-level MSQT to locate the appropriate leaf quads. Then it locates the appropriate object index lists by direct searches on the mesh structures. Finally, objects are accessed by first following object indices to object references and then following object references. Since the 2-space is subdivided into very fine cells in a multicell quad tree, each object index list only contains indices for a small number of objects. As we have argued previously, the query algorithm makes relatively few object examinations in this case.

3. Implementation Techniques

We have used two implementation techniques to speed up the query algorithm of a multicell quad tree. One technique enhances the search speed of region queries by reducing the operations executed at each nonleaf node, and the other accelerates the examination speed of large-window queries by introducing rectangle containment and partial containment tests on leaf quads. These implementation techniques are quite general and can be applied to a MSQT and other variations of quad trees. We shall illustrate these implementation techniques with a MSQT.

For a nonleaf quad $Q$ and a query window $W$, the query algorithm for a conventional MSQT executes 4 rectangle intersection tests to determine if $W$ intersects the 4 subquads of $Q$. Let $Q$ and $W$ have coordinates $(Q_{\text{xmin}}, Q_{\text{ymin}}, Q_{\text{xmax}}, Q_{\text{ymax}})$ and $(W_{\text{xmin}}, W_{\text{ymin}}, W_{\text{xmax}}, W_{\text{ymax}})$ respectively. The rectangle intersection test intersect() is typically implemented as the following macro:

```c
#define intersect(Q, W) 
(Q_{\text{xmin}} \leq W_{\text{xmax}}) && (Q_{\text{xmax}} \geq W_{\text{xmin}}) 
&& (Q_{\text{ymin}} \leq W_{\text{ymax}}) && (Q_{\text{ymax}} \geq W_{\text{ymin}})
```

There are 4 comparisons in this macro. Thus, when this macro is applied to the 4 subquads of $Q$, there are 16 comparisons to be executed in the worst case. By careful inspection of these comparisons, we have found that half of them are redundant and can be eliminated. Take the lower left subquad $Q_0 = (Q_{\text{xmin}}, Q_{\text{ymin}}, M_x, M_y)$ of $Q$ as an example where $M_x = (Q_{\text{xmin}} + Q_{\text{xmax}})/2$ and $M_y = (Q_{\text{ymin}} + Q_{\text{ymax}})/2$. Then

```c
intersect(Q_0, W) = (Q_{\text{xmin}} \leq W_{\text{xmax}}) &&
(M_x \geq W_{\text{xmin}}) &&
(Q_{\text{ymin}} \leq W_{\text{ymax}}) &&
(M_y \geq W_{\text{ymin}})
```

Note that comparisons $(Q_{\text{xmin}} \leq W_{\text{xmax}})$ and $(Q_{\text{ymin}} \leq W_{\text{ymax}})$ in $\text{intersect}(Q_0, W)$ also appears in $\text{intersect}(Q, W)$. Since $\text{intersect}(Q, W) = \text{TRUE}$ (otherwise no region query would be executed on $Q$), we have

$(Q_{\text{xmin}} \leq W_{\text{xmax}}) = (Q_{\text{ymin}} \leq W_{\text{ymax}}) = \text{TRUE}$.

In other words, only two comparisons $(M_x \geq W_{\text{xmin}})$ and $(M_y \geq W_{\text{ymin}})$ are really necessary to determine the value of $\text{intersect}(Q_0, W)$. Similar conditions also hold for the other three subquads of $Q$. Thus only 8 of the 16 comparisons need to be executed for each nonleaf node. Moreover, since these necessary comparisons involve $M_x$ and $M_y$ instead of the coordinates of $Q$, one can store $M_x$ and $M_y$ in a nonleaf node in place of the conventional 4 coordinates of $Q$. The advantages are a saving of memory space and a further elimination of the computations for $M_x$ and $M_y$.

Once the query algorithm locates an appropriate leaf quad, it executes rectangle intersection tests on objects inside the quad to determine which of them intersect the query window. A large query window typically contains many leaf quads. For these leaf quads, no rectangle intersection tests really need to be executed since all objects inside these quads intersect the query window. To find out whether a leaf quad $Q$ is contained in a query window $W$, we execute the following rectangle containment test:

```c
contain(Q, W) = (Q_{\text{xmin}} \geq W_{\text{xmin}}) &&
(Q_{\text{ymin}} \geq W_{\text{ymin}}) &&
(Q_{\text{xmax}} \leq W_{\text{xmax}}) &&
(Q_{\text{ymax}} \leq W_{\text{ymax}})
```

For large-window queries, the overhead of rectangle containment tests is very little compared with the gain due to the elimination of many rectangle intersection tests.

The query speed of large-window queries can be further improved by applying rectangle partial containment tests on leaf quads. Consider a large query window $W$ which is applied to a MSQT. There are 3 types
of leaf quads, namely $T_0$, $T_1$ and $T_2$, intersecting $W$. Leaf quads of type $T_0$ are entirely contained in $W$. A leaf quad is of type $T_1$ if two of its vertices are covered by $W$. Leaf quads of type $T_2$ are those quads one of whose vertices is covered by $W$. A containment test determines whether a leaf quad is of type $T_0$. A rectangle partial containment test determines whether a leaf quad is of type $T_1$ or $T_2$. A rectangle partial containment test is implemented by associating a flag (a bit) with each of the four comparisons in the rectangle containment test. A flag will be set if its corresponding comparison is TRUE. A leaf quad is of type $T_1$ ($T_2$) if three (two) of the four flags are set. For a leaf quad of type $T_1$, only one of the four comparisons in the intersection test needs to be executed to determine whether an object inside the leaf quad intersects the query window. By classifying leaf quads into types $T_1$ and $T_2$, one can simplify rectangle intersection tests considerably making large-window queries more efficient.

4. Experiments

We have implemented both the multicell quad tree and the MSQT on our SUN 4/330 workstation with C language. Many layout data were generated by a random number generator to test the performances of the algorithms. Test results of 3 typical data are demonstrated in this section. In each test case, there are 40,000 objects. Objects in these test cases, data-1, data-2 and data-3, have their x and y dimensions varying from 100 to 450, from 100 to 1000 and from 100 to 1300, respectively. In these experiments, the threshold of the MSQT was set to the typical value 30. The multicell quad tree (MCQT) had its parameters $T$ and $M$ set to 256 and 8 respectively.

Table I shows the number of object references/indices stored in these test runs. There are two kinds of duplicates in a multicell quad tree: duplicates at the leaf quad level (i.e. those in the object reference lists) and duplicates at the cell level (i.e. those in the object index lists). By Table I, the multicell quad tree has less duplicates at the leaf quad level than the MSQT while it has more duplicates at the cell level. The memory requirements of both trees are also listed in Table 1. The multicell quad tree uses less memory space than the MSQT for all 3 test cases. We can also see that the more complex the test case is, the more memory space is saved with a multicell quad tree.

Table II shows the average number of object references the query algorithm examines per query window. Three kinds of representative windows are considered: 10,000 random point windows (0 * 0), 10,000 random medium windows (1/8 * 1/8 of the whole space) and 10,000 random large windows (1/2 * 1/2 of the whole space). The number of object references examined in a multicell quad tree is less than that in a MSQT for all types of windows. Note that for medium/large-window queries, the multicell quad tree is most effective in reducing the object reference examinations when the layout data are very complicated. But for point windows, the multicell quad tree is more effective when the layout data are less complicated. Besides counting the number of object references examined, some direct time measurements were also collected. Query times for the MSQT and the multicell quad tree are reported in Table III. The search technique and the rectangle containment and partial containment tests are applied to the multicell quad tree. The rectangle containment test is also used in the query procedure of the MSQT.

The effectiveness of the implementation techniques are measured separately and listed in Tables IV and V. Table IV shows the search time for 10,000 random windows. The search time for a multicell quad tree includes the time for visiting the tree nodes and the time spent on the mesh structure to find the appropriate object index lists. The results of applying rectangle containment and partial containment tests on data-3 are reported in Table V. (These tests would not be applied to small windows.)

The multicell quad tree is a little more complicated than the MSQT. However, test results (not reported in this paper) have indicated that there is no significant penalty in building a multicell quad tree than a MSQT. The insertion of an object into a multicell quad tree is not so efficient as that for a MSQT in the worse case. But the maximum insertion time observed in these test cases is no more than 0.08 second which is well tolerable in interactive applications.

5. Conclusions

Compared with other variations of the quad tree, the multicell quad tree has the following novel features: The multi-branch tree structures [2] only improve the search speed of small-window queries while the multicell quad tree improves the examination speed as well as the search speed. The multicell quad tree is also suitable for large-window queries. In particular, when the layout data are complicated, the query algorithm for the multicell quad tree is remarkably more efficient than that for the MSQT. Note that when the layout data are not complicated, most quad tree query algorithms including the multicell quad tree are efficient anyway. Despite its efficiency in query speed, the multicell quad tree spends less memory than the MSQT. And the difference is more apparent when the
layout data are complicated.

References


<table>
<thead>
<tr>
<th>object references/indices</th>
<th>memory(Mbyte)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSQT</td>
</tr>
<tr>
<td>data-1</td>
<td>64166</td>
</tr>
<tr>
<td>data-2</td>
<td>106010</td>
</tr>
<tr>
<td>data-3</td>
<td>162809</td>
</tr>
</tbody>
</table>

Table I. Duplicates and memory consumption

<table>
<thead>
<tr>
<th></th>
<th>0*0</th>
<th>1 1/8</th>
<th>1 1/22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSQT</td>
<td>MCQT</td>
<td>MSQT</td>
</tr>
<tr>
<td>data-1</td>
<td>15.6</td>
<td>5.7</td>
<td>1283</td>
</tr>
<tr>
<td>data-2</td>
<td>21.2</td>
<td>10.5</td>
<td>2092</td>
</tr>
<tr>
<td>data-3</td>
<td>21.4</td>
<td>13.6</td>
<td>3126</td>
</tr>
</tbody>
</table>

Table II. Average object references examined

<table>
<thead>
<tr>
<th></th>
<th>0*0</th>
<th>1 1/8</th>
<th>1 1/22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSQT</td>
<td>MCQT</td>
<td>MSQT</td>
</tr>
<tr>
<td>data-1</td>
<td>0.81</td>
<td>0.40</td>
<td>50.78</td>
</tr>
<tr>
<td>data-2</td>
<td>1.12</td>
<td>0.65</td>
<td>62.98</td>
</tr>
<tr>
<td>data-3</td>
<td>1.25</td>
<td>0.81</td>
<td>84.72</td>
</tr>
</tbody>
</table>

Table III. Query time for 10,000 windows

---

<table>
<thead>
<tr>
<th></th>
<th>0*0</th>
<th>1 1/8</th>
<th>1 1/22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSQT</td>
<td>MCQT</td>
<td>MSQT</td>
</tr>
<tr>
<td>data-1</td>
<td>0.28</td>
<td>0.13</td>
<td>2.77</td>
</tr>
<tr>
<td>data-2</td>
<td>0.30</td>
<td>0.13</td>
<td>3.57</td>
</tr>
<tr>
<td>data-3</td>
<td>0.35</td>
<td>0.14</td>
<td>5.42</td>
</tr>
</tbody>
</table>

Table IV. Search time for 10,000 windows

---

<table>
<thead>
<tr>
<th></th>
<th>0*0</th>
<th>1 1/8</th>
<th>1 1/22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSQT</td>
<td>MCQT</td>
<td>MSQT</td>
</tr>
<tr>
<td>No contain</td>
<td>97.87</td>
<td>64.58</td>
<td>1328.62</td>
</tr>
<tr>
<td>contain</td>
<td>84.72</td>
<td>62.90</td>
<td>1137.60</td>
</tr>
<tr>
<td>+ partial</td>
<td>81.63</td>
<td>54.15</td>
<td>1124.28</td>
</tr>
</tbody>
</table>

Table V. Rectangle (partial) containment tests

---

Fig. 1. Leaf quad subdivided into cells

Fig. 2. Structure of leaf node