Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Array based HV/VH tree: an effective data structure for layout representation*

Jie REN†, Wei-wei PAN, Yong-jun ZHENG, Zheng SHI†‡, Xiao-lang YAN
(*Institute of VLSI Design, Zhejiang University, Hangzhou 310027, China*)
†E-mail: {renjie, shiz}@vlsi.zju.edu.cn

Received July 4, 2011; Revision accepted Oct. 25, 2011; Crosschecked Feb. 8, 2012

Abstract: We present a new data structure for the representation of an integrated circuit layout. It is a modified HV/VH tree using arrays as the primary container in bisector lists and leaf nodes. By grouping and sorting objects within these arrays together with a customized binary search algorithm, our new data structure provides excellent performance in both memory usage and region query speed. Experimental results show that in comparison with the original HV/VH tree, which has been regarded as the best layout data structure to date, the new data structure uses much less memory and can become 30% faster on region query.

Key words: Very large scale integration (VLSI), Layout representation, HV/VH trees, Region query
doi:10.1631/jzus.C1100193 **Document code:** A **CLC number:** TN47

1 Introduction

The fabrication of an integrated circuit (IC) is costly and time-consuming; therefore, it is essential that the layout's correctness be verified before the final tape-out. A series of processes are standard in this layout verification routine, such as design rule check, parasitic extraction, transistor extraction, and layout versus schematic (LVS). All these processes are based on geometric operations on circuit layout; thus, it is important to choose a proper data structure for layout representation. The data structure needs to be memory-efficient and faster enough for common layout operations.

An important operation on circuit layout is referred to as region query, which is defined as finding the objects that intersect with a given rectangular window. Region query is greatly used in the above processes. Its speed is the main indicator for evaluating layout data structures.

Various spatial data structures have been proposed for layout representation, ranking from the simple linear linked list to corner stitching (Ousterhout, 1982), then the more sophisticated *k*-d trees (Rosenberg, 1985) and the large family of quad-trees (Kedem, 1982; Brown, 1986; Pitaksanonkul *et al.*, 1989; Weyten and de Pauw, 1989; Lai *et al.*, 1993; 1996; Berg *et al.*, 2008). A summary is listed in Table 1.

Table 1 Various spatial data structures for IC layout representation

Reference	Spatial data structure	
	Abbreviation	Full name
Kedem, 1982	BLQT	Bisector List Quad Tree
Rosenberg, 1985	*k*-d tree	*k*-dimensional tree
Brown, 1986	MSQT	Multiple Storage Quad Tree
Weyten and de Pauw, 1989	QLQT	Quad List Quad Tree
Pitaksanonkul *et al.*, 1989	BQT	Bounded Quad Tree
Lai *et al.*, 1993	HVT	Horizontal/Vertical Tree

Most of the research in IC layout data structures was undertaken in the early 1980s through the

‡ Corresponding author
* Project supported by the National Natural Science Foundation of China (No. 61106034) and the National Science and Technology Major Project (No. 2009ZX02023-004-1)

mid-1990s. This area has not been very active since then. However, it is believed that there remain opportunities for developing better layout data structures (Samet, 1990a; 1990b; Mehta, 2005).

Comprehensive experiments comparing HVT with BQT, *k*-d, and QLQT showed that the data structures ordered from best to worst in terms of space requirements were HVT, BQT, *k*-d, and QLQT. In terms of region query speed, the best data structures were HVT and QLQT followed by BQT and finally *k*-d (Samet, 2006; Mehta and Zhou, 2008). Since it was published, HVT has been the data structure of choice in terms of both memory usage and region query speed.

An HV/VH tree consists of alternate levels of H- and V-nodes. An H-node splits the 2D space assigned to it into two halves with a horizontal bisector, while a V-node does the same with a vertical bisector. An H/V-node is not split if the number of objects assigned to it is less than some fixed threshold. In such a case, the node is marked as a leaf node, and objects in it are stored into a single linked list. This threshold is referred to as the HV-threshold.

Objects intersecting an H/V-node's bisector are stored in the bisector list of the node. Bisector lists of HVT are implemented with binary cut trees (Kedem, 1982; Lai *et al*., 1993). A horizontal bisector is divided into identical halves by a vertical cut-line, and objects intersecting with it are stored in cut tree's root. Other objects are assigned to either the left or the right child of the root node, depending on its relative position to the vertical cut-line. Then we repeat this procedure to both children of the root recursively. The subdivision is stopped when the number of objects in a cut tree node is less than another fixed threshold, which we call the B-threshold. Cut tree nodes also use a linked list to store objects.

In our method, cut trees and linked lists are all replaced with arrays; thus, a considerable number of pointers are eliminated and the memory usage is reduced. Meanwhile, by well-organizing and applying a customized binary search algorithm to these arrays, the region query speed of our data structure is also competitive. By experiments, we show that the array based HV/VH tree is faster than all its rivals on region query and uses the least memory at the same time.

2 Array based HV/VH tree

Array based HV/VH trees are similar to HV/VH trees. They are also composed of alternate levels of H- and V-nodes. The HV-threshold and B-threshold we mentioned above remain valid in the new data structure. For convenience, we will use ABHVT for array based HV/VH tree in the following text.

Fig. 1 shows a sample layout and its corresponding implementation of both HVT and ABHVT.

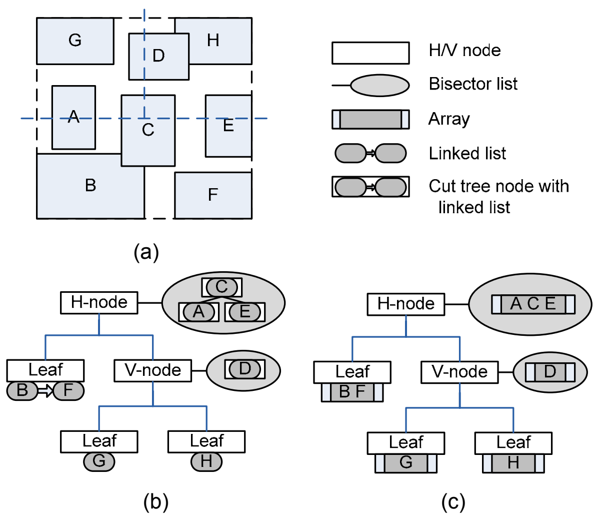


Fig. 1 The sample layout (a), HVT implementation (b), and ABHVT implementation (c)
The horizontal dashed line in (a) is the bisector of the root node (an H-node). It divides the layout into two halves. Objects intersecting this divider compose the root's bisector list, and the bisector list contains three objects A, C, and E. Then the procedure is repeated for the lower and upper halves of the layout in a recursive manner. The vertical dashed line in (a) is the bisector of the upper half. The lower half is not further divided since the number of objects in it is two, which equals the HV-threshold

Comparison of Figs. 1b and 1c shows that the major difference between HVT and ABHVT locates in leaf nodes and bisector lists. In HVT, linked lists are used as the container for layout objects and bisector lists are implemented with cut trees, whereas in ABHVT, both leaf nodes and bisector lists are implemented with plain arrays. With ABHVT, we can get rid of the complicated cut trees, and the array implementation also seems simple. However, this change would also cause serious degradation in the speed aspect due to the existence of very large arrays. These large arrays usually belong to bisector lists of high-level H/V-nodes. In the following sections, we will show how we eliminate this adverse effect.

2.1 Optimization to bisector list arrays

The optimization process consists of two steps, grouping and sorting. Here we take a horizontal bisector list for demonstration. Vertical bisector lists can be handled in a similar manner. Note that, if the number of objects in a bisector list is smaller than the B-threshold, no optimization is performed. For leaf node arrays, no optimization is performed either, since their array-size is also limited by the HV-threshold. A full check is always performed to small arrays during region query.

Given a horizontal bisector list containing more objects than the B-threshold, first we partition its objects into several subgroups (if needed); thus, within each subgroup, the objects are of similar width. In Fig. 2, the 12 objects are divided into two subgroups G1 and G2. Objects C and H are picked out because their widths are obviously larger than those of the others.

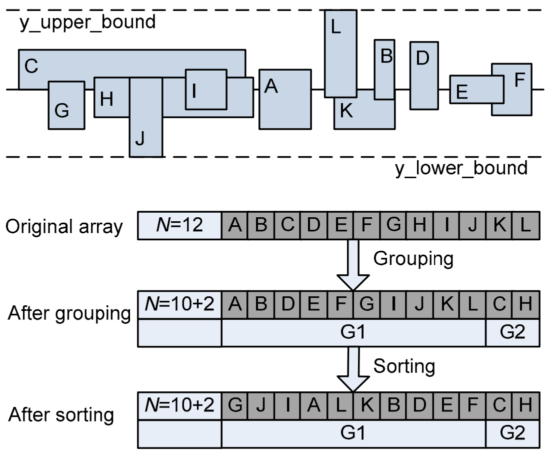


Fig. 2 Optimization to a horizontal bisector list

Then, we sort each subgroup by the left boundary coordinate of the objects. See the array marked by 'After sorting' in Fig. 2. Within each subgroup, there is a guarantee that

$$\text{Array}[i].\text{lx} \leq \text{Array}[j].\text{lx}, \ \text{if } i<j.$$

Array[i].lx stands for the left boundary coordinate of the ith object of the subgroup array.

Additionally, we keep two integers in the bisector list to record its boundaries in the orthogonal direction of the bisector. See y_lower_bound and y_upper_bound in Fig. 2.

2.2 Binary search algorithm

The binary search algorithm is vital in making ABHVT's region query fast. It takes in a subgroup along with a search window, and tries to obtain a minimal subset of the subgroup. The subset obtained guarantees that any objects out of it definitely do not intersect with the search window.

Assume Fig. 3 depicts the corresponding layout of a subgroup, where window.x1 and window.x2 are the left and right boundaries of the search window, respectively, and max_width is the maximal object width of this subgroup. Fig. 3 shows that the array is divided into four zones by three vertical cut-lines window.x1, window.x2, and window.x1−max_width. For the four zones, we know:

1. Zone 1 and Zone 4 objects do not intersect with the search window.

2. Zone 3 objects intersect with the search window in the x direction.

3. Zone 2 objects probably intersect with the search window in the x direction.

Thus, objects of Zone 2 and Zone 3 are returned as the binary search result. They will be passed on for further checks. Note that, during binary search, we have no choice but to put Zone 2 objects into our results since they are suspects. And their presence degrades the efficiency of region query. However, we could diminish the quantity of Zone 2 objects by minimizing the parameter max_width. This explains why we would group them and keep objects of similar width in the same subgroup in the first place.

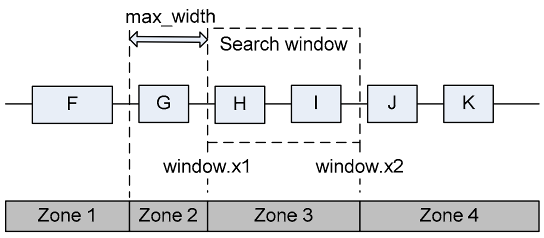


Fig. 3 Binary search with a horizontal subgroup array

2.3 Region query with ABHVT

Region query with ABHVT can be implemented with a simple recursive procedure. The pseudo code in C is written below.

```
region_query(treenode, window) {
    // treenode could be either an H-node or a V-node
    if (treenode is leaf node) {
        perform full check to the leaf node array;
    }
    // blist stands for the bisector list of treenode;
    if (blist intersects window) {
        if (blist is horizontal)
            region_query_to_hbisector(blist, window);
        else
            region_query_to_vbisector(blist, window);
    }
    if (child1 intersects window)
        region_query(child1, window);
    if (child2 intersects window)
        region_query(child2, window);
}
region_query_to_hbisector(blist, window) {
    if (window does not intersect y bounds of blist)
        return;
    foreach (subgroup) {
    // window.x1 and window.x2 stand for the left and
    // right bounds of window, respectively
        int x0=window.x1–max_width;
        int pos0=bisearch1(subgroup, x0);
        int pos1=bisearch2(subgroup, window.x1);
        int pos2=bisearch2(subgroup, window.x2);
        foreach (object whose index∈[pos0, pos1))
            check intersection with window;
        foreach (object whose index∈[pos1, pos2))
            if (divider of blist intersects window)
                add this object to results immediately;
            else
                check intersection with window in y direction only;
    }
}
int bisearch1(array, x) {
    do binary search on array to find an index pos satisfying
        array[pos–1].x1<x and array[pos].x1≥x;
    return pos;
}
int bisearch2(array, x) {
    do binary search on array to find an index pos satisfying
        array[pos–1].x1≤x and array[pos].x1>x;
    return pos;
}
```

Continuing to use the previous horizontal bisector list example, the corresponding layout of its subgroup G1 is obtained, as shown in Fig. 4, in which all objects are relatively small. Our goal here is to find the objects intersecting with the search window with minimal calculations.

For Search 1, the entire subgroup can be skipped since by checking *y* boundaries, we know that none of the objects in G1 intersect with the search window.

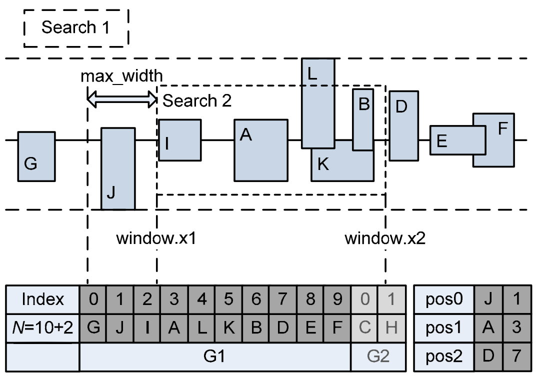


Fig. 4 Region query on subgroup G1

For Search 2, we obtain three indices pos0, pos1, and pos2 by performing binary searches:

pos0=bisearch1(window.x1–max_width),
pos1=bisearch2(window.x1),
pos2=bisearch2(window.x1).

Clearly, objects out of index range [pos0, pos2) do not intersect with the search window. Our search area is thus reduced from the entire array to the objects within range [pos0, pos2).

Normally, checking whether an object intersects with a rectangular window requires four comparisons, two in the *x* direction and two in the *y* direction:

$$\text{window.x1}\le\text{object.x2},\ \ \text{window.x2}\ge\text{object.x1},$$
$$\text{window.y1}\le\text{object.y2},\ \ \text{window.y2}\ge\text{object.y1}.$$

In ABHVT, however, these four comparisons are not always simultaneously necessary. In Fig. 4, since both the Search 2 window and the bisector list objects cross the bisector, *y*-direction comparisons are not needed for any object of G1. This is based on the fact that if two rectangles are crossed by a common horizontal line, then they must overlap with each other in the vertical direction. Furthermore, *x*-direction comparisons are needed only for objects within index range [pos0, pos1), while the objects within range [pos1, pos2) can be determined to be intersecting with the Search 2 window without any comparison.

2.4 Data structures

A leaf node in ABHVT has an unsigned integer recording the array size and a pointer to the array header:

```
class CLeaf{
    size_t count;
    CObject **obj_list;
};
```

A bisector list may contain several subgroup arrays, but we still use one single array for storage, and we record the starting indices for each sub-array.

```
class CBisector{
    size_t count;            // total number of objects
    CObject **obj_list;
    int lower_bound, upper_bound;
    size_t num_of_groups;
    int *max_sizes;          // max object width/height
    size_t *group_sizes;  // size of each subgroup
};
```

An ABHVT tree node has two child-pointers, a divider of integer type, and a pointer to either a leaf node or a bisector list. With the dual use of the body field, an extra flag is needed to distinguish between the two cases. In order to save memory, we embed this flag into the integer mid at its highest bit (next to the sign bit).

```
class CTreeNode{
    CTreeNode *left, *right;
    int mid; // bisector
    union{
        CLeaf *leaf;
        CBisector *bisector;
    } body;
};
```

Note that given a non-leaf H/V-node, if the number of objects in its bisector list is smaller than the B-threshold, a leaf instead of a bisector list will be created for the body field.

3 Experimental results

We implemented QLQT and HVT to compare with ABHVT. All implementations were compiled with GCC 4.2.4 and the same optimization flag O2. Our test cases consisted of 10 circuits selected from IWLS 2005 benchmarks (Table 2). The B-threshold was fixed to 32 in all our tests.

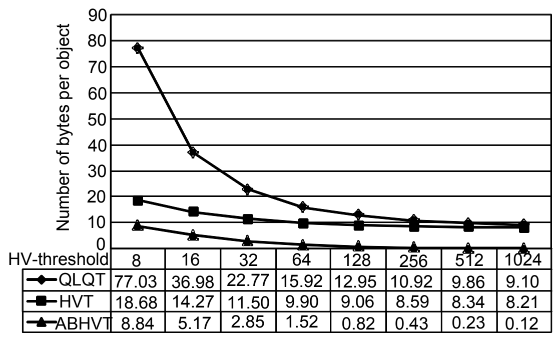
3.1 Space

To describe an object, we always need four integers to store object boundaries and one pointer for indexing. Memory used for this purpose is called 'object-description', and it is necessary for any kind of layout data structure. Hence, it makes sense that we exclude them in comparing memory usage between QLQT, HVT, and ABHVT.

Table 2 Ten circuits selected from IWLS 2005 benchmarks for testing

Circuit	Number of objects ($\times 10^3$)	Circuit	Number of objects ($\times 10^6$)
b01	2	DSP	1.0
b10	7	RISC	1.8
ac97_ctrl	400	ethernet	2.2
b22	600	vga_lcd	4.1
DMA	600	b19	4.4

Fig. 5 shows the results of our tests for space. The *x* coordinate of the chart is the HV-threshold and the *y* coordinate is the average number of bytes required per object (not including object descriptions). Data was collected on a 64-bit Linux server, where an integer takes four bytes and a pointer takes eight. ABHVT completely outperformed its two rivals. At a typical HV-threshold of 64, ABHVT, HVT, and QLQT used 1.52, 9.9, and 15.92 bytes of memory per object, respectively. Note that ABHVT used even less memory than the method of simply representing a layout with a single linked list.



HV-threshold	8	16	32	64	128	256	512	1024
◆ QLQT	77.03	36.98	22.77	15.92	12.95	10.92	9.86	9.10
■ HVT	18.68	14.27	11.50	9.90	9.06	8.59	8.34	8.21
▲ ABHVT	8.84	5.17	2.85	1.52	0.82	0.43	0.23	0.12

Fig. 5 Memory usage vs. the HV-threshold

3.2 Time

Fig. 6 plots the speedup factor of HVT and ABHVT, both compared with QLQT. The speedup factor is defined by dividing the region query time of HVT or ABHVT by that of QLQT. The *x* coordinate remains the HV-threshold. The *y* coordinate is the speedup factor. At each HV-threshold value, thousands of random region queries were performed for each layout, and the average speedup factor of the 10 layouts was used for plotting.

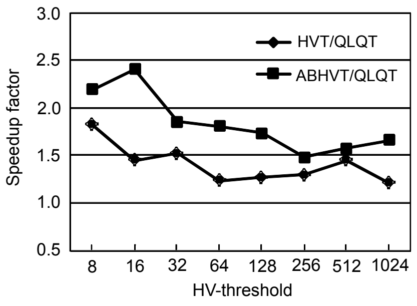


Fig. 6 Speedup factor vs. the HV-threshold

Fig. 6 indicates that ABHVT was the fastest among the three candidates, outperforming HVT at all HV-threshold values. At an HV-threshold of 64, ABHVT was about 45% faster than HVT and 80% faster than QLQT.

Further tests indicated that ABHVT is suitable especially for large region queries. Fig. 7 plots the speedup factor of HVT and ABHVT again compared with QLQT. The x coordinate is the side length of the square search window. ABHVT's advantage over HVT gets greater as the search window gets larger.

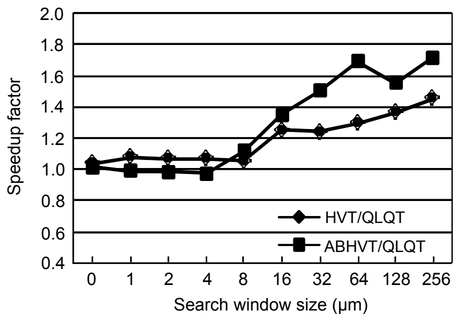


Fig. 7 Speedup factor vs. the search window size

4 Conclusions

A new spatial data structure is presented for IC layout representation. In comparison with HVT, which has been regarded as the best data structure for region query, our new data structure provides even better performance in both space and speed. By using arrays as the underlying data structure for layout objects, ABHVT uses even less memory than simple linked list implementation. Also, by grouping and sorting those arrays, the region query speed of ABHVT is very high.

Experimental results show that in ABHVT, only two bytes are required per object at typical thresholds (not including object descriptions). In terms of speed, ABHVT is 30% faster than HVT.

Note that the insertion and deletion operations with ABHVT are slow since inserting an element into an array is never as convenient as inserting it into a linked list. However, in most layout verification processes, the layouts are used in read-only mode and editing operations are rarely called. Thus, this disadvantage of ABHVT is negligible in practice. We believe that ABHVT is the perfect data structure for a wide range of IC layout applications.

References

Berg, M.D., Cheong, O., Kreveld, M.V., Overmars, M., 2008. Computational Geometry: Algorithms and Applications. Springer, the Netherlands, p.219-238.

Brown, R.L., 1986. Multiple storage quad trees: a simpler faster alternative to bisector list quad trees. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **5**(3):413-419. [doi:10.1109/TCAD.1986.1270210]

Kedem, G., 1982. The Quad-CIF Tree: a Data Structure for Hierarchical On-line Algorithms. Proc. 19th Design Automation Conf., p.352-357.

Lai, G.G., Fussell, D., Wong, D.F., 1993. HV/VH Trees: a New Spatial Data Structure for Fast Region Queries. Proc. 30th Int. Design Automation Conf., p.43-47. [doi:10.1145/157485.164562]

Lai, G.G., Fussell, D.S., Wong, D.F., 1996. Hinted quad trees for VLSI geometry DRC based on efficient searching for neighbors. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **15**(3):317-324. [doi:10.1109/43.489102]

Mehta, D.P., 2005. Handbook of Data Structures and Applications. Chapter 52: Layout Data Structures. Chapman & Hall/CRC, USA, p.1046-1063.

Mehta, D.P., Zhou, H., 2008. Handbook of Algorithms for Physical Design Automation. Chapter 4: Basic Data Structures. Auerbach Publications, FL, USA, p.55-69.

Ousterhout, J.K., 1982. Corner stitching: a data structure technique for VLSI layout tools. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **3**(1):87-100. [doi:10.1109/TCAD.1984.1270061]

Pitaksanonkul, A., Thanawastien, S., Lursinsap, C., 1989. Comparisons of quad trees and 4-D trees: new results. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **8**(11):1157-1164. [doi:10.1109/43.41501]

Rosenberg, J.B., 1985. Geographical data structures compared: a study of data structures supporting region queries. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **4**(1):53-67. [doi:10.1109/TCAD.1985.1270098]

Samet, H., 1990a. Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addison-Wesley, MA.

Samet, H., 1990b. The Design and Analysis of Spatial Data Structures. Addison-Wesley, MA.

Samet, H., 2006. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, CA, USA, p.474-483.

Weyten, L., de Pauw, W., 1989. Quad list quad trees: a geometrical data structure with improved performance for large region queries. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **8**(3):229-233. [doi:10.1109/43.21842]