HV/VH Trees: A New Spatial Data Structure for Fast Region Queries

Glenn G. Lai, Don Fussell, and D. F. Wong
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract—**Rosenberg compared linked lists, quad trees with bisector lists, and *kD* trees, and showed that *kD* trees significantly outperformed their two rivals on region queries. Quad trees with bisector lists performed poorly because of their need to search bisector lists at successive levels; therefore, later improvements to quad trees took the form of eliminating the bisector lists in one way or the other to achieve better region-query performance. In this paper, we explode the myth that bisector lists imply slow region queries by introducing a new data structure, HV/VH trees, which, even though it uses bisector lists, is as fast as or faster than *kD* trees and two improved forms of quad trees on region queries performed on data from real VLSI designs. Furthermore, we show that HV/VH trees achieve this superb perfomance while using the least amount of memory.**

I Introduction

Searching for objects that intersect a specified area in 2-D space is an operation used very frequently by VLSI tools; this is commonly referred to as *region query*. Among its many uses, the most important of all is probably in finding all the objects that come within a specified distance of an object—an operation that directly determines the speed with which design-rule checking can be performed. Usually, an object is represented by a rectangular bounding box, the search area constrained to be rectangular, and a region query answered by finding all the rectangles intersecting a rectangular search window; this abstraction is independent of the shapes of the objects and works for both Manhattan and nonManhattan geometry. We shall concentrate on this generic approach to solving the region-query problem, and not concern ourselves with special-purpose data structures, such as corner stitching[3, 8], that require a modified or new region-query algorithm to accommodate additional geometrical shapes.

In 1982, Kedem [2] published a paper describing the use of quad trees with bisector lists for storing and retrieving objects using their minimal bounding boxes. This was followed by Rosenberg's paper [4] comparing linked lists, quad trees with bisector lists, and *kD* trees. Several improvements to quad trees, all made by eliminating the bisector lists, were published later. Brown [5] stored pointers to an object in all the leaf quadrants it intersects and used mark and unmark operations to avoid finding the same object twice. Weyten and Pauw [6] remedied a problem in Brown's scheme and removed the mark and unmark operations by using four lists at the leaves to store the object pointers; they called the improved method QLQT, short for ***quad-list quad trees.*** Pitaksanonkul, Thanawastien, and Lursinsap [7] assigned a rectangle to the quadrant that contains its lower-left corner and kept bounds information for each quadrant; we shall refer to their method as BQT for ***bounded quad trees.*** They claimed that BQT, with a suitable threshold value, could be made to outperform *kD* trees.

Bisector lists, practically abandoned now, have long been regarded as the bane of region-query performance. In this paper, we present a new data structure, HV/VH trees, which, running counter to conventional wisdom, uses bisector lists as the primary structure to store rectangles. Using three sets of test data from real VLSI designs, we compare BQT, *kD* trees, HV/VH trees, and QLQT on region-query performance and memory usage, and show that HV/VH trees achieve excellent performance while minimizing memory usage.

II The HV/VH-Tree Data Structure

HV/VH trees are composed of HV nodes. An HV node that splits the space assigned to it into two halves with a horizontal divider is an H node; one that does it vertically with a vertical divider, a V node. An HV node is not split if the number of rectangles assigned to it is less than or equal to some fixed threshold; in this case, it stores the rectangles on a linked list. Children of H nodes are V nodes; children of V nodes, H nodes. An HV/VH tree with only one HV node is a special case: the node does not split the space assigned to it, thus whether it is an H node or a V node is undefined. Rectangles intersecting an H node's horizontal divider are stored in the node's horizontal bisector list; ones intersecting a V node's vertical divider, the node's vertical bisector list.

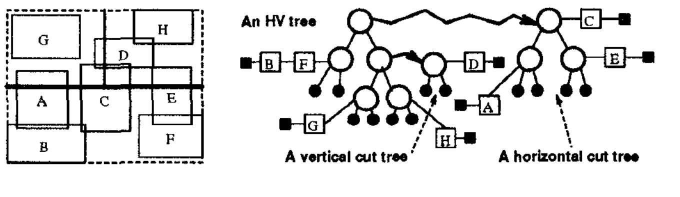

Fig. 1: An HV tree with an HV threshold of 2 and a cut threshold of 2

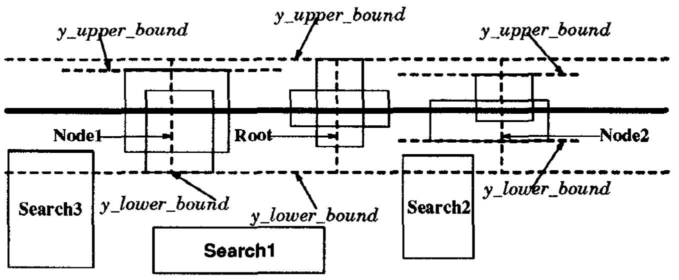

Fig. 2: Ways a horizontal bisector list or parts thereof can be skipped

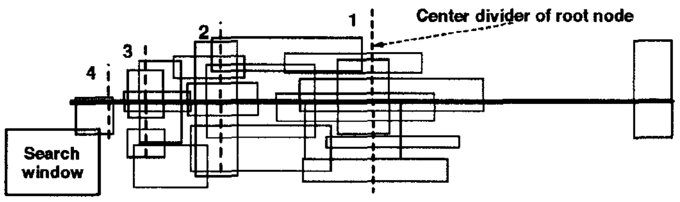

Fig. 3: Searches on lists 1—3, if performed, are wasted

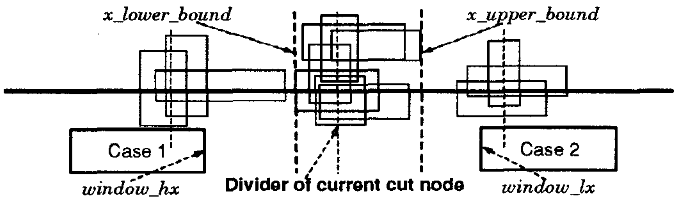

Fig. 4: Checking bounds information at a horizontal cut node

The only difference between HV trees and VH trees is that the former have H-node roots while the latter have V-node roots.

Our bisector lists are implemented with cut trees which are similar to Kedem's binary trees; however, we use a different termination criterion to stop splitting a node: when the number of rectangles assigned to a node is less than or equal to some fixed threshold. For convenience, a divider, whether it splits HV space or cut space, is defined as the shortest line segment that halves the assigned space. We shall restrict our discussion to HV trees and horizontal cut trees. Fig. 1 shows an example of an HV tree.

Unless a search window is very large in at least one dimension, it does not usually intersect any rectangles on most of the bisector lists. To take advantage of this observation, we store two bounds, *y_lower_bound* and *y_upper_bound*, at each horizontal cut node to filter out the horizontal bisector lists or parts thereof that do not contain rectangles intersecting the search window; see Fig. 2. For the three search windows shown, we can skip the bisector list entirely for **Search1**, we need to do something at **Root** but can skip **Node2** for **Search2,** and we need to do some checks at booth **Root** and **Node1** for **Search3.** These two bounds are the most crucial in making HV/VH trees perform region queries efficiently, as they avoid, as much as possible, the performance problem caused by the need to search all levels of bisector lists required by traditional methods using bisector lists.

The two bounds described above weed out needless checks on horizontal cut trees or their subtrees, but, sooner or later, a region query will usually need to check the rectangles in one or more cut trees. Visiting a short cut tree presents no major performance problems, but when a tree is relatively tall many useless checks would be performed if we did not consider the fact that most search windows usually intersect only a small number of rectangles stored in the cut nodes. The potential problem is illustrated in Fig. 3.

Our solution is to keep *x_lower_bound* and *x_upper_bound*, as illustrated in Fig. 4, at each horizontal cut node to bound the rectangles it stores in the X dimension. Searches can safely ignore the rectangles stored at non-leaf nodes, knowing that they do not intersect the search window, in two cases:

- if only the left subtree of the current node needs to be checked and $window\_hx < x\_lower\_bound$
- if only the right subtree of the current node needs to be checked and $window\_lx > x\_upper\_bound$.

If a search proceeds down to a leaf, the search window is probably very near to some of the rectangles stored in the leaf, and the four bounds described above are usually uesless in avoiding needless checks; this is the reason that we do not make use of the information at the leaves. To take advantage of the information the bounds provide, up to four extra comparisons could be made at each horizontal cut node, the cost of which is approximately the same as checking an extra rectangle for intersection with the search window; this is negligible considering the potential benefits to be reaped from the comparisons.

Lastly, to reduce the search overhead at a cut node, we record the mid-point that splits the cut space in halves at each cut node, instead of passing the bounds of the cut space to compute the mid-point at each cut node. This slightly increases the amount of memory required by a cut node, but it appreciably speeds up the traversal of a cut tree; we store the mid-point for an HV node for the same reason and gain.

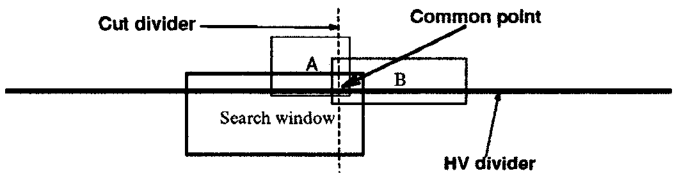

Fig. 5: Rectangles A and B share a common point with the search window

2.1. *Intersection Check*

Checking whether a rectangle intersects the search window usually requires the following four comparisons:

- $window\_lx \leq rect\_hx$
- $window\_hx \geq rect\_lx$
- $window\_ly \leq rect\_hy$
- $window\_hy \geq rect\_ly$

Since the rectangles stored in an HV node's cut tree intersect the the node's divider and it is always necessary to decide whether to visit one or both children of an HV node or a cut node, we may as well use the information to eliminate some of the comparisons. For example, if a region query needs to visit both children of an H node and, while traversing the corresponding horizontal cut tree, it needs to visit both children of a cut node, we know that the search window straddles the H node's horizontal divider and the cut node's vertical divider. In this case, no more comparisons are needed to learn that all the rectangles stored in the cut node intersect the search window, since they contain a common point; see Fig. 5. This is an extreme case, but it occurs very often for large windows; in all cases, we can always perform an intersection check for a rectangle stored in a cut node using fewer than four comparisons.

2.2. *Node Structures Used by HV/VH Trees*

In an HV/VH tree, rectangles are stored on linked lists attached either to an HV node or a cut node. As shown below, a rectangle node has four coordinates, representing the rectangle's lower-left and upper-right corners, and a pointer to the next rectangle node on the list.

```
typedef struct _box {
    INT32 rect[4];
    struct _box *next;
} BOX;
```

A cut node has a rectangle list, two bounds for the rectangle list, two bounds for the tree rooted at the node, a divider that splits the space assigned to it into two halves, and two pointers to its two children. Leaf nodes are their own left children and do not have valid divider values. The structure of a cut node is shown below:

```
typedef struct _cut {
    BOX *box;
    INT32 bound1, bound2;
    INT32 l_bound, h_bound;
    INT32 mid;
    struct _cut *left, *right;
} CUT;
```

For a horizontal cut node, *bound*1 and *bound*2 are, respectively, *x_lower_bound* and *x_upper_bound* that bound its rectangles in the X dimension; *l_bound* and *h_bound* are, respectively, *y_lower_bound* and *y_upper_bound* that bound rectangles stored in the tree rooted at the node in the Y dimension. For a vertical cut node, exchange the X, Y coordinates above. A cut node stores rectangles on a linked list pointed to by *box*; its associated divider is stored in *mid*.

An HV node has two child pointers, a divider that splits the space assigned to it into two halves, and a pointer to either a cut tree or a rectangle list. An HV node has the following structure:

```
typedef struct _hvNode {
    struct _hvNode *left, *right;
    INT32 mid;
    union {
        CUT *cut;
        BOX *box;
    } u1;
} HV_NODE;
```

With the dual usage of the `u1` field, we must distinguish between two types of leaf nodes: the ones assigned more than the threshold number of rectangles and those that are not. The former type of nodes are made leaves because all of their assigned rectangles intersect their dividers; we make such nodes their own left children. Since we let the latter type of nodes have null left children, the usage of the `u1` field is unambiguous.

III Experimental Results

In comparing against our HV/VH implementation, we used Rosenberg's published C code for *kD* trees and implemented BQT and QLQT ourselves, since neither came with actual code in the original paper; to ensure correctness, results returned by all region queries were verified with those obtained by a linked-list implementation. To speed up Rosenberg's code, we removed the parts used to collect statistics; Rosenberg called a function that makes four comparisons to check for intersection—we made the four comparisons in-line. Rosenberg's description of cutting off the searches in the main text was correct; however, his code incorrectly left out '=' in all eight places checking the bounds—we put the '='s back in. All implementations were compiled with GCC 2.3.1 using the same optimization flag ('O2') and their run times collected on a 33-MHz 386 machine with a 32-K cache. Our data sets, **Test1, Test2,** and **Test3** contain 11,307, 21,073 rectangles, 55,644 rectangles, respectively; the rectangles are the bounding boxes of the objects of three real VLSI designs. We used all of the rectangles themselves as the search windows to perform an exhaustive test for each data set.

A few notes before we present our results:

1. QLQT, as originally defined, stores a pointer to an object description in all the quadrants intersecting the object's smallest enclosing rectangle. We are concerned only with the enclosing rectangles; therefore, in our QLQT implementation the pointers point to rectangles, not object descriptions.

2. The threshold values used were 32 for QLQT and 10 for BQT; kD trees do not require such a parameter. Both the HV and cut thresholds were set to eight for HV/VH trees. These values were used because they generally gave better results than other values did.

3. If more than N rectangles share the same lower-left corner, the BQT building process will not terminate when its threshold is set to a value less than N; this problem occurred for **Test2** and **Test3**. We corrected the problem by adding an extra termination criterion to stop splitting a quadrant: when all the rectangles assigned to a quadrant have the same lower-left corner.

3.1. Space

Assuming that one word is required for an integer and a pointer of any type, Table I shows the average number of words used by a rectangle for all four tree structures.

TABLE I: NUMBER OF WORDS USED BY A RECTANGLE

	BQT	HV/VH	kD	QLQT
Test1	8.38	6.95	10	10.74
Test2	8.58	7.02	10	10.97
Test3	8.56	6.95	10	14.73

A rectangle in a kD tree always uses 10 words of storage; the average number of words used by a rectangle is basically decided by the threshold(s) chosen and stay relatively constant for both BQT and HV/VH trees; the memory usage of a rectangle for QLQT does not show signs of stability. For **Test1** and **Test2,** QLQT stores 1.96 and 1.98 pointers/rectangle, resulting in 10.74 words and 10.97 words per rectangle. For **Test3,** the number of pointers per rectangle grows to 3.43, resulting in 14.73 words per rectangle—this is more than twice the memory used by HV/VH trees. The phenomenon is probably the result of the large number of rectangles that enclose many small rectangles in **Test3**.

3.2. Time

Shown in Figures 6–8 are the HV/VH speedup factors for all the search windows tested: the X coordinate is computed by dividing the time required by a tree structure (kD trees, BQT, or QLQT) by the corresponding HV/VH time and rounded off by adding 0.05 and then truncating the result to one place after the decimal point; the Y coordinate is the number of test cases that have the corresponding X value. Table II lists the averages of the HV/VH speedup factors. When we turned off the test machine's 32-K cache, the time taken to complete an exhaustive test on a data set for all four methods was lengthened by factors ranging from about 2.7 to 3.2, but the speed differentials relative to HV/VH trees shown in the figures stayed roughly the same.

TABLE II: HV/VH SPEEDUP FACTOR

	BQT	kD	QLQT
Test1	1.55	2.30	1.16
Test2	1.79	2.13	1.00
Test3	2.12	2.72	0.91

IV CONCLUSIONS

We have shown that HV/VH trees, running against the tide with their use of bisector lists, significantly outperform both BQT and kD trees while using less memory. All three tree structures work similarly by utilizing bounds information to eliminate unnecessary searches, but HV/VH trees have the best strategy summarized below:

1. If there are rectangles intersecting a divider, do not procrastinate. Check to see whether the entire set of rectangles is far enough away from the search window or not. If it is, the number of rectangles left to check becomes smaller; if it is not, the rectangles are likely to be in the vicinity of the search window, making it necessary to check them in the first place. This no-delay policy is virtually free, as the cost of performing the check is buried in deciding which half or halves of the HV space to search for rectangles intersecting the search window.

2. Rectangles stored on a bisector list intersect the divider, and this information, combined with that gleaned from which halves of the HV space and cut space are being searched, makes it possible to use fewer than the four comparisons required by the other three methods to check for intersection between a rectangle and the search window.

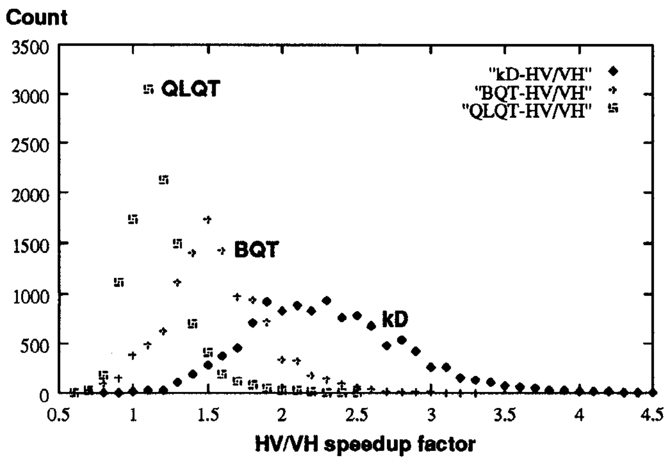

Fig. 6: **Test1**

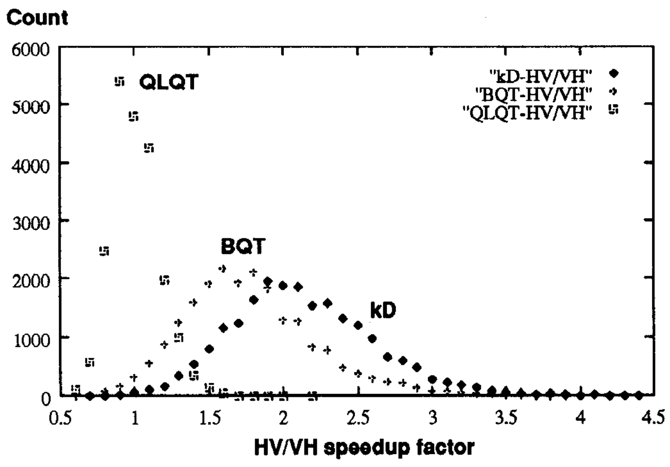

Fig. 7: **Test2**

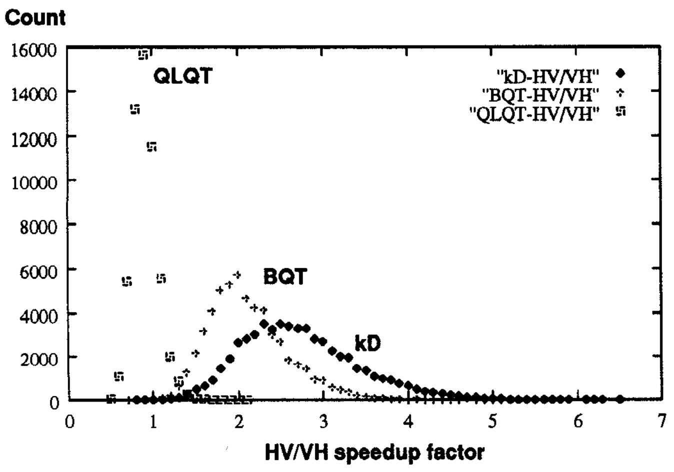

Fig. 8: **Test3**

Rectangles intersecting a divider, normally a problem, is addressed by BQT and kD trees in their own ways and converted into an advantage by HV/VH trees; all use bounds information to speed up region queries. QLQT employs a different philosophy: trade space for time. As Table II shows, QLQT is very competitive with HV/VH trees in terms of speed. However, as shown in Table I, QLQT uses 55% and 56% more memory for **Test1** and **Test2,** where it does not have any speed advantage over HV/VH trees. For **Test3,** where QLQT is 10% faster, it uses 112% more memory! Not to mention the fact that, with the same amount of physical memory, HV/VH trees will be able to process more data, it should be obvious that the small speed advantage that QLQT might sometimes have could easily be erased if the extra memory usage causes severe paging or even swapping. HV/VH trees, with their effective compromise between memory usage and speed, thus represent a significant breakthrough in the development of spatial data structures for efficient region queries.

REFERENCES

[1] Carver Mead and Lynn Conway, *Introduction to VLSI systems*, Addison-Wesley, Reading, MA, 1980.

[2] G. Kedem, "The Quad-CIF tree: a aata structure for hierarchical on-line algorithms," *Proc. 19th Design Automation Conf.*, pp. 352–357, 1982.

[3] J. K. Ousterhout, "Corner stitching: a data-structuring technique for VLSI layout tools," *IEEE Trans. on Computer-Aided Design,* **CAD-3**, 1 (January 1984).

[4] J. B. Rosenberg, "Geographical data structures compared: a study of data structures supporting region queries," *IEEE Trans. on Computer-Aided Design*, **CAD-4**, 1 (January 1985).

[5] R. L. Brown, "Multiple storage quad trees: a simpler faster alternative to bisector list quad trees," *IEEE Trans. on Computer-Aided Design*, **CAD-5**, 3 (July 1986).

[6] L. Weyten and W. de Pauw, "Quad list quad trees: a geometrical data structure with improved performance for large region queries," *IEEE Trans. on Computer-Aided Design*, **8**, 3 (March 1989).

[7] A. Pitaksanonkul, S. Thanawastien, and C. Lursinsap, "Comparisons of quad trees and 4-D trees: new results," *IEEE Trans. on Computer-Aided Design*, **8**, 11 (November 1989).

[8] D. Marple, M. Smulders, and H. Hegen, "Tailor: A layout system based on trapezoidal corner stitching," *IEEE Trans. on Computer-Aided Design*, **9**, 1 (January 1990).