AQUASUN: adaptive window query processing in *CAD* applications for physical design and verification

Michiel De Wilde* mdewilde@elis.rug.ac.be
Dirk Stroobandt† dstr@elis.rug.ac.be
Jan Van Campenhout jvc@elis.rug.ac.be

Ghent University, ELIS Department
Sint-Pietersnieuwstraat 41
9000 Gent, Belgium

ABSTRACT

CAD applications for physical design and verification very often require enumerating all layout objects whose bounding box intersects an axis-aligned rectangular area. A number of multidimensional access methods exist to process such window queries. The performance of some important design and verification algorithms heavily depends on the processing speed of the used access method. For complex layouts, these methods require huge amounts of resident memory to attain this speed.

In this paper, we present a new access method called AQUASUN, which brings a significant query processing performance improvement over other adaptive methods—methods which can cope with a continuously changing layout. These methods generally descend from the database world and are designed to perform the equivalent query in n-dimensional space. Our method is specifically tailored to two dimensions, exploiting 2D optimisations that significantly accelerate window queries within oblong objects like PCB tracks. Furthermore, AQUASUN makes use of an efficient compression technique which greatly cuts down on memory usage.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Graphics*; H.2.2 [**Database Management**]: Physical Design—*Access methods*; E.1 [**Data Structures**]: Trees

General Terms

Algorithms, Performance

*Research Assistant of the Fund for Scientific Research – Flanders (Belgium)(F.W.O.)

†Post-doctoral Fellow of the Fund for Scientific Research – Flanders (Belgium)(F.W.O.)

GLSVLSI'02, April 18-19, 2002, New York, New York, USA.

Keywords

Multidimensional access methods, Rectangle indexes

1. INTRODUCTION AND MOTIVATION

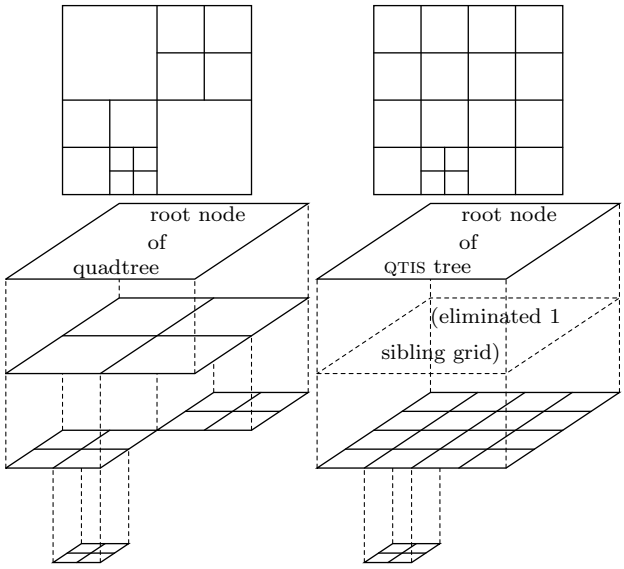
In a PCB or chip layout, a request to find all objects that overlap a given axis-aligned rectangular area is called a *window query*. CAD applications for physical design and verification heavily use such queries. Amongst these are the onscreen layout drawing procedures and some important algorithms like automatic placement and design rule checks.

Fast processing of window queries is required for interactive zooming to keep the application responsive to the user. Furthermore, design rule check algorithms spend substantial processing time issuing a large amount of queries with small windows, typically as many as the number of objects the layout contains. Therefore, it is of the utmost importance that the window query processing method be as fast as possible.

In general, a window query processing method which operates directly on a collection of arbitrarily shaped objects is not feasible: iterating an intersection check over the entire object collection would simply be too slow. Furthermore, taking the detailed object shape into consideration is too time-consuming; the use of a simpler shape like the bounding box of the object is much faster. Only when a bounding box (just called box hereafter) overlaps the window, further inspection of the corresponding real boundary of the object needs to conclusively confirm or deny window intersection. This approach works fine in practice and is widely accepted as the most feasible method of work.

To reduce the search scope, window query processing methods require a special data structure containing all boxes, called a *box index*. The development of a box index and the corresponding searching algorithm is not an easy task. The plethora of existing linear searching methods is of no direct use, as generally no total ordering of boxes exists which preserves spatial proximity. Luckily, a number of methods specially targeting window query processing have been developed: Gaede et. al. [5] and Ahn et. al. [1] present some broad surveys. As some methods are suited to process the equivalent of window queries in n dimensions, they are commonly referred to as *multidimensional access methods* (MAMs).

In methods used in older CAD applications, the box index had to be rebuilt from scratch after each layout change



(a) quadtree (b) QTIS tree

Figure 1: Quadtree and QTIS tree examples

(when it was needed). However, some newer layout verification-correction algorithms continuously issue layout changes during operation, invalidating the original index. Hence, newer *adaptive* methods must be able to adapt the box index to layout changes.

Most current adaptive methods are designed to handle an arbitrary number of dimensions. As a result of their generality, they are suboptimal with respect to some typical 2D CAD features, such as searching in collections of oblong objects like PCB tracks.

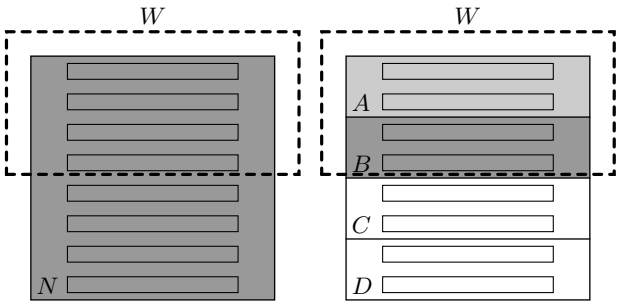
Our method, called AQUASUN,[1] specifically optimises 2D window query processing for use in (interactive) CAD applications (point and enclosement queries are optimised too, but they are not handled in this paper). In addition, it uses a simple but effective compression technique to cut down on the resident memory requirements, a scarce resource which is too greedily consumed by the existing indexing methods.

In the next section, starting from a generic 2D MAM system, we extend the overall structure of its box index to that of AQUASUN. In section 3, we discuss how a layout is represented in this structure. With this knowledge, an efficient searching algorithm is developed in section 4. In section 5, we deal with structural maintenance of the box index to assure permanent optimal query processing performance. In section 6, our contribution is related to prior work. Finally, in section 7, real-life performance is examined.

2. DESIGNING THE SEARCH INDEX

The object box index for any 2D MAM is a data structure consisting of *nodes*, each associated with a set of nearby boxes. This data structure is set up in such a way that as few as possible sets need inspection during a window query. For instance, if a window happens to enclose a point common to *all* boxes associated with a node, we know that *each* box in the set will overlap the window. On the other hand, if the global bounding box of the *entire* set happens to be disjoint

[1]Adaptive QUAdtree Skipping UNderpopulated levels



(a) no oblong children yet (b) oblong children of N

dark nodes require inspection
light nodes are contained by W
white nodes are disjoint of W

Figure 2: Oblong nodes are beneficial

of the window, *none* of the boxes will overlap it.

In both cases, the node can be handled as a whole with respect to the query, which speeds up processing enormously. The slow case is then limited to situations similar to the one where the set of object boxes associated with a node is partially disjoint of the window. In this case, all boxes have to be individually inspected for window intersection, which is a slow linear operation.

The technique of creating different nodes of nearby boxes can be recursively applied to the set of boxes that a node contains, by creating smaller nodes that are attached as *children* to their *parent*. In this way, a tree structure is created. All major non-hashing 2D MAMs operate this way; only the precise way in which boxes associate with nodes and in which the tree structure is managed differs.

2.1 Quadtrees

AQUASUN's box index is most easily explained starting from a generic structure for regular planar decomposition: the quadtree [12] (figure 1(a)). A quadtree is a tree structure in which each node represents a square in the plane. Each internal node has four children, constructed by cutting the node both horizontally and vertically in half.

When quadtrees are used as a box index in the most straightforward way, boxes are associated with the smallest enclosing node. This is called the *tightest enclosure* association rule. Window queries can then be processed with a simple procedure: the tree is traversed in a top-down fashion, at each node only descending to children that overlap with the window. When the window fully encloses a given node, all boxes associated with the node and its offspring can be directly reported without further inspection. All boxes associated with nodes intersecting the window boundary have to be individually inspected.

2.2 Reducing the tree height

A major disadvantage of the naive use of quadtrees is their unnecessary large tree height. For example, suppose that a large collection of boxes have dimensions some 100 times smaller than the bounding box of the entire layout. Then, most boxes would be at depth $6 = \lfloor \log_2 100 \rfloor$ in the quadtree. As a result, processing a window query would cause significant tree descending and ascending over the mostly empty intermediate-depth nodes, an overhead that we want to avoid.

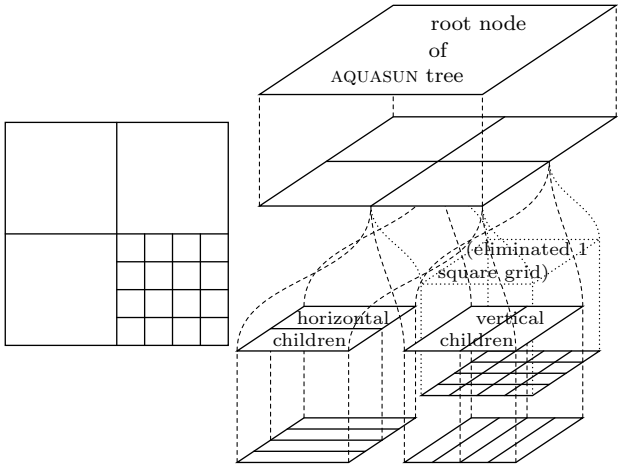


Figure 3: Example of an AQUASUN tree

To overcome this, De Pauw's [3] QTIS[2] structure allows to eliminate a node's four children by directly attaching it to its sixteen grandchildren. This locally decreases the tree height by one (figure 1(b) shows an example). The few boxes which may have been associated with the removed nodes are re-associated with their common parent.

Since in this way multiple offspring levels can be eliminated, in general any internal node contains a square grid of 2^{2n} children, for any positive integer n.

2.3 Oblong nodes

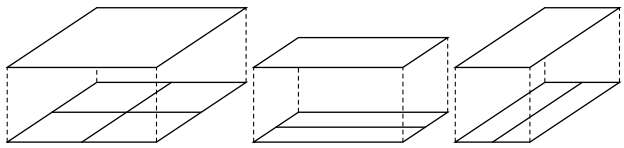
Like current box indexes for adaptive MAMs, quadtrees (including QTIS) don't cope well with very oblong objects. Specifically, if many of them are packed together (e.g., the tracks on a PCB layout) query performance decreases significantly. To illustrate this problem, figure 2(a) shows a set of oblong boxes associated with some node N. This node will be large, because of the large width of the objects involved, and contains many boxes, because of their small height. A window query W that only encloses the upper half of N has to individually inspect the long list of *all* boxes.

If N would be able to give birth to four horizontally oblong child nodes, all boxes would be re-associated with these children, as shown in figure 2(b). Re-issuing the same query W, only the two boxes associated with node B, which intersects the window's edge, have to be individually inspected. Both boxes associated with node A can be directly included into the query result, since A is fully window-enclosed.

To structurally incorporate oblong nodes, we extend QTIS with optional oblong node subtrees, resulting in the AQUASUN box index structure (figure 3). Here, a square node is allowed to have an optional single-column grid of 2^h horizontal children and another optional single-row grid of 2^v vertical children, both covering the full area of the node. This grid may possibly coexist alongside the 2^{2n} square children. Both oblong node types are created by cutting their parent's rectangle into equal parts using cuts in one direction. Oblong nodes can have even more oblong children themselves, by further cuts in the same direction. We will call *square*, *horizontal* and *vertical* the three different *shape types*. Items of the same shape type are called *shape-corresponding*.

Siblings of a given shape type are always organized in a regular grid, albeit a single-row or single-column one for oblong nodes. We will keep them together in a data structure representing the grid ordering, called a *sibling grid*. From the viewpoint of the common parent of the nodes, we will also call this structure a *child grid*. A sibling grid is called *minimal* if it contains the lowest possible number of nodes, given its shape type: 4 square nodes or 2 oblong nodes (figure 4). Furthermore, whenever we mention the *shape type* of a sibling grid, we are actually referring to the shape type of its nodes.

[2]QuadTrees With Internal Storage



(a) square (b) horizontal (c) vertical

Figure 4: Minimal child grids

3. BOX-NODE ASSOCIATION RULES

The *tightest enclosure* association rule treats small boxes positioned on the border between two adjacent nodes suboptimally.

For instance, if some very small box happens to contain the center of the root node, it can be fully enclosed only by the root node, as it would be outside or crossing the edge of any other node. Therefore, we would have no choice but to associate this box with the root node, which is intended to store much larger boxes. Since the boxes associated with the root have to be individually inspected for *every* window query, this approach is clearly not optimal. We would rather like to find some way to associate the small box with some node deeper in the tree, as smaller nodes have lower probability to need individual box inspection.

Since the index performance relies on the grouping of nearby rectangles, we cannot entirely dispose of the rule that a box is associated with its smallest enclosing node. Therefore we will only weaken this requirement, imposing a box-node association rule commonly used in MAMs:

> **Association rule 1:** Nodes can only associate with boxes whose lower left corner (LLC) is located in the node.

In this way, a kind of box proximity is still guaranteed only using a simple reference point constraint. Rule 1 by itself only identifies the possible candidate nodes to be associated with the rectangle. Size is not used, but will be incorporated in the following rules. Since boxes can now protrude beyond the top and right edges of the associated node, our method falls into the *overlapping regions approach* category of MAM taxonomy [11].

With respect to node association, leaving box size unconstrained would cause some problems if we attempt to devise a top-down query processing algorithm. To see what would happen, consider figure 5, which shows part of a sibling grid at a given tree level. The indicated window query does not have to consider the white nodes at all: the constraint on the LLC of boxes associated with them or their offspring prohibits any window overlap. Searching the nodes of the dark window-intersecting area I is obviously inevitable. The nodes in region $L \cup N$ and their offspring, however, could contain large boxes overlapping the window as well, since at this stage we have no indications as to their sizes.

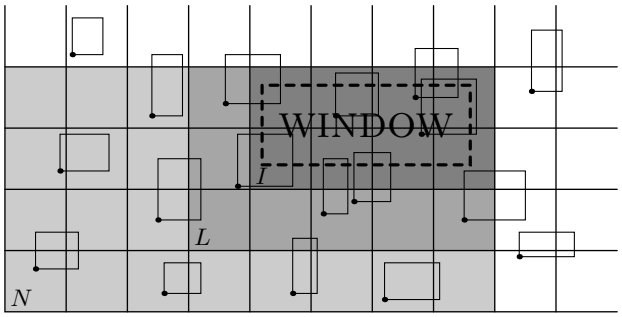


Figure 5: Rectangle size limitation

We want to minimize the area to be inspected for performance reasons. Entirely forbidding boxes in $L \cup N$ to intersect I would be too strict, as this would fully reimpose the suboptimal tightest enclosure rule. The best we can do is to limit the extra nodes to be inspected for boxes overlapping with I to the L *region* of one node wide. This can be done by putting limits on the height and width of the boxes associated with any node. To this end, we impose rule 2:

> **Association rule 2:** Nodes can only associate with boxes the dimensions of which do not exceed the corresponding dimensions of the node.

Furthermore, as a node can potentially have three different child grids, we want to ensure that boxes of a given aspect ratio end up in the proper grid:

> **Association rule 3:** Oblong nodes can only associate with boxes which are oblong in the same direction. In addition, the longer side of the box must be larger than half of the longer side of the node.

The fourth and final rule exploits the obvious fact that smaller nodes have lower probability to require individual box inspection, and aims at locating boxes at the lowest possible level:

> **Association rule 4:** A smaller node takes precedence over a larger one regarding association with a given box.

It can be shown that all rules jointly cause every box to associate with exactly one node. When a new box is to be inserted into the index due to layout changes, its associated node can also easily be found. Starting at the root node, it can be shown that only a single descending path composed of nodes fulfilling rules 1–3 exists. As the deepest node on this path will be the smallest one, it will also comply with rule 4 with respect to the box.

4. WINDOW QUERY PROCESSING

Given our new association rules, we can now devise an efficient recursive window query processing algorithm. The root node is interpreted as the single-node sibling grid R. The following function OVERLAP_SET(R, W) returns the set of boxes that overlap a given window W:

```
function OVERLAP_SET(G: sibling grid, W: window) {
    result ← ∅
    N1 ← nodes of G enclosed by W
    foreach(n ∈ N1) {
        result ← result ∪ all boxes associated with
                  all nodes of the subtree rooted at n
    }
    N2 ← all nodes of G intersecting
        the boundary of W ∪
        all nodes in the L region
    foreach(n ∈ N2) {
        foreach(b ∈ BOXES(n)) {
            if OVERLAP(b, W) {
                result ← result ∪ {b}
            }
        }
        foreach(g ∈ CHILD_GRIDS(n)) {
            result ← result ∪ OVERLAP_SET (g, W)
        }
    }
    return result
}
```

5. TREE MAINTENANCE

The precise composition of an AQUASUN tree does not affect the ability to insert new boxes and the correctness of the window query processing algorithm. However, processing performance depends heavily on the specific tree composition. For optimal performance, we should be able to adapt the tree construction to the layout at hand. To this end, a notion of optimality of the tree composition of is derived. Furthermore, a method is developed to permanently hold on to this optimality while dealing with layout changes.

5.1 Optimality of tree composition

As a first step towards optimal searching performance, we can minimize the number of individual box inspections for a random query. To this end, for each box the tree should contain the smallest structurally conceivable node obeying association rules 1–3 with regard to the box.

It is possible to devise a tree structure whose nodes only have minimal child grids (figure 4). Because of this, we can assure that for a given layout both dimensions of each node will be smaller than twice the corresponding dimensions of any of its associated boxes. In this way, each box is associated with the smallest conceivable node.

However, other effects come into play. On the one hand, in the OVERLAP_SET searching algorithm the iteration over N_2 causes a depth-first tree traversal. A breadth-first traversal is not feasible, as that would require huge memory amounts. Because of this and the 2D tree nature, it is not possible to place subsequently considered nodes in nearby memory. Jumping from one node to the next is therefore coupled with some overhead—a page fault in the worst case.

On the other hand, the inspection of a single node can be implemented as a fast iteration, which typically only accesses a single virtual memory page. Therefore, if the nodes of a sibling grid are associated with very few boxes, overall performance can be improved by eliminating this grid if it decreases the total number of nodes. After elimination, the few boxes that were associated with the removed nodes are re-associated with their common parent, increasing their individual inspection probabilities.

An implementation target specific *box threshold* c (typically 15–75) can be derived on the total amount of boxes associated with the nodes of one sibling grid. If this value drops below c, the performance increase coming from a node-decreasing grid elimination will outweigh the increased in-

spection probabilities. It can be shown that c is *not* dependent on the sibling grid size [4].

In summary, we formulate the optimal composition of a tree as follows:

> The AQUASUN box index tree is optimally composed when the number of nodes is minimal under the following restriction:
>
> If we would—in any possible way—add a child grid to a leaf node or insert a sibling grid between an internal node and a non-minimal child grid of the latter, the box threshold of the new sibling grid would not be reached after box re-association.

5.2 Holding on to optimality

In an empty layout, the optimal tree structure is obviously a sole root node that does not associate with any boxes. When the layout is changed, boxes are inserted or removed by imposing or withdrawing the box-node association that obeys the rules.

As AQUASUN applies *regular* space decomposition, sibling grid creation and elimination can be implemented very efficiently. Still, such an operation is much more costly than a simple box insertion or removal. When the number of boxes that are associated with the nodes of some grid oscillates around the box threshold, the tree will need unduly repeated restructuring. To avoid this, we will allow some small hysteresis $c^- \leq c \leq c^+$ with regard to the box threshold c. This bounded deviation from 'real' optimality is called *practical optimality*.

We want to continuously hold on to practical optimality. After every box insertion or removal this optimality can be disturbed, which needs to be verified and corrected if necessary. Since this check occurs often, it must be very low-overhead.

Sibling grid insertion. To verify whether insertion of some new sibling grid is needed, we need to know the amount of boxes that would associate with the new grid nodes if we inserted it. To this end, we attach to each node n some named *node-box count* values that amount to the total number of associated boxes fulfulling a particular condition:

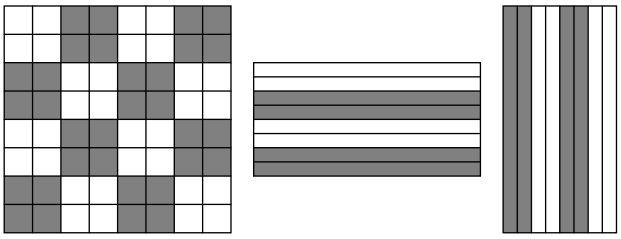
n: node of size $w_n \times h_n$
b: box of size $w_b \times h_b$ associated with n

NAME	ATTACHED TO	CONDITION
square	sq.	$w_b \leq w_n/2$ and $h_b \leq h_n/2$
horizontal	sq. and hor.	$w_b > w_n/2$ and $h_b \leq h_n/2$
vertical	sq. and vert.	$w_b \leq w_n/2$ and $h_b > h_n/2$

It can be shown that when any of these counters reaches c^+, we need to give n a new child grid whose shape type corresponds to the name of the counter [4].

The new grid will be a minimal one (figure 4) if the node does not already have a shape-corresponding child grid. Otherwise, the new grid must be inserted between the node and the existing child grid. The correct node size of the new grid can be inferred from the dimensions of the c^+ boxes that are referred to by the overflowing counter; more precise details on this are out of the scope of this paper [4]. In any case, after insertion some of the referred boxes will be re-associated with nodes of the new grid, decreasing the overflowing counter value below c^+.

Furthermore, a grid also needs to be inserted if doing so would decrease the total number of nodes. To this end, we attach a *cluster counter* grid to each sibling grid. As shown in figure 6, each cluster counter refers to an adjoining group of four square or two oblong nodes. The cluster counters amount to the number of boxes associated to the clustered nodes plus c^- for each child grid of these nodes.



(a) square (b) horizontal (c) vertical

Figure 6: Clustering adjoining nodes

To each grid, an additional *insertion count* value is attached that amounts to the number of cluster counter values below c^-. It can be shown a grid insertion exists that decreases the total number of nodes when the insertion counter of some grid exceeds the fraction $\frac{1}{16}$ (square grid) or $\frac{1}{4}$ (oblong grid) of the number of nodes in that grid [4]. If this is the case, we need to insert a new grid—between the current grid and its parent—whose nodes are of double size in both dimensions (square grid) or only in the shorter dimension (oblong grid).

Sibling grid elimination. When the elimination of some sibling grid G is needed, it can be shown its nodes will only have shape-corresponding child grids [4]. If G consists only of leaf nodes, elimination is straightforward as G is easily removed and only some box re-associations are needed.

Otherwise, elimination is somewhat more elaborate. As an elimination example, if figure 1 represented two AQUASUN trees, tree 1(b) would result from 1(a) after removing the sibling grid at depth 1. In general, a sibling grid G is eliminated as follows:

1. Each non-minimal child grid g of G is equally divided in 4 square or 2 oblong pieces, which are attached as child grids to the nodes of a fresh shape-corresponding minimal grid replacing g.
2. To nodes of G without children an empty shape-corresponding minimal child grid is attached. (figure 4)
   After this step, *each* node of G is parent of a single shape-corresponding minimal grid.
3. All boxes associated with the nodes of G are re-associated with their common parent.
4. All minimal child grids of nodes of G are collected into a new—finer—grid, which replaces G.

After layout changes, we need to verify whether such elimination is necessary somewhere in the tree. To this end, we attach to each sibling grid the *grid-box count* value, which amounts to the sum of (1) the shape-corresponding node-box counter value of its parent, (2) the total number of boxes associated with the nodes of the grid, and (3) only for a square grid: c^- for each oblong child grid.

It can be shown that a sibling grid should be eliminated if its grid-box counter value drops below c^- [4]. Still, we are only allowed to do this if the total number of nodes would decrease as a result.

To this end, to each sibling grid we attach the *elimination count* value, which amounts to the number of grid nodes

having shape-corresponding children. It can be shown that when this counter value exceeds the fraction $\frac{3}{4}$ (square grid) or $\frac{1}{2}$ (oblong grid) of the number of grid nodes, eliminating the grid will decrease the total number of nodes [4].

Counter optimisation. All counter values can be incrementally updated. The threshold value by which any of our counters overflows is already known at the point where the grid or node containing the counter is created. Because of this, we can already subtract this threshold from the counter value when it is first initialized. In this way, each counter only needs a fast zero value check after it has changed.

When a fully built-up layout is only incrementally changed, the overhead of verifying practical optimality is very low: the average box insertion/removal will only require between 2 and 3 counter values in the *entire* box index to be incremented/decremented and compared to zero.

6. RELATION TO PRIOR WORK

6.1 Pre-existing work

The techniques of regular spatial decomposition and sibling grid elimination are derived from quadtrees [12], and QTIS [3], respectively. However, QTIS is not adaptive and its conditions for sibling grid elimination are much too strict to be useful.

A huge amount of other spatial indexing techniques exist, and since enumerating them results in a tutorial paper in its own right, we will not mention them here. On this subject, we are much obliged to the surveys by Gaede et. al. [5] and Ahn et. al. [1] for helping to understand the problems connected with spatial indexing.

6.2 Original work

Low-overhead structural optimality. AQUASUN is the first indexing structure employing regular spatial decomposition with sibling grid elimination whose tree representation is fully dynamically maintainable. This is done in such a way that very few data items need to be inspected or changed to permanently assure efficient searching performance.

Oblong nodes. Our method uses a tree structure whose main nodes are square, but with oblong subtrees (grafted onto square nodes) whose nodes are increasingly oblong with rising tree depth. As a result, our method can cope very well with layouts containing objects whose aspect ratios are spread within a wide range. This optimisation is *not* efficiently extensible to higer dimensions, yet for two dimensions it proves to be a fine solution.

Reduced memory needs. Boxes associated with a node need to be stored in some way. We have chosen to organize them in chains of fixed-size arrays. Each array is preceded by a small header containing the coordinates of the LLC of its associated node. The arrays themselves store offsets from this LLC to the four boundary coordinates of each box. In addition, a reference to the real object is memorized.

The size and position of boxes which can associate with some node is restricted by association rules 1 and 2. Therefore, the number of bits necessary to represent their coordinate offsets is limited as well. We obtain substantial data reduction if we exploit this limitation, by providing different array types using 8, 16, 32 and —perhaps— 64 bits to represent the coordinate offsets. Using this simple compression technique, an entire AQUASUN box index can occupy even less memory than the equivalent full-precision array of all box coordinates.

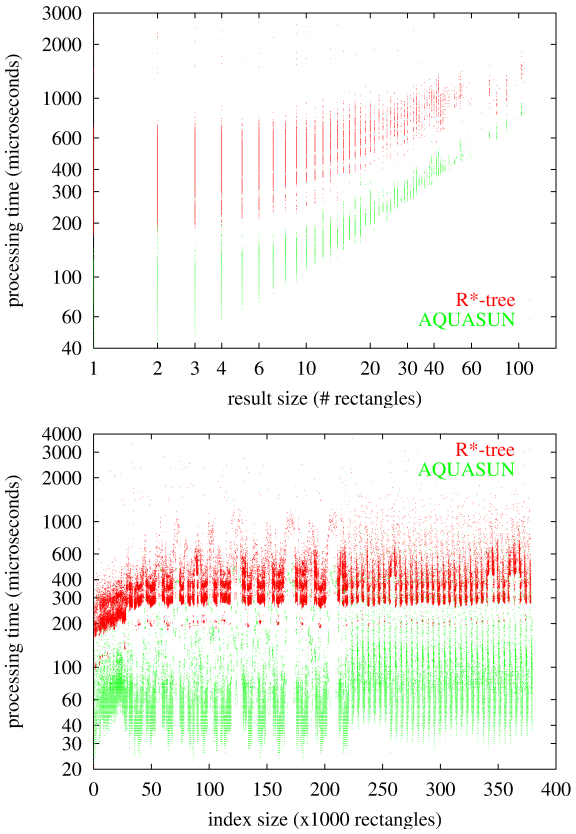


Figure 7: Window query processing performance

7. PERFORMANCE

We have compared the searching performance of AQUASUN to the R*-tree [2] box indexing method, which is based on irregular planar decomposition. Although the original R-tree has been conceived almost two decades ago [6], its more recent variants are currently widely used in the field [8], and consititute the prime box indexing methods of most software tools for spatial indexing, e.g., libGiST of UC Berkeley (v2.0: April 2000) [7]. We used the original C implementation of the R*-tree by Seeger [2], because it turned out to be the fastest implementation available, yet the least versatile as to its usage. We also wanted to include a performance comparison with the Segment R-tree (SR-tree) [10], which is currently believed to be the fastest adaptive MAM that is suitable for our purposes [9]. Unfortunately, as currently only a (slow) java implementation of the SR-tree seems to be available, including it in our benchmarks would be unfair.

We collected our benchmarking data from *magplot*, an application that converts *Magic* VLSI layout files into a printable PostScript file. The major part of this conversion consists of a stepwise insertion of all chip layout objects into a box tree. At each layout object insertion, the box tree needs to be searched for all objects that intersect the new one. The intersecting objects are then removed from the index, cut into pieces totally disjunct from or enclosed by the new object, and the pieces are reinserted into the box index. This process causes a lot of window queries with small

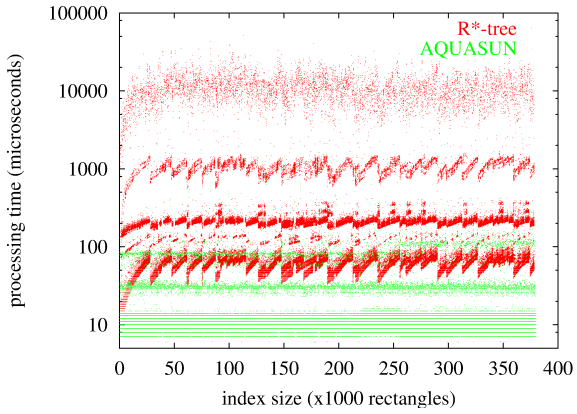


Figure 8: Insertion performance

windows, which is typical for layout applications such as a design rules check.

The benchmarks were carried out on a Pentium II 233MHz (96MB RAM) using gcc 3.0.3 with maximal optimisation and involve the conversion of a rather complex RSA decoder chip layout. Figure 7 shows that the searching performance is very dependent on the size of the result, but almost not on the index size. For larger search results (larger windows), performance is asymptotically equal for both structures. However, very small search results occur much more often in practice, and here AQUASUN is about 5 times faster than the R*-tree implementation. In figure 8, the processing time for the insertion of one box into the index is shown. The results show that AQUASUN is about 8 times faster than the R*-tree; this is most likely a result of the low-overhead optimality checking mechanism that we detailed in section 5. As to memory usage, almost all of the original 32-bit box coordinates were automatically truncated to 8 bits without any loss of precision.

As a final performance remark, the use of AQUASUN results in a significant reduction in wall clock running time: while the duration of the entire conversion process is 6'33" for the R*-tree, the use of our method reduces this to 2'08", which is faster by a factor of more than 3.

8. CONCLUSION

In this paper, we have introduced AQUASUN, a new method to process window (and enclosement) queries on PCB and chip layouts. The method maintains a tree structure, the *box index*, which stores the bounding boxes of all layout objects. The boxes in this index are organized according to a chosen set of rules. As a result of our choice, we have been able to derive both an efficient searching algorithm and a technique which significantly cuts down on memory requirements.

In addition, we have made provisions to efficiently cope with layouts containing very oblong objects, like the tracks on a PCB. In this case, searching performance is guaranteed by a controlled grafting onto the box index of subtrees in which both dimensions are unequally treated.

Furthermore, our box index is fully adaptable to layout changes. To this end, we use a number of reference counters which are updated and checked with *very* low overhead at every object addition or removal: on average, only 2 or 3 counter values in the *entire* box index need updating. When a counter value reaches zero, the tree may need restructuring to accelerate forthcoming window queries. Each restructuring operation by itself only makes very incremental adjustments to the box index structure and can hence be performed with low cost.

We have field-tested our method in a VLSI tool that converts a chip layout into PostScript; AQUASUN was compared herein to a fast R*-tree implementation and our test results show a reduction of the wall clock processing time by a factor of more than 3.

9. ADDITIONAL AUTHORS

Peter Verplaetse (`pvrplaet@elis.rug.ac.be`) kindly provided the C++ code of *magplot* and the RSA decoder layout.

10. REFERENCES

[1] H. K. Ahn, N. Mamoulis, and H. M. Wong. A survey on multidimensional access methods. Lecture COMP630c, "Spatial, Image and Multimedia Databases", University of Science and Technology, Clearwater Bay, Hong Kong, Oct. 1997.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, May 1990.

[3] W. De Pauw. *(in Dutch) Datastructuren voor grafische informatie bij CAD (Data structures for graphical information in CAD)*. PhD thesis, Ghent University, Belgium, ELIS D9144, 1991–1992.

[4] M. De Wilde. The internals of the adaptive box indexing method AQUASUN (in progress). Technical Report PARIS 02-01, Ghent University/ELIS, 2002.

[5] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[6] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 47–54, 1984.

[7] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *International Conference on Very Large Databases*, pages 562–573, Sept. 1995.

[8] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using r-trees: Breadth-first traversal with global optimizations. In *International Conference on Very Large Databases*, pages 396–405, Aug. 1997.

[9] G. Kollios, D. Gunopulos, V. Tsotras, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Transactions on Knowledge and Data Engineering*, Sept. 2001.

[10] C. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multidimensional interval data. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 138–147, June 1991.

[11] B. Ooi, R. Sacks-Davis, and J. McDonell. Spatial indexing by binary decomposition and spatial bounding. *Information Systems Journal*, 16(2):211–237, 1991.

[12] H. Samet. The quadtree and related hierarchical data structure. *ACM Computing Surveys*, 16(2):187–260, 1984.