An Efficient Routing Database

Narendra V. Shenoy
Synopsys Inc.
700 E. Middlefield Rd.
Mountain View CA 94043
nshenoy@synopsys.com

William Nicholls
Synopsys Inc.
1907 NW Cornelius Pass Rd.
Hillsboro OR 97124
nicholls@synopsys.com

ABSTRACT
Routing is an important problem in the process of design creation. In this paper, we focus on the problem of designing a database for the non-partitioned routing problem. New technology libraries describe constraints that are hard to manage in grid-based approaches to the routing database. While general region query based data-structures have been proposed, they typically suffer from speed problems when applied to large blocks. We introduce an interval-based approach. It provides more flexibility than grid-based techniques. It exploits the notion of preferred direction for metal layers to manage the memory efficiently. It supports efficient region queries. We finally present a comparison study for real industrial designs on this database.

Categories and Subject Descriptors
J.6 **[Computer-Aided Engineering]**: Computer-aided design

General Terms
Algorithms, Design

Keywords
Physical design, routing, database.

1 INTRODUCTION
Managing data in any software tool is a challenge when the data sizes become large. This is particularly true in the routing problem. Routing has been a subject of fertile research, particularly from the algorithmic point of view. Less emphasis on the database has been found in the research community. In this paper, we investigate the key issues that arise in the design and management of a routing database. We propose an efficient data-structure that meets the demands of new technologies. In the rest of Section 1 we discuss the motivation, requirements and previous work in this field. In Section 2, we present our approach and discuss relevant issues that a routing database must address. Section 3 deals with experimental data that shows the merits of our approach. Section 4 provides some concluding remarks.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

1.1 Motivation and previous work
We believe that the design and implementation of an efficient routing database is an interesting and challenging problem for the following reasons. With the recent advances in combining placement and synthesis[21][22], the next focus on design closure will be in the area of routing. In recent technologies, capacitance prediction using routing topology (which is done during placement today) is insufficient. Post-route timing results will be less predictable unless a feedback loop to a timer and extractor is present during routing. This means (assuming a single process) that the memory available for routing will be reduced. A flat database will make this simple to manage. Another compelling reason to investigate new routing database representations is the lack of flexibility in grid-based schemes. Newer technology libraries describe complex pin and obstruction geometry. Abstracting such structures onto a fixed grid representation means that a certain part of the solution space is lost. Finally, recent technology constraints such as wire widths and wire spacing vary across layers. Grid based schemes often require these numbers to be multiples of a fixed number so that a common grid can be created for all layers. This can lead to inefficient use of routing resources.

A routing database requires the following essential ingredients to be successful. It should support fast region query, as the search engine will make millions of queries. A classic search such as Dijkstra's [1] approach or variants such as Lee's [2][3] approach operates in the following fashion. A heap is used to maintain the current frontier of the search. The least cost point is popped off the heap and its neighbors are examined for possible expansion. It is this step that needs fast access to the database. An expansion can be prohibited due to metal geometry or obstructions at or near the expansion point. In addition it should be easy to modify and manipulate the database during routing. It should be memory efficient. The simplest routing database is the grid-point database. The area under concern is populated by a set of grid-points. Each point stores the information required to respond to the queries made during expansion. Each point also stores memory for the cost of the search and for backtracking.

An alternative routing database relies on data-structures tailored for efficient region queries. There is a large body of literature in this area[4][5][6][7][8][9][10][11][12]. Finkel and Bentley[4] develop quad-trees as a data structure for higher dimensional query. Later Bentley[5] proposes the kd-tree as an improvement over the quad-trees. The book by deBerg et al.[13] provides an excellent reference to this literature. Kedem[6] describes the use of quad-trees with bisector lists for storing/retrieving objects

using minimal bounding boxes. Ousterhout[7] designs the corner-stitching technique for fast incremental local operations. Rosenberg[8] conducts a detailed study of data structures that support area query in two dimensions. Later work by Brown[9] eliminates bisector lists in the quad-trees. Fontayne and Bowman[10] describe a radix hash tree for region query. Marple et al.[11] extend corner-stitching to manage trapezoidal objects in the database. Lai et al.[12] introduce VH/HV-trees for region queries. If we choose to design a database relying on any of these structures, the following issues need to be kept in mind:

- Data-structures that rely on trees can become quickly unbalanced as objects are inserted. Explicit balancing schemes need to be designed and implemented. This can add a significant level of complexity to the implementation.
- A routing database will contain millions of rectangles. Any structure whose memory growth is not linear in the input size will be hard pressed to handle large designs. Data structures such as corner-stitching can cause severe fragmentation are also ill suited.
- A common approach to speed up queries is to maintain shadows (regions of effect in the neighborhood) of metal objects along with the original geometry so that all region query can be reduced to a simple point query.

The ability to implement efficient area based search algorithms on the database is also important. In this context, we can find various approaches in [2][3][15][16][14]. Lee's approach and Rubin's variant [2][3] implement a search on a grid graph. These often require the routing problem to be partitioned. We do not favor the partitioning approach because it leads to a local view of routing; making global properties of a design such as timing, and antenna rules harder to determine. Line search techniques[15][16] provide a mechanism to speed up the search at the expense of sub-optimal solutions. Both these techniques rely on a grid graph that restricts the search space for a router. The grid-graph coincides with the grid-point database. This can be expensive in terms of memory as shown by Cong et al.[17] and Hetzel [14]. One approach to solve this is to ignore the grid and build the search graph on the fly[17][18][19]. Memory is saved but time is consumed to construct the graph for every search. Hetzel[14] proposes a data-structure relying on intervals for routing. He describes a fast search engine that can deal with large graph sizes. Since we are seeking a balance between performance, capacity and more flexibility than a grid based solution, our approach is strongly motivated by his work. In [14] the author focuses on a new search algorithm for large grid graphs. We design a database that coincides with the search graph required by him. Our main contribution is to extend the graph on which Hetzel describes his search to support all the requirements of a routing database.

1.2 Terminology

The routing problem is defined on a set of planes, numbered from zero (layer zero referring to the metal layer closest to the substrate). Each plane has a preferred direction of routing which alternates between horizontal and vertical directions. A layer k has a default metal width denoted by mw_k. The spacing between any two metal geometries is required to be at least sp_k units apart. Often metal objects with large geometries may require other objects to be more than sp_k units apart (width based spacing). So the spacing rule can actually be a function of the size of the objects. Each adjacent layer pair will have a set of vias used for connection. The geometries of the vias can be often complex with metal overhangs and can also be asymmetric. Each adjacent layer pair has a set of default vias. These are used to connect default metal on adjacent layers. Vias can be placed on top of one another to jump across multiple layers. The actual via invoked to implement this is called a stack via whose geometry need not be the same as a combination of default vias. For the sake of the discussion we will assume that a default via has the same shape as a square piece of default metal on the appropriate layer. Extending the database to other vias is conceptually straightforward, but requires a fair amount of book-keeping.

The routing database is populated by three kinds of objects.

1. Objects such as pins and pre-routes that need to be connected by the router. For sake of discussion, we only consider **pins**.
2. Objects that appear as **obstructions** to the router. The router needs to satisfy spacing constraints with respect to these objects.
3. In addition we have to be able to annotate **route geometry**. This includes routes with different widths and different vias.

Pins and obstructions are input constraints and hence fixed for a routing problem instance. Route geometry is constructed by the router and hence is changeable. We call a route in the non-preferred direction of a plane as a jog.

2 OUR APPROACH

We have mentioned the fact that grid-point based schemes are expensive in terms of memory requiring a partitioning of the design. General point query structures are too expensive in terms of speed. We shall demonstrate both these facts in the experimental section. We naturally look for an approach that provides the middle ground. We define a data representation that depends on intervals. From an intuitive sense, an interval captures more than a grid point but less than a rectangle that describes the true geometry.

Consider the data on a given plane for routing. Most technologies define a preferred direction for routing. This means that a large set of metal geometry will run parallel to the preferred direction (except in the case of lower layers such as layer zero and layer one possibly). The database is designed to make use of this natural bias. We construct a set of gridlines that run the span of the chip in the preferred direction. The spacing between the gridlines can be non-uniform. Since an interval is constrained to lie on a gridline, its co-ordinates in the direction of the gridline can take on any value (i.e. are not restricted to be grid points as in the case of a grid based system).

2.1 Interval database

Consider a plane in the routing database with a preferred routing direction. We use the term "xy" to denote the axis of the preferred direction and "yx" to denote the axis of the non-preferred direction for a plane. We create a set of gridlines parallel to the "xy" axis, whose "yx" co-ordinates are described by the user. The gridlines as managed as an array. Each gridline holds a set of intervals. There are two ways to store the intervals.

- If we choose to distinguish between the impact of different metal geometries at a point, we can use an interval-tree to represent the set of intervals. An interval tree requires linear memory and provides logarithmic response to query. However the tree requires explicit balancing.
- If we choose not to distinguish between the impact of different metal geometries at a point, we can use a simpler representation. In the case of the routing problem this distinction is not necessary and we prefer this approach. See Section 2.2 for further details.

The intervals are **non-overlapping** and maintained in a tree such as a skip-list[20] or a red black tree[1]. Skip-lists are an interesting alternative to balanced trees, as they do not require explicit balancing. Preliminary experiments show that red-black trees are better than skip-lists by a constant. The main advantage of a skip-list is that you get a pointer to the "next" interval for free. Hence skip-lists are more efficient for neighborhood query at a point. Skip-lists use randomization to provide amortized O(log(n)) insertion, deletion and query, where n is the number of intervals in the skip-list. The intervals have a next and previous pointer, which make traversal along a gridline very efficient.

Let us now focus on the query response time. Managing the gridlines as a static array of size N provides O(log(N)) access for a gridline with a relationship (equal, less than, greater than) to a given "yx" value. Querying for the object at a point in a plane p takes O(log(N) + log(n)). Obtaining all the objects in a region requires a series of logarithmic queries. If n is the maximum number of intervals in any gridline, and the query range intersects G gridlines, and the number of intervals in the region is k, it is easy to show that the complexity of a query is O(log N + G log n + k). Neighborhood queries of a point (p, p_{xy}, p_{yx}), where p is the plane, p_{xy} and p_{yx} are the co-ordinates in the "xy" and "yx" axis respectively, are accomplished as follows. Let g be the index for the gridline at p_{yx} on plane p. The next and previous intervals can be efficiently obtained. The interval in the gridline in the positive yx direction is obtained by querying the gridline at $g+1$ with location p_{xy}. Similarly the neighbor in the lower gridline is obtained by querying the gridline at $g-1$ with location p_{xy}. The neighbor in the above(below) plane is obtained as follows. We query for the gridline on $p+1$ ($p-1$) with a yx value of p_{xy}. We then query that gridline for the interval at p_{yx}. Thus neighbor queries require O(log N + log n). Note that it is possible that there is no gridline on $p+1(p-1)$ with a yx value of p_{xy}.

Let us now compare the memory consumption of the proposed routing database with that of a uniform grid. Assume that we reserve one word for database annotation. Consider a row of n grid points. This requires n words. A skip-list requires three words on an average per object for its manipulation (akin to each node in a binary tree having a pointer to its parent and the two children). Each interval also has a pointer to the previous interval, the xy value of the high end point of the interval and a pointer to the gridline that contains it. Hence the total memory required per interval is seven words: for k intervals the memory used will be $7k$. However we know that the sum of the lengths of the intervals in grid units must be n (as the intervals are contiguous). Hence as long as the average length of the interval (n/k) exceeds 7 grid units, the memory in the skip-list will be less than the uniform grid. This requirement is easily met in the higher layers. On the lower layers, the metal pins and obstructions in the standard cells cause a fair amount of fragmentation of the gridline and this requirement may not be met.

2.2 Manipulating the database

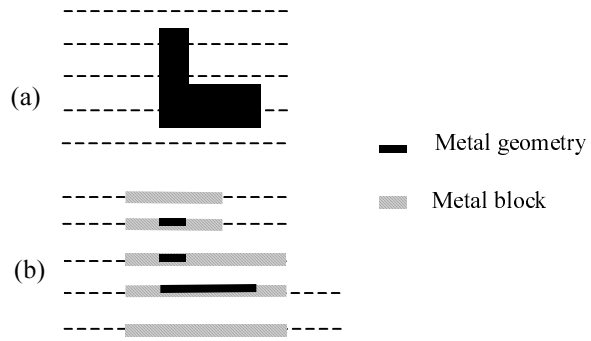
In our implementation, we reserve two words on each interval for annotating the database. Each object that is to be inserted in the database has a descriptor that describes it. One word on the interval is reserved to store a pointer to this descriptor, called its "owner". There are two notions of owning an interval. The first one is the due to actual metal belonging to an object intersecting the interval. The second notion is a weaker one, which expresses a design rule to the interval in question without actual metal overlaying it. The second word is used to encode the nature of "ownership". Thus the effect of each object in the routing database is mapped onto the set of gridlines. We term the intent of an object to express interest on an interval as the "color" of an object. We encode the following "ownership" rules using different bits:

1. Interval owner is permanent metal (cannot be ripped up).
2. Interval owner is temporary metal (can be ripped up).
3. Interval owner permanently blocks interval for other route metal.
4. Interval owner temporarily blocks interval for other route metal.

These consume four bits on the second word. The differentiation between temporary and permanent geometry is required to facilitate search during the rip-up phase. The metal block shadow will include the original geometry or polygon expanded by the spacing rule (sp_k for layer k) plus half of the default metal width ($mw_k/2$). This assumes a default metal routing width per layer and a default via for each pair of routing layers. This scheme of marking the shadow of the blockage, will allow us to mark the impact of an object in the database once. During the expansion step of a search, the region query is replaced by a point query.

We now describe the mechanism to annotate the database for the three kinds of objects that populate the routing database. Pin geometry marking marks all intervals in the region that it occupies. In Figure 1, we show an L shaped pin over the gridlines in the database (a). In (b), the impact of the pin is marked in the database. Each interval that overlaps the rectangle is marked with the pin as the owner. In addition there is a metal block due to the pin in the same region. The metal block region extends outside the pin. Often via geometry is larger than metal half-width and hence via block regions can extend outside the metal block region.

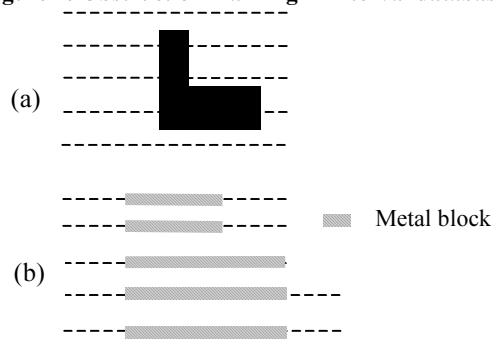
Figure 1 : Pin marking in interval database



Obstruction geometry marking is the same as pin geometry except for two points of note. We set the ownership pointer to a value, which will not be a legal pointer and use it for all obstruction geometry. Secondly there is no need to mark the permanent metal color, instead we simply mark the metal block space. This reduces fragmentation without affecting the information required for routing. Figure 2 shows the marking of an L shaped polygonal obstruction. So far we have presented a very simplistic view of pin and obstruction marking. Extensions to pin marking that enable centerline marking of routes and to block marking for Euclidean spacing rules and to permit obstructions to lie between gridlines have been omitted due to space constraints.

Routes are maintained as centerline objects, where each route segment is an interval on the gridline. These intervals have the ownership pointer set to the descriptor of the route. The temporary

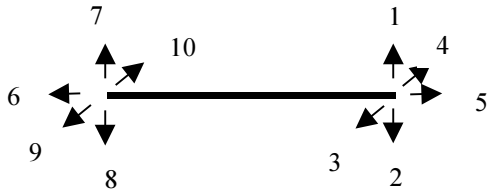
Figure 2: Obstruction marking in interval database



metal bit is set. Appropriate encoding on the second word of the interval represents vias and jogs. To better explain this, let us examine the possible neighbors of an interval. A route segment can have up to ten neighbors as shown in Figure 3. We enumerate the neighbors at the positive end as (index refers to neighbor in Figure 3):

1. Via up at positive end of interval (pos-via-pos)
2. Via down at positive end of interval (pos-via-neg)
3. Jog (wrong way segment) in the negative direction at positive end of interval (pos-jog-neg)
4. Jog in positive direction at positive end of interval (pos-jog-pos)
5. Next interval at positive end of interval

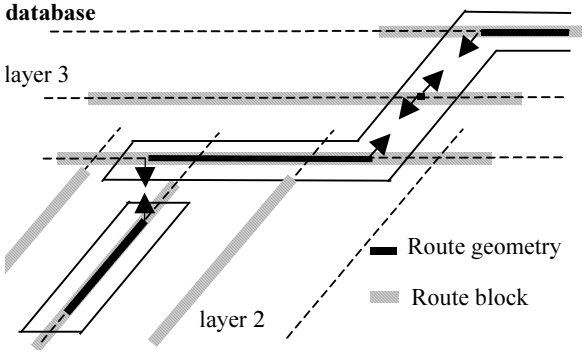
Figure 3: Interval and its neighbors



Indices 6-10 are similarly neighbors at the negative end of the segment. These consume ten bits on the second word. This allows us to do two things efficiently. Given an interval we can enumerate all neighbors to the interval on the same net or (route) efficiently. In addition it is possible to generate a routing path given a point on it. The marking of route geometry differs from pins and obstructions. The centerline of the path is annotated on the intervals. To obtain the metal geometry for an interval segment in plane k, we simply bloat the segment by $mw_k/2$. To compute the geometry due to a via, we look at the neighbor bits to decide the direction of a via. Since we only support a default via per adjacent layer pair, the via geometry can be easily recovered. To mark the metal blockage due route objects, we follow the procedure of bloating the geometry (of a route segment or via) by the spacing rule (sp_k for layer k) plus half of the default metal width ($mw_k/2$).

We present a simple example in Figure 4. We have a vertical route on layer two with a via up at the high end. The route on layer three is a horizontal segment with a jog in the positive direction at the high end. The centerline markings and neighbor markings for these connections are shown in the Figure. Note that the jog on the center gridline on layer three causes a singleton interval. Also observe that the jog on layer three can appear at any X co-ordinate and is not constrained to happen at a grid-point. The hatched segments indicate the metal blockage (some blockage intervals and gridlines have been omitted for clarity).

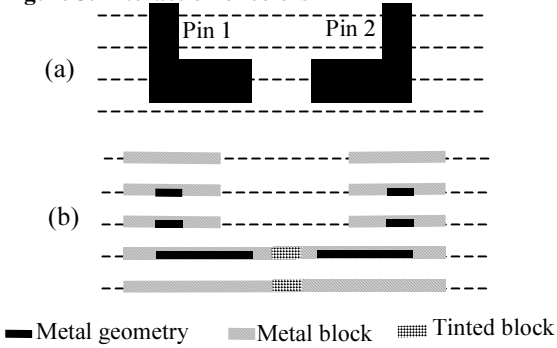
Figure 4: Route geometry marking in the interval database



Often multiple objects in a neighborhood will express interest on a common interval. The intervals in the common neighborhood will inherit the "tinted" color of these pins. Tinting is the process of resolving the resultant color when multiple objects express interest in an interval with their own colors. When multiple objects do interact, the owner word of the interval is set to a value that is not a legal pointer; unless the multiple objects belong to the same net. In this case any one of the owners is maintained. In the first case the interval is orphaned. The nature of ownership word on the interval is set to be the union of the bits. Once an interval is orphaned, it will continue to remain so for all future tints. Consider the example in Figure 5. We have two pins whose blockage markings interact in the region between them. Clearly no net (neither the net that owns pin 1 nor the net that owns pin 2, nor any other net) can occupy the common interval (marked as tinted block) without causing a design rule violation. Should we seek to preserve the original interactions, then instead of maintaining the intervals as non-overlapping (or contiguous) objects, we use a segment tree.

An interesting problem arises when route geometry interacts with other geometry (pin, obstruction or route geometry) leading to orphaned intervals. During the rip-up phase, if such route geometry is deleted, then we must clear the effect of blockage markings due to the deleted object. This is easily accomplished in our system by a simple two step approach. First, we erase all blockage markings in a region where the deleted objected could have had an impact. Next, we rebuild all blockages in this region. In order to do this we need the geometries of pins, obstructions and routes that impact this region. The route geometries are calculated by region queries in the interval database. The pin and obstruction geometries are obtained by querying an on-disk version of kd-trees.

Figure 5: Interaction of colors



We are now in a position to discuss some of the natural

advantages of the interval database over the traditional grid-based schemes. The interval database provides the search engine with the flexibility of making jogs that need not be on a grid point. This is extremely useful in finding connections to pins that are hard to reach on congested layers such as layer zero and layer one. Since we only need to determine the "yx" co-ordinates for the grid-lines on a layer, we can support routing of default widths and spacing that vary across layers easily. Traditional grid-based schemes require the grid-points on adjacent layers to align up. This means that if the grid separations are relatively prime numbers, only a small subset of feasible via locations is captured.

3 EXPERIMENTAL RESULTS

We now present experimental results that justify the efficiency of the interval database. In Table 1 we describe characteristics of some industrial problems for routing. These problems use a fixed area standard cell methodology. All designs include macros along with standard cells. Power routing is also done, so these appear as constraints to the routing problem. Columns 1-3 give the design name, number of nets to be routed (K being used to represent 1000) and number of layers. Columns 4 and 5 describe layers with the same metal width and spacing as the layers within braces. For example, in the case of D5, the metal widths are 190 for layer zero, 220 for layer one and 240 for layers two to five. Similarly the spacing rules are 200 for layer one, 220 for layer two, and 240 for the rest. Observe that design rules vary across layers for all but one design. The technology libraries for all designs are different and are at least 0.25m or below. The technologies describe vias, which have metal overhangs and are not inline with the routes.

Table 1: Experimental designs

Design	# nets	# layers	Metal width rules	Metal spacing rules
D1	107K	5	{0}{1-3}{4}	{0}{1-3}{4}
D2	115K	4	{0-3}	{0-3}
D3	148K	6	{0-3}{4-5}	{0-3}{4-5}
D4	150K	6	{0}{1-4}{5}	{0}{1-4}{5}
D5	319K	6	{0}{1}{2-5}	{0}{1}{2-5}
D6	373K	5	{0}{1-3}{4}	{0}{1}{2-5}

In Table 2 we consider the memory impact for three different database schemes.

1. Grid-point database: We calculate the number of grid-points implied by the intersection of the user-defined tracks on all the layers. We assume that 1 word (4 bytes) per grid point is a reasonable memory cost.
2. Interval database: We take the fully routed designs (almost – see Table 4) and calculate the total number of intervals in the database. Note that this includes fragmentation due to via blockages as well. We calculate the memory as 7 words (28 bytes) per interval.
3. Tree database: We count the total number of rectangles due to pins, pre-routes and metal routes. We use 10 words (40 bytes) per rectangle as the cost of an object. We ignore the impact of complex via-geometry to be conservative.

Table 2 : Comparison of memory consumption

Design	Grid-point database		Interval database		Tree database	
	Count	MB	Count	MB	Count	MB
D1	94.7	379	10.2	286	4.9	200
D2	110.8	443	6.7	189	4.2	172
D3	111.4	446	11.3	319	5.0	200
D4	56.0	224	14.2	399	4.7	190
D5	340.2	1300	39.2	1000	17.6	705
D6	323.6	1200	34.5	967	15.2	608

In Table2 the columns labeled "Count" (columns 2, 4 and 6) count the number of objects (grid-points in column 2, intervals in column 4 and rectangles in column 6). The object counts are in millions. The memory required to manage these is provided in the columns denoted by "MB" (in megabytes). We observe that the grid-point system is most expensive in memory, the tree-based system consumes least memory and the interval database is in the middle. A point to note, the search algorithm can be directly implemented on the grid-point database or on the interval database. However any search on a tree-based system typically requires the construction of a search graph on the fly, or repeated queries into the tree database. The latter is computationally expensive (as shown in the next paragraph). So the comparison with a tree based system must be done, keeping in mind that an adjoining structure for the search needs to be built. Also note that the interval database is sensitive to the actual geometries and the spacing rules. So for a testcase like D4 although the grid-point database is smaller than that for D1, the interval database is larger.

We now study the CPU time for point queries, as these are indicative of the speed of a search. Once again we compare three different database schemes populated with the objects after full detailed routing.

1. Grid-point database: We implement this as a simple two-dimensional array per layer. Given an (x, y), we need to find the indices which define that grid point. The query simply returns the appropriate entry in the array.
2. Interval database: The implementation is as described in Section 2. It includes all the extensions to handle vias correctly. A query to at a point (x, y), returns the interval in the database that contains the point.
3. Tree database: We build a kd-tree on a per layer basis. All pin geometry, obstruction geometry and route geometry arising from segments are added to the appropriate trees. We balance the kd-trees before any query. Note that the geometry for vias is not added to the kd-trees. We believe these should be handled separately for the following reasons. First a design such as D5 has 3.6 million vias at the end. This corresponds to roughly a fifth of the rest of the geometry. Second, via geometry can be inferred from a point query in a two-dimensional kd-tree (where points are inserted instead of rectangles). This will consume less memory.

We query each grid-point (defined by the intersection of horizontal and vertical tracks on a plane) on all the layers. We report the total CPU time required to query all the grid-points on a SUN 400MZ, 4G machine.

Table 3: Comparison of query times

Design	Grid-point database (seconds)	Interval database (seconds)	Tree database (seconds)
D1	119	287	460
D2	134	354	689
D3	132	420	654
D4	74	194	278
D5	443	1866	3600
D6	435	1249	2517

We observe that the grid-point database is the most efficient. The tree database is the least efficient, while the interval database is in between. Quite clearly it would be impractical to have a router that operates on the trees alone and capable of handling large search regions. Also recall that via geometry will need to be queried in addition. In addition, the trees have been balanced statically with all the route geometries added. In reality the route geometries will be added as the routing proceeds and the trees will not be as well balanced. So the times in column 4 are more generous than reality permits.

Finally we present preliminary results for a routing system based on the interval database. This demonstrates the viability of the interval database for routing. All routes have been verified to be design rule clean for spacing and width rules using an independent design rule checker. We have a few opens as we are still refining our rip-up strategy.

Table 4: Routing system with intervals

Design	Open nets/Total nets	Time (seconds)
D1	1/107K	2458
D2	0/115K	3009
D3	4/148K	4769
D4	1/150K	3137
D5	1/319K	11686
D6	18/373K	12071

The run time is for the total run and includes time to build the database, do global routing and detail routing. We use a single CPU for all examples. Bulk of the time is spent in detail routing and rip-up routing.

4 CONCLUSIONS

We have proposed a routing database using intervals as the basic structure instead of grid-points or rectangles. We have shown that it is flexible to support recent trends in design rules for spacing and widths. Our approach is more efficient in memory management than the grid-point based approach. Its performance is superior to a kd-tree based approach. We are quite confident that the interval-based scheme will outperform region queries on any tree-based approach. Our database has handled designs as large as 370K nets in a flat manner.

5 REFERENCES

[1] T. Cormen, C. E. Leiserson and R. L. Rivest, Introduction to Algorithms, MIT Press and McGraw Hill, New York, 1990.

[2] C. Y. Lee, "An algorithm for path connection and its applications", IRE Transactions on Electronic Computers, EC-10:346-365, 1961.

[3] F. Rubin, "The Lee path connection algorithm", Transactions on Computers, C-23:907-914, 1974.

[4] R. A. Finkel and J. L. Bentley, "Quad trees: a data structure for retrieval on composite keys", Acta Inform. 4:1-9, 1974.

[5] J. L. Bentley, "Multidimensional binary search trees used for associative searching", Commun. ACM, 18:509-517, 1975

[6] G. Kedem, "The Quad-CIF tree: a data structure for hierarchical on-line algorithms", Proceedings of the 19th Design Automation Conference, pages 352-357, 1982.

[7] J. K. Ousterhout, "Corner Stitching: a data structuring technique for VLSI layout tools", IEEE Trans. On Computer-Aided Design, Vol 3, No 1, January 1984.

[8] J. B. Rosenberg, "Geographical data structures compared: a study of data structures supporting region queries", IEEE Trans. On Computer-Aided Design, Vol. 4, No. 1, pages 53-67, January 1985.

[9] R. L. Brown, "Multiple storage quad trees: a simpler faster alternative to bisector list quad trees", IEEE Transactions on Computer-Aided Design, Vol 5, No. 3, July 1986.

[10] Y. D. Fontayne and R. J. Bowman, "The Multiple Storage Radix Hash Tree: An Improved Region Query Data Structure", International Conference on Computer-Aided Design, pages 302-305, 1987.

[11] D. Marple, M. Smulders and H. Hegen, "Tailor: A layout system based on trapezoidal corner stitching", IEEE Trans. On Computer-Aided Design, Vol. 9, No. 1, pages 66-90, January 1990.

[12] G. G. Lai, D. Fussel, D. F. Wong, "HV/VH Trees: A New Spatial Data Structure for Fast Region Queries", Proceedings of the 30th ACM/IEEE Design Automation Conference, pages 43-47, June 1993.

[13] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, "Computational Geometry Algorithms and Applications", Springer Verlag 1997.

[14] A. Hetzel, "A Sequential Detailed Router for Huge Grid Graphs", European Design Automation Conference, pages 332-338, 1998.

[15] D. W. Hightower, "A solution to line-routing problem on the continuous plane", Proceedings of the 6th Design Automation Workshop, pages 1-24, 1969.

[16] W. Heyns, W. Sansen, and H. Beke, "A line expansion algorithm for the general routing problem with a guaranteed solution", Proceedings of the 17th Design Automation Conference, pages 243-249, 1980.

[17] J. Cong, J. Fang, and K.-Y. Khoo, "An Implicit Connection Graph Maze Routing Algorithm for ECO Routing", Proceedings of the International Conference on Computer-Aided Design, pages 163-167, 1999.

[18] K. L. Clarkson, S. Kapoor, and P. M. Vaidya, "Rectilinear shortest paths through polygonal obstacles in $O(n (\log n)^2)$ time", Proceedings of the 3rd Annual Symposium on Computational Geometry, pages 251-257, 1987.

[19] S. Q. Zheng, J. S. Lim and S. S. Iyengar, "Finding Obstacle-Avoiding Shortest Paths Using Implicit Connection Graphs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 1, Jan. 1996.

[20] W. Pugh, "Skip Lists: A Probablistic Alternative to Balanced Trees", Communications of the ACM, pages 668-676, June 1990, Volume 33, Number 6.

[21] S. Hojat and P. Villarubia, "An Integrated Placement Synthesis Approach for Timing Closure of PowerPC Microprocessor", Proceedings of the International Conference on Computer Design, pages 206-210, 1997.

[22] N. Shenoy et. al, "A Robust Solution to the Timing Convergence Problem in High-Performance Design", Proceedings of the International Conference on Computer Design, pages 250-257, 1999.