

Analog Topological Placement with Symmetry Constraints Using a $O(n \log \log n)$ Evaluation Algorithm*

Karthik Krishnamoorthy Sarat C. Maruvada Florin Balasa

Dept. of Computer Science, University of Illinois at Chicago, 851 S. Morgan St., Chicago, IL 60607, U.S.A.

Abstract – This paper presents a novel algorithm for device-level analog placement with symmetry constraints. Based on the exploration of symmetric-feasible binary tree representations [3] of the layout, the novel approach employs an efficient model of priority queue introduced by Johnson [8], and brought to the attention of the CAD community by Tang and Wong [15]. The use of this data structure entails a worst-case complexity of $O(n \log \log n)$ for each code evaluation, which is better than of any other existent topological algorithm supporting symmetry [2, 14, 11, 3]. The computation times exhibited by this novel approach are significantly better than those of the algorithms using the traditional exploration strategy based on the absolute representation (like, for instance, [5, 10, 12]).

1 Introduction

In high-performance analog circuits it is often required that groups of devices are placed symmetrically with respect to one or several axes. The main reason of symmetric placement and routing is to match the layout-induced parasitics in the two halves of a group of devices. Failure to match these parasitics in, for instance, differential analog circuits can lead to higher offset voltages and degraded power-supply rejection ratio [5]. Placement symmetry can also be used to reduce the circuit sensitivity to thermal gradients. Failure to adequately balance thermal couplings in a differential circuit can even introduce unwanted oscillations. In order to combat potentially-induced mismatches, the thermally-sensitive device couples should be placed symmetrically relative to thermally-radiating devices.

The problems related to analog placement are numerous: for instance, eliminating systematically-induced mismatches due to dissimilar geometrical choices for matching devices identically specified; arranging devices such that critical structures are shared in common in order to reduce both layout density and induced parasitics; improving the thermal distribution profile, or dealing with noise coupling. Some of the analog-specific constraints can be handled during a combinatorial optimization solving the placement problem by embedding accurate models in the CAD tool and updating the cost function with appropriate penalty terms. Different from these, symmetry constraints can be *directly* taken into account during the exploration of the solution space, diminishing significantly its size when placement

configurations are represented topologically.

The traditional way of approaching device-level placement problems for analog layout is to explore a huge search space, where the cells are represented by means of absolute coordinates, using typically the simulated annealing combinatorial optimization algorithm (e.g., [5]). This exploration, which allows cell overlaps as the cells move (by translations and changes of orientation) in the chip plane, is used by placement tools in several software systems for analog layout [5, 10]. Since this strategy often exhibits a slow convergence, the alternative approach is to use nonslicing topological representations – like, for instance, sequence-pairs [13], ordered tree (O-tree) [7] and binary tree (B*-tree) representations [4], or the transitive closure graphs (TCG) [11] – in order to explore only feasible placement solutions. While placement techniques based on the absolute representation trade-off a larger number of moves in a combinatorial optimization framework for easier- and quicker-to-build layout configurations not always physically realizable, the techniques adopting topological representations trade-off a smaller number of moves for more complex-to-build, feasible layouts.

Selecting an appropriate encoding is not straightforward, though. Many analog designs contain – along with an asymmetric component – an arbitrary number of symmetry groups of devices (that is, groups having distinct symmetry axes), each group containing an arbitrary number of pairs of symmetric devices with the same geometry, as well as self-symmetric devices – presenting a geometrical symmetry and sharing the same axis with its group. All the symmetric devices in a group can have mirror-symmetric (*mirror* symmetry) or identical (*perfect* symmetry) orientations [5]. The symmetry constraints for a pair of devices (B_i, B_j) in the k -th symmetry group have the form: $(x_i + w_i) + x_j = 2 \cdot x_{symAxis_k}$ and $y_i = y_j$, where (x_i, y_i) are the left-bottom coordinates of device B_i , w_i denotes its width, and $x_{symAxis_k}$ is the abscissa of the symmetry axis of the k -th group (assuming the axes are vertical since this is the typical way most layouts are designed). Similarly, a self-symmetric device B_i must satisfy the constraint: $x_i + w_i/2 = x_{symAxis_k}$. Due to this characteristic, only a tiny fraction of the solution space of a topological representation contains symmetric-feasible codes.¹

This paper presents a novel algorithm for solving analog placement problems exploring a subset of binary tree topological representations (B*-trees) [4] – called *symmetric-feasible* [3], the typical presence of an arbitrary number of symmetry groups of

*This research was partly sponsored by the Design Automation Conference Graduate Scholarship Program.

¹An extensive discussion as well as numerical examples can be found in [3].

devices being *directly* taken into account. The novel approach employs an efficient model of priority queue introduced by Johnson [8], achieving a worst-case complexity of $O(n \log \log n)$ per code evaluation, where n is the number of devices. Note that this data structure was brought to the attention of the CAD community by Tang and Wong who used it for the computation of the longest common subsequence in a pair of weighted sequences in the evaluation of sequence-pairs [15]. Our algorithm is essentially different since the topological representation is different, and also an arbitrary number of symmetry constraints are present in the design. Actually, the present algorithm exhibits a better complexity than of any other existent topological algorithm supporting symmetry [2, 14, 11, 3]. Note that the paper focuses on the efficiency of the exploration of the solution space; other important aspects specific to analog placement like, for instance, eliminating systematically-induced mismatches, arranging devices such that critical structures are shared in common, handling thermal constraints, dealing with noise coupling, are beyond the scope of the paper, although the implementation takes into account some of these aspects.

This paper is organized as follows. Section 2 presents the efficient priority queue model due to Johnson [8]. For the sake of consistency, Section 3 will briefly review the concept of *symmetric-feasible* B^* -tree and some essential prior results. The core of the paper is Section 4 which describes the novel evaluation algorithm for analog placement. A simple example will illustrate its flow (Section 4.1). The algorithm complexity (Section 4.2) and different techniques enhancing the efficiency (Section 4.3) will be thoroughly analyzed. Finally, Section 5 will present an overview of the experimental results and Section 6 will summarize the conclusions of this research.

2 Johnson’s priority queue

The keys, integers in the set $\{1, \dots, N\}$, in Johnson’s priority queue model [8] are kept in n *buckets*. The non-empty buckets, together with bucket 0 (always present and used as a header) are kept in the *bucket list*, a doubly-linked list sorted on the key values. Buckets in the list comprise the leaves of a binary tree T . The nodes of T are indexed by defining a complete [6] *host* binary tree $H = \{1, \dots, 2^h + N\}$, where $h = \lceil \log(N + 1) \rceil$. Each node q in H , different from the root, is a child of node $\lfloor q/2 \rfloor$.

Example Figure 1 shows a priority queue for keys in the set $\{1, \dots, 6\}$, containing, in addition to bucket 0, only the keys 2 and 4. The list of buckets is shown below the leaves of the tree T , drawn with bold solid lines. The host tree H is represented with dashed lines. Because of reason of space, the figures in the rest of the paper will show only the bucket lists of the priority queue.

The driving idea is to conceive a complete binary tree with $N + 1$ leaves on the lowest level, but dynamically to leave as much of the tree as possible unconstructed. Whenever a new bucket is inserted, its path would be constructed upward, until the new path intersected the path of some non-empty bucket. Then

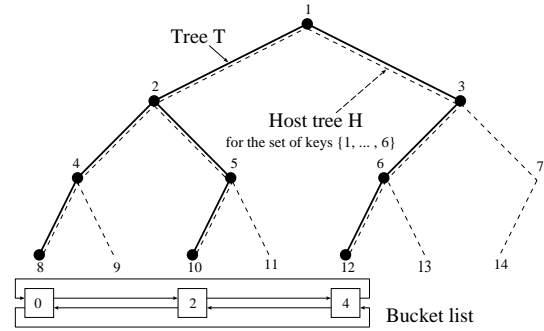


Figure 1: Johnson’s priority queue [8]

the path would be followed downward towards leaves to find the non-empty bucket adjacent to which the new bucket must be inserted into the list. Deletion would be a reversal of this process.

Since the height h of the host binary tree is $O(\log N)$, and a path in the tree T of length k is traversed using a mechanism somewhat similar to the binary search, visiting at most $2\lceil \log k \rceil$ nodes [8], it follows that key insertions and deletions take $O(\log \log N)$ time.

3 Symmetric-feasible B^* -trees

For the sake of consistency, this section will briefly review some background results essential to understand the present research.

A binary tree layout representation (B^* -tree) [4], which nodes represent rectangular cells, induces the following vertical (y -) and horizontal (x -) positioning constraints:

- (a) each cell whose node is in the left subtree is above the cell whose node is the parent;
- (b) if the y - projections of two cells are overlapping, the cell whose node is visited first in a *preorder* traversal of the tree (i.e., visit recursively any node before its left and right subtrees) is to the left of the cell whose node is visited the second.

It is well-known that the pair of *preorder* and *inorder* (i.e., visit recursively any node in between visiting its left and right subtrees) traversals uniquely determine the binary tree. Several algorithms building the tree from its traversals in optimal time and space are known (e.g., [1]).

Definition A binary tree representation is called *symmetric-feasible* if its pair of (*inorder*, *preorder*) traversal has the following property: for any distinct cells A, B in a symmetry group,

$$(S) \quad A \overset{inorder}{\prec} B \iff sym(B) \overset{preorder}{\prec} sym(A)$$

i.e., node A *precedes* B in the *inorder* traversal of the binary tree iff node $sym(A)$ – corresponding to cell A ’s symmetric pair – *succeeds* node $sym(B)$ in the *preorder* traversal. In addition, any two cells A, B belonging to different symmetry groups cannot satisfy the inequalities

$$A \overset{inorder}{\prec} B \overset{inorder}{\prec} sym(B) \overset{inorder}{\prec} sym(A) \quad \text{and}$$

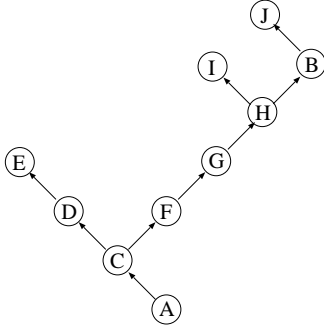


Figure 2: Symmetric-feasible B^* -tree representation of a layout with a group of 2 pairs of symmetric devices (F, G) and (C, J).

$$B \stackrel{preorder}{\prec} A \stackrel{preorder}{\prec} sym(A) \stackrel{preorder}{\prec} sym(B)$$

This second condition eliminates those trees where cell A results above cell B , whereas cell $sym(B)$ results above $sym(A)$, the two pairs preventing each other to be aligned horizontally within the groups.

Example The binary tree representation in Fig. 2 is symmetric-feasible relative to a group of two symmetric pairs of cells (F, G) and (C, J). Indeed, the (*inorder, preorder*) traversals are: ($EDCFGHJBA, ACDEFGHIBJ$), and condition (S) is satisfied: $C \stackrel{inorder}{\prec} G$ and their symmetric pairs $F \stackrel{preorder}{\prec} J$, and so on.

Remark Since traversing a forest of rooted trees in postorder is exactly the same as walking the corresponding binary tree in inorder [9], the definition above can be easily adapted to O-trees replacing *inorder* with *postorder*.

Previous research has shown that the exploration of the solution space of placement problems with symmetry constraints can be reduced to the exploration of those binary tree representations which are symmetric-feasible relative to every symmetry group of cells [3], the benefit being a significant reduction in size of the search space.

Next, Section 4 will present an evaluation algorithm of complexity $O(n \log \log n)$ building the placement from a given symmetric-feasible binary tree, whose nodes correspond to the devices. This algorithm is executed in each inner-loop iteration of the simulated annealing, evaluating the layout cost after each move. The move set of the annealer is designed to preserve the feasibility of B^* -trees relative to symmetry [3], comprising also device rotations, mirroring, moves of entire symmetry groups. The analog devices to be placed on the chip area are represented by rectangular blocks B_1, \dots, B_n , each block B_i having the width w_i and the height h_i , and having (x_i, y_i) as coordinates of its left-bottom corner. The nodes in the binary tree (B^* -tree) representation of the layout are labeled B_1, \dots, B_n as well, each node B_i corresponding to the block with the same name.

4 Fast evaluation of symmetric-feasible B^* -trees

The algorithm computing the coordinates y_i operates basically by walking in preorder the binary tree representation. In the preorder traversal the nodes corresponding to the devices *below* are visited before the nodes of the devices *above* due to the vertical positioning constraints (a) induced by the B^* -tree representation (see Section 3). In the absence of symmetry constraints one traversal of the tree would suffice; however, a second traversal may be necessary in order to satisfy the vertical (y -) symmetry constraints of the type: $y_i = y_j$ for two symmetric devices (B_i, B_j). It can be proven [3] that if the binary tree encoding possessing the symmetric-feasibility property (S), then the y -coordinates of the devices can be computed after at most two preorder traversals of the tree, such that the vertical positioning and symmetry constraints be satisfied. As the traversal of the binary tree is performed in linear time, the computation of y_i 's is done in $O(n)$ time. An illustrative example is given in Section 4.1.

After the computation of the y -coordinates, the device abscissae are determined inserting the devices in the layout in the sequence given by the *preorder* and, subsequently, *inverse preorder* visit of the nodes of the binary tree, and updating the contour of the right and left border of the partial placement configuration while dealing with the symmetry constraints. This computation is efficiently performed using a novel algorithm employing Johnson's priority queue (JPQ) [8] in order to achieve a $O(n \log \log n)$ worst-case time complexity (see Sections 4.2 and 4.3) for each evaluation of a symmetric-feasible B^* -tree.

A priority queue as proposed by Johnson [8] is used in the algorithm to register the fragmentation of the right or left contour of the (partial) placement configuration. The keys of the JPQ are the y -coordinates where the contour of the right/left border of the analog block is broken. Therefore, the maximum key is H – the height of the analog block (which is already known since the y -coordinates have been already computed) and the minimum key is zero. Each bucket n of the JPQ – called $JpQBucket$ – has attached the key $n.key$ and a value $n.x$, the abscissa of the border segment starting from the ordinate $n.key$ upwards.

Firstly, the binary tree topological representation is walked in *preorder*, such that the blocks to the *left* are visited before the ones to the *right* (see in Section 3 the horizontal positioning constraints (b) induced by the B^* -tree representation). When the node B_i is visited, the JPQ is updated storing the contour of the *right* border as a result of the insertion of the new y -spanning interval $[y_i, y_i + h_i]$ of the device corresponding to the node B_i . In the absence of symmetry constraints, one preorder traversal would suffice. In most of the cases, one single traversal cannot yield a feasible placement in symmetry point of view (see the illustrative example in Section 4.1). The fact that the binary tree code possesses the property of “symmetric-feasibility” [3] is crucial: as proven in [3], it ensures that one additional traversal of the tree encoding in *inverse preorder* is sufficient to fix all the x -symmetry constraints of the form $(x_i + w_i) + x_j = 2 \cdot x_{symAxis}$,

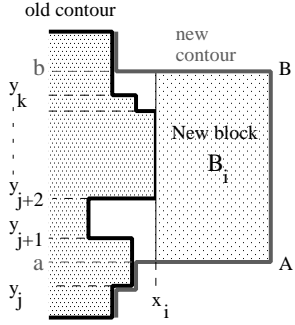


Figure 3: Modification of the border contour in the procedure `insertInterval([y_i, y_i + h_i])`.

where (B_i, B_j) is a pair of symmetric devices. In this second (*inverse preorder*) traversal, the JPQ registers the contour of the left border of the analog block.

The general scheme of the algorithm computing the device abscissae is given below. For the sake of readability, the algorithm was simplified assuming that all the devices subject to symmetry constraints belong to a single symmetry group. The implementation is more refined though, able to deal with an arbitrary number of symmetry groups, as our experimental results (Section 5) clearly show.

General scheme of the algorithm computing the device abscissae

```

JpqBucket n=JPQ.insert(0); n.x=0; // the key 0
// is inserted as a sentinel in the JPQ, setting its x-value to 0
initialize x_i=0 and x_symAxis=0;
// initialize the device abscissae, the position of the sym. axis
W=0; // W - crt. width of the analog block
for each node B_i in B*-tree (visited in preorder)
    JPQ.insertInterval([y_i, y_i + h_i]); // modify the JPQ
end_for // to model the right border; x_i, x_symAxis are updated
JPQ.makeEmpty(); // reset JPQ, eliminating all the buckets
// (except the sentinel bucket with the key 0)
n=JPQ.bucket(0); n.x=W=max{x_i+w_i}; // set
// the x-value of this JpqBucket with the crt. width of the block
initialize x_i=W; // re-initialize the device abscissae
for each node B_i in B*-tree (inverse preorder visit)
    JPQ.insertInterval([y_i, y_i + h_i]); // modify the JPQ
end_for // to model the left border; the final x_i's are computed
JPQ.makeEmpty();
W=W-min{x_i}; // W - total width of the analog block
    
```

The procedure `insertInterval` (shown below) modifies the priority queue storing the lateral contour when a new device B_i is added to the partial placement. Assuming the B^* -tree is walked in preorder, to modify the contour of the right border due to the block B_i , the largest key (y -coordinate) $\leq a$ (y_j in Fig. 3) is detected first (`findMaxKeyLE(a)`). All the larger keys inside the interval (a, b) (that is, y_{j+1} to y_k in Fig. 3) will be removed from the JPQ and new keys a and b are inserted. (Special care is taken when a and/or b coincide with y_j , resp. y_k .) x_i – the

abscissa of block B_i , is the maximum of the x -coordinates attached to the JPQ buckets having the keys y_j, y_{j+1}, \dots, y_k . In order to keep minimal the number of buckets in the priority queue, the neighbor buckets of the one having the key a are eliminated if they have attached the same x -coordinates since the segment AB would be collinear with neighbor segments of the contour. (Such a situation will be exemplified in Section 4.1.)

```

procedure insertInterval([a, b]) // [a, b] = [y_i, y_i + h_i]
{ JpqBucket* q=JPQ.findMaxKeyLE(a);
  // find bucket of the JPQ whose key is the maximum one ≤ a
  // (that is, the bucket with key y_j from Fig. 3)
  x_i=q->x;
  while ( (p=q->right)->key < b)
    if (preorder) x_i=max{x_i, p->x}
    else // inverse preorder // x_i=min{x_i, p->x-w_i};
      last_x=p->x; JPQ.remove(p->key); // remove
  end_while // the keys covered by (a,b): y_{j+1} to y_k in Fig. 3
  if (q->key < a) {JPQ.insert(a); q=q->right;}
  if (b < q->right->key) // insert new keys a and b
  { JPQ.insert(b); q->right->x=last_x; }
  // the new bucket with key b gets the same x-coord. as the
  // last bucket removed (i.e., of bucket with key y_k – Fig. 3)
  if device B_i has a symmetric B_j
    if node B_j in B*-tree has been visited
      t=2x_symAxis-(x_j+w_j); // update x_i due to a sym.
      if (preorder) x_i=max{x_i, t} // constraint
      else // inverse preorder // x_i=t;
    elseif (preorder) x_symAxis=max{x_symAxis, x_i+w_i};
    if (preorder) q->x=x_i+w_i else q->x=x_i;
    EliminateAdjacentBucketsWithSameX(q);
  } // to keep the JPQ minimal in size
}
    
```

4.1 Illustrative example

Consider the symmetric-feasible B^* -tree in Fig. 2 representing a layout with ten rectangular blocks having the widths and heights indicated in the caption of Fig. 4. In the first preorder traversal ($ACDEFGHIBJ$), the computation of the y -coordinates yields successively $y_A = 0$, $y_C = y_A + h_A = 3$, $y_D = y_C + h_C = 5$, $y_E = y_D + h_D = 8$, $y_F = y_G = y_H = y_A + h_A = 3$, $y_I = y_H + h_H = 7$, $y_B = y_A + h_A = 3$, $y_J = y_B + h_B = 4$. Note that in the last iteration, after the computation of y_J , the ordinate of its symmetric $-y_C$ – is equally modified from 3 to 4. But this would lead to an overlap between blocks C and D . In the second traversal y_D and y_E are recomputed using the new value y_B and are updated to 6 and, respectively, 9.

The computation of the device abscissae is initiated by inserting the key 0 into the empty priority queue (JPQ). In this way, the procedure `findMaxKeyLE` will always return a pointer different from NULL, since all $y_i \geq 0$. The first visited node in the B^* -tree is A ; since the y -spanning interval of device A is $[y_A, y_A + h_A] = [0, 3]$, the keys 0 and 3 must be inserted in the JPQ provided they do not exist already, while removing the keys

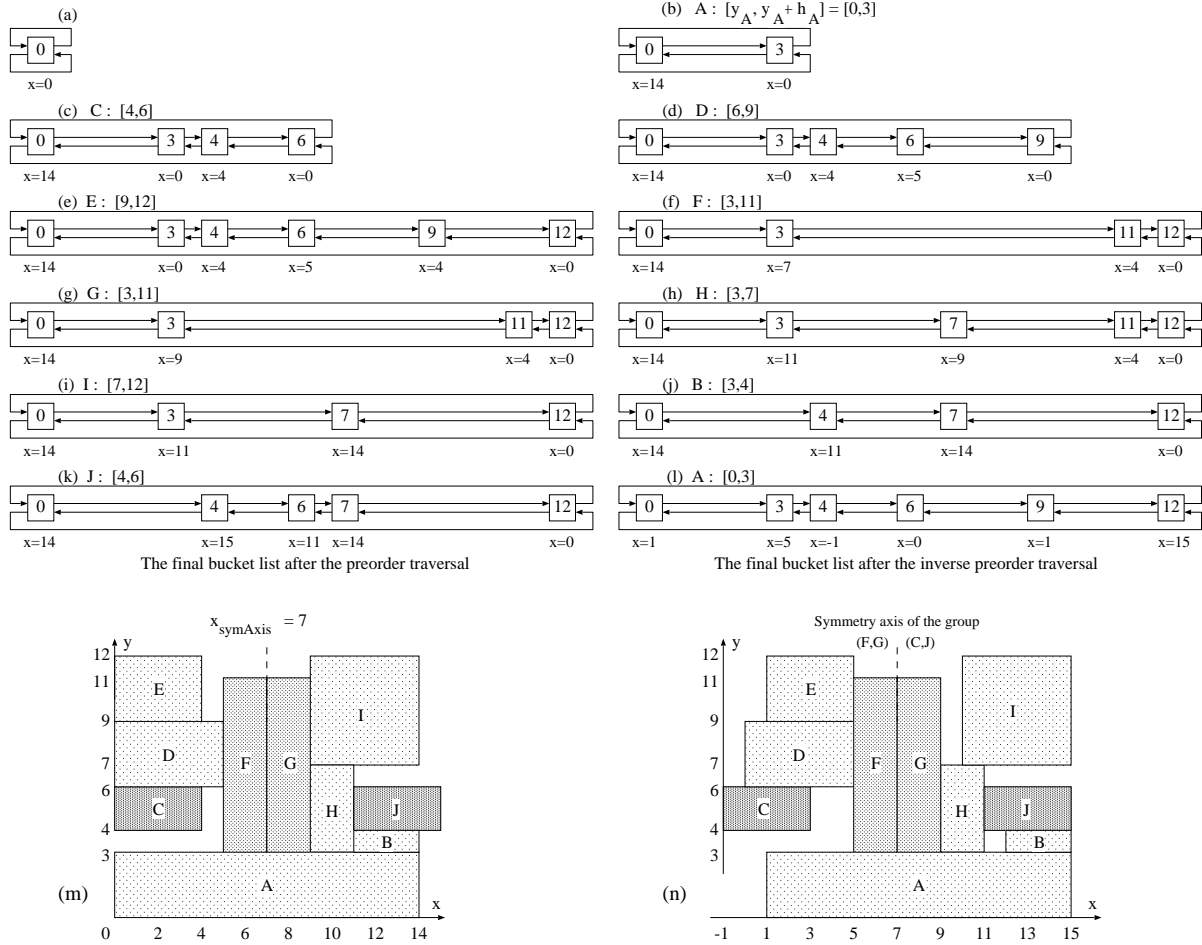


Figure 4: Illustrative example: the evaluation algorithm applied to the symmetric-feasible B*-tree representation in Fig. 2 with a group of two pairs of symmetric devices (F,G) and (C,J). The widths and heights of the devices are: A(14x3), B(3x1), C(4x2), D(5x3), E(4x3), F(2x8), G(2x8), H(2x4), I(5x5), J(4x2). (a-k) The update of the bucket list of Johnson’s priority queue after the (first) visit of each tree node in the preorder traversal (ACDEFGHIBJ). (l) The final bucket list of the priority queue after the (second) visit of each tree node in the inverse preorder (JBIHGFEDCA). (m) Device placement after the preorder traversal, corresponding to the bucket list (k). (n) The final placement, after the inverse preorder traversal, corresponding to the bucket list (l).

covered by the interval $[0, 3]$. Only the key 3 need to be inserted and no other key is deleted. The procedure `insertInterval` with the `preorder` flag set to `true` yields $x_A = 0$ and assigns to the JPQ bucket 0 the x -value $x_A + w_A = 14$ (see Fig. 4(b)). The visit of the subsequent nodes C, D, and E modifies the JPQ in a similar way. Since the y -spanning interval of the device F is $[y_F, y_F + h_F] = [3, 11]$, the keys 4, 6, and 9 – covered by this interval – are removed, and the key 11 is inserted into the JPQ (see Fig. 4(f)). The abscissa x_F is the maximum of the x -values of the removed buckets (4, 6, and 9) and of the bucket 3, that is, $x_F = \max\{0, 4, 5, 4\} = 5$. The x -value of bucket 3 becomes $x_F + w_F = 7$, while the x -value of the bucket 11 inherits the abscissa of 9 (the last bucket removed), that is, 4.

The bucket list of the JPQ in Fig. 4(k) corresponds to the placement in Fig. 4(m). Note that after each iteration, the JPQ models

the contour of the right border of the partial placement, the keys being the y - coordinates where the contour changes direction.

After the first (*preorder*) traversal of the B*-tree representation (Fig. 2), the x - symmetry constraint for the pair of devices (C,J) is still not satisfied: indeed, $((x_C + w_C) + x_J = 15) > (2x_{symAxis} = 14)$. In this traversal, the position of the symmetry axis is determined and the rightmost devices in the symmetry pairs are positioned attempting to meet the x - symmetry constraints $(x_i + w_i) + x_j = 2x_{symAxis}$. However, due to the topologic constraints induced by the B*-tree, some of these devices may be pushed further to the right (like J in this case). A second traversal of the B*-tree in the inverse *preorder* (that is, *JBIHGFEDCA*) will yield the final placement in Fig. 4(n), which correctness is guaranteed by the *symmetric-feasibility* of the binary tree encoding, as proven in [3]. The JPQ is reinitial-

ized, the only difference being that the x -value of the node 0 is set to $W = 15$, the current width of the analog block. Afterwards, the JPQ evolves such that after each iteration it describes the *left* border of the chip, and the last bucket list in Fig. 4(l) corresponds to the final placement.

It can be proven that this evaluation algorithm produces a placement of minimum area relative to the symmetry requirements and to the positioning constraints induced by the binary tree representation.

4.2 Complexity analysis

The priority queue used to compute the device abscissae contains keys in the set $\{0, \dots, H\}$, where H is the height of the analog block. Since the key insertions and deletions take $O(\log \log H)$ each² [8], and *findMaxKeyLE* as well, we may be tempted to consider $O(\log \log H)$ the worst-case time bound per iteration. However, this is not always true: when the bucket list of the priority queue grows, it can gain at most two new keys per iteration (see Fig. 4(c)); but when it decreases in size (see Fig. 4(f)), as many as $O(n)$ keys can be deleted.³ However, using the *aggregate method of amortized analysis* [6], it can be shown that the amortized time bound is $O(\log \log H)$ per iteration. Intuitively, the reason is that each key can be deleted at most once for each time it is inserted. In an amortized analysis, the time required to perform a sequence of operations is averaged over all the operations performed. Since there are n iterations per binary tree traversal, the overall time complexity is $O(n \log \log H)$.

Actually, the worst-case complexity can be reduced to $O(n \log \log n)$. The next section will discuss two techniques for enhancing the efficiency.

4.3 Enhancing the algorithm efficiency

1. The priority queue can have at most $n + 1$ keys as there are at most n segments on the border contours determined by the y -spanning intervals $[y_i, y_i + h_i]$. The y -coordinates of the devices can be “normalized” by replacing each of them by its index in their increasingly-ordered set $\mathcal{S} = \{y_i\} \cup \{y_i + h_i\}$. In this way, the y -coordinates can be considered, without loss of generality, integers in the range $[0, n]$ (n being the number of devices). Another important consequence of the “normalization” is the fact that the size of the priority queue will be kept minimal.

For instance, in the illustrative example of Section 4.1, after the sorting and elimination of duplicates, the set $\mathcal{S} = \{y_i\} \cup \{y_i + h_i\} = \{0, 3, 4, 6, 7, 9, 11, 12\}$ has 8 elements. When visiting, say, node F of the B^* -tree, instead of the y -spanning interval $[y_F, y_F + h_F] = [3, 11]$, the actual parameter of the procedure *insertInterval* will be the normalized interval $[1, 6]$ (since 3 and

²Actually, according to [8], the asymptotic upper bound is even tighter: $O(\log \log(\max_i h_i))$, where h_i are the heights of the devices. But in the worst case, $h_i = \Theta(H)$.

³Such a situation could occur if the JPQ had $O(n)$ keys and in the next iteration the block had the height of the whole chip; the JPQ would be reduced to only two buckets having the keys 0 and H .

11 are the 1st and, respectively, the 6th elements of the set \mathcal{S}). Note also that the height of the host binary tree of the priority queue is 4 in the illustrative example of Section 4.1 (since the keys are in the set $\{0, \dots, 12\}$), whereas when normalized coordinates are used instead, the height of the host binary tree of the priority queue is only 3 (since the keys are in the set $\{0, \dots, 7\}$).

The normalization should not be done with a general purpose sorting algorithm (of worst-case complexity $O(n \log n)$) since the overall complexity would be dominated by the sorting. Instead, the radix sort of complexity $O(n + H)$ [6], or sorting by inserting the keys in Johnson’s priority queue – of complexity $O(n \log \log(H/n))$ [8] – should be used. This sorting algorithms have a linear complexity when $H = O(n)$. When using, for instance, the radix sort for normalization, the overall complexity becomes $O(H + n \log \log n)$; if, in addition, $H = O(n)$, the overall complexity is $O(n \log \log n)$.

2. A more subtle strategy – currently used in our latest implementation – is to use as *keys* the indices of the B^* -tree nodes in the inorder sequence, rather than y -coordinates from the set $\mathcal{S} = \{y_i\} \cup \{y_i + h_i\}$. For instance, the inorder sequence of the tree in Fig. 5(a) is $(BDC A)$, hence the index of A is 4, the index of B is 1, etc. The inorder indices are easily generated during the first preorder traversal of the B^* -tree computing the y -coordinates of the devices. In this way, all the keys are in the set $\{0, \dots, n\}$ (rather than $\{0, \dots, H\}$), the key 0 being used as a sentinel. Each `JpqBucket` n has also attached the bounds $n.a$ and $n.b$ of an interval $[a, b]$, initially equal to the y -spanning interval $[y, y + h]$ of the corresponding device. The priority queue is initialized inserting the key 0, whose interval is $[0, M]$, where M is an arbitrary integer larger than H – the height of the analog block (see Fig. 5(b)). The procedure `insertInterval` is replaced in the evaluation algorithm by `insertInorderIndex`. Only its first part, different from the former procedure, is given below:

```

procedure insertInorderIndex( $K$ ) // index  $K$  of node  $B_i$ 
{
  JpqBucket*  $q = \text{JPQ.insert}(K)$ ;
   $q \rightarrow a = y_i$ ;  $q \rightarrow b = y_i + h_i$ ;
  while ( ( $p = q \rightarrow \text{left}$ )  $\rightarrow b \leq q \rightarrow b$ ) //  $p = \text{pred}(q)$ 
     $x_i = (\text{preorder}) ? \max\{x_i, p \rightarrow x\} : \min\{x_i, p \rightarrow x - w_i\}$ ;
    JPQ.remove( $p \rightarrow \text{key}$ ); // block  $p \rightarrow \text{key}$  all covered
  if ( $p \rightarrow a < q \rightarrow b$ ) // when block of index  $p \rightarrow \text{key}$  only
     $p \rightarrow a = q \rightarrow b$ ; // partially covered, trim its interval  $[a, b]$ 
     $x_i = (\text{preorder}) ? \max\{x_i, p \rightarrow x\} : \min\{x_i, p \rightarrow x - w_i\}$ ;
  if device  $B_i$  has a symmetric  $B_j$  ...
}

```

Figure 5(b) shows the update of the bucket list when the enhanced evaluation algorithm is applied. Notice that all the predecessor keys of the newly inserted one, having smaller interval upper-bounds, are deleted since their corresponding devices are completely covered from the right by the new one. For instance, note the removal of the key 1 in the last bucket list (Fig. 5(b)) meaning that block B (*index* = 1) becomes fully covered when block D (*index* = 2) is inserted.

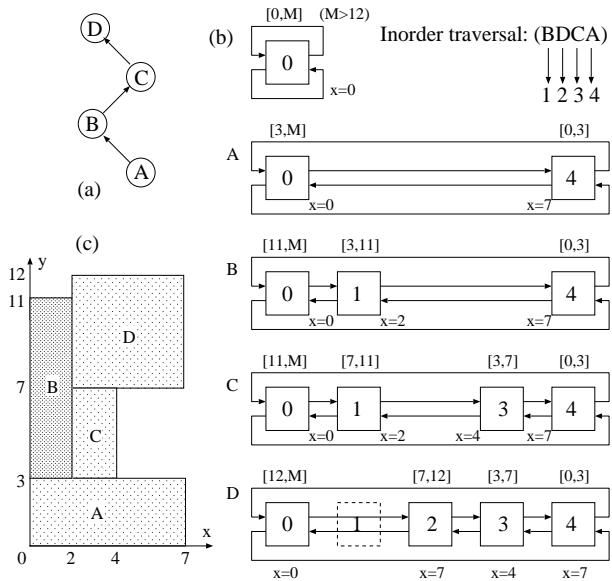


Figure 5: Enhanced evaluation algorithm: (a) B^* -tree. The widths and heights of the devices are: A(7x3), B(2x8), C(2x4), D(5x5). (b) The update of the bucket list of the priority queue after the visit of each tree node in the preorder traversal (ABCD). (c) Device placement corresponding to the last bucket list.

A similar analysis as in Section 4.2 proves that the overall complexity is $O(n \log \log n)$.

5 Experimental results

A placement tool for analog layout using selectable exploration algorithms has been implemented in C++ on a SUN Blade 100 workstation. This prototype tool can operate both with different topological representations (sequence-pairs, O-trees, and B^* -trees) and different code evaluation algorithms. In addition, a complementary placement algorithm based on the traditional absolute representation has been embedded in the tool as well. The tool uses a simulated annealing optimizer with a complex cost function comprising, along with the chip area and estimated wire length, different penalty terms like, e.g., modeling device separation constraints.⁴ Besides symmetry constraints, the tool handles systematically-induced device mismatches, alignment constraints, and also performs shape optimizations.

Table 1 displays the results of our experiments. The performance of the algorithm described in this paper (the last two columns) has been evaluated in comparison with two algorithms able to handle symmetry constraints based on symmetric-feasible (S-F) sequence-pairs [2] and O-trees [14] of quadratic complexity per code evaluation, and an algorithm exploring symmetric-feasible B^* -trees and using segment trees – a data structure often used in computational geometry [3]. Tests with a traditional

⁴Table 1 displays only the area rather than an abstract cost, though.

algorithm based on the absolute representation have been performed as well.

The benchmarks are analog blocks containing symmetry groups of devices, components of a spread spectrum transceiver used in cordless telephone applications and wireless modems. The size of these benchmarks is quite typical: analog circuits seldom contain more than 100 cells per hierarchical level. Figure 6 displays the placement solutions for two of these examples. The first one, for instance, is a frequency divider with selectable (2 or 4) ratio.

The experiments show that using topological representations in analog placement is significantly better both in terms of computational effort and placement quality than using the more traditional absolute representation. The running times obtained when using our novel evaluation approach based on the highly efficient priority queue [8] are regularly better than the computation times obtained when using (S-F) sequence-pairs [2] or ordered trees [14] (although in terms of placement quality all these algorithms proved to be quite effective). In addition, the current technique proved to be faster than a previous approach using symmetric-feasible B^* -trees as well [3]. The increase of efficiency is due to the use of Johnson's priority queue entailing a better complexity of $O(n \log \log n)$ per inner-loop iteration, whereas the previous evaluation approach using a segment tree data structure has a complexity of $O(n \log n)$ [3].⁵

6 Conclusions

This paper has presented a novel analog placement technique operating on a subset of binary tree topological representations of the layout, where symmetry constraints – typical in analog placement – are directly taken into account during the exploration of the solution space. The novel evaluation approach, executed after each modification of the B^* -tree representation, employs an efficient model of priority queue which ensures a worst-case complexity of $O(n \log \log n)$.

References

- [1] A. Andersson, S. Carlsson, "Construction of a tree from its traversals in optimal time and space," *Inform. Processing Letters*, Vol. 34, pp. 21-25, 1990.
- [2] F. Balasa, K. Lampaert, "Symmetry within the sequence-pair representation in the context of placement for analog design," *IEEE Trans. CAD of IC's and Syst.*, vol. 19, no. 7, pp. 721-731, 2000.
- [3] F. Balasa, S.C. Maruvada, K. Krishnamoorthy, "On the exploration of the solution space in analog placement with symmetry constraints," *IEEE Trans. CAD of IC's and Syst.*, vol. 23, no. 2, pp. 177-191, Feb. 2004.

⁵The areas obtained with these two methods were the same since they operate with the same topological representation – (S-F) B^* -trees – and the same move set; in addition, the random number generators started with the same seeds.

