FAST-SP: A Fast Algorithm for Block Placement based on Sequence Pair *

Xiaoping Tang and D.F. Wong
University of Texas at Austin Austin, TX 78712 USA
(tang, wong)@cs.utexas.edu

Abstract

In this paper we present FAST-SP which is a fast block placement algorithm based on the sequence-pair placement representation. FAST-SP has two significant improvements over previous sequence-pair based placement algorithms: 1) FAST-SP translates each sequence pair to its corresponding block placement in $O(n \log \log n)$ time based on a fast longest common subsequence computation. This is much faster than the traditional $O(n^2)$ method by first constructing horizontal and vertical constraint graphs and then performing longest path computations. As a result, FAST-SP can examine more sequence pairs and obtain a better placement solution in less runtime2) FAST-SP can handle placement constraints such as pre-placed constraint, range constraint, and boundary constraint. No previous sequence-pair based algorithms can handle range constraint and boundary constraint. Fast evaluation in $O(n \log \log n)$ time is still valid in the presence of placement constraints and a novel cost function which unifies the evaluation of feasible and infeasible sequence pairs is used. We have implemented FAST-SP and obtained excellent experimental results. For all MCNC benchmark block placement problems, we have obtained the best results ever reported in the literature (including those reported by algorithms based on O-tree and B*-tree) with significantly less runtimeFor example, the best known result for ami49 (36.8 mm^2) was obtained by a B*-tree based algorithm using 4752 seconds, and FAST-SP obtained a better result (36.5 mm^2) in 31 seconds.

1. Introduction

Rapid advances in integrated circuit technology have led to a dramatic increase in the complexity of VLSI circuits. According to the SIA National Technology Roadmap for Semiconductors [1], we will soon have designs in less than 0.1 micron technology with over 100 million transistors. Circuits with such enormous complexity have to be designed hierarchically. Circuit placement within each level of the hierarchy is a complex block placement problem. A good block placement solution not only minimizes chip area, but also minimizes interconnect cost which is crucial in determining circuit performance in deep submicron designs. Although block placement is a classical problem with many previous algorithms [2, 3], it remains to be a hard problem [4]. In [5], Murata et al introduced an elegant representation of block placement called sequence pair. It has been widely recognized in the CAD community with many follow-up works extending sequence pair to handle obstacles, soft module, rectilinear block and analog layout [6, 7, 8, 9, 10, 11]. However, with the advent of new representations such as O-tree [12, 13] and B*-tree [14] producing better results than sequence-pair based algorithms, people are beginning to believe that sequence pair is an inferior representation due to its inherently larger solution space (as compared with O-tree and B*-tree).

In this paper we present FAST-SP which is a fast block placement algorithm based on the sequence-pair placement representation. FAST-SP has two significant improvements over previous sequence-pair based placement algorithms:

1) *Fast evaluation of sequence pair*: All sequence-pair based block placement algorithms use simulated annealing where the generation and evaluation of a large number of sequence pairs is required. Therefore a fast algorithm is needed to evaluateeach generated sequence pair, i.e., to translate the sequence pair to its corresponding block placement. The most commonly used evaluation method is the $O(n^2)$ algorithm based on constructing a pair of horizontal and vertical constraint graphs and computing longest paths in both graphs [5]. [15] attempted to improve the speed of sequence pair evaluation to $O(n \log n)$ time. However,[15] did not show how to compute the positions of the individual blocks. Recently, an $O(n \log n)$ algorithm was proposed in [16] to evaluate a sequence pair based on computing longest common subsequence in a pair of weighted sequences. In this paper, we improve the algorithm in [16] to run in $O(n \log \log n)$ time. As a result, FAST-SP can examine more sequence pairs and obtain a better placement solution in less runtime.

2) *Handle placement constraints* FAST-SP can handle placement constraintssuch as pre-placed constraint, range constraint, and boundary constraint. No previous sequence-pair based algorithm can handle range constraint and boundary constraint. Fast evaluation in$O(n \log \log n)$ time is still valid in the presence of placement constraints.It is clear that some sequence pair is not feasible with respect to satisfying the given placement constraints. It can be shown that any placement that satisfies the given constraints is represented by a sequence pair. Moreover, a necessary and sufficient condition of feasible sequence pair is derived. Finally, we give a novel cost function which unifies the evaluation of feasible and infeasible sequence pairs.

We have implemented FAST-SP and obtained excellent experimental results. For all MCNC benchmark block placement problems, we have obtained the best results ever reported in the literature (including those reported by algorithms based on O-tree and B*-tree) with significantly less runtime. For example, the best known result for ami49 (36.8 mm^2) was obtained by a B*-tree based algorithm using 4752 seconds, and FAST-SP obtained a better result (36.5 mm^2) in 31 seconds.

2. Block Placement by Sequence Pair

A sequence pair is a pair of sequences of n elements representing a list of n blocks. The sequence pair structure is actually a meta-grid, which imposes the relationship between each pair of blocks as follows:

$$(< ..b_i..b_j.. >, < ..b_i..b_j.. >) \Rightarrow b_i \text{ is to the left of } b_j \quad (1)$$

$$(< ..b_j..b_i.. >, < ..b_i..b_j.. >) \Rightarrow b_i \text{ is below } b_j \quad (2)$$

*This work was partially supported by the National Science Foundation under grant CCR-9912390, by the Texas Advanced Research Program under Grant No. 003658288, and by grants from IBM and Intel.

0-7803-6634-4/2001/$10.00

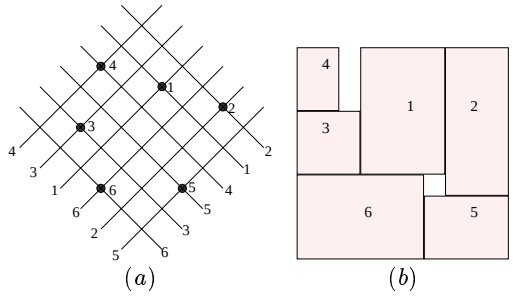

Figure 1: (a) Oblique grid for Sequence pair (<4 3 1 6 2 5>, <6 3 5 4 1 2>) giving the relative positions of the blocks; (b) the corresponding packing. The dimensions for the 6 blocks are: 1(4×6), 2(3×7), 3(3×3), 4(2×3), 5(4×3), and 6(6×4).

As an example, figure 1 illustrates the grid and the packing. Consequently, given a sequence pair (X, Y), the horizontal relationship among blocks follows a horizontal constraint graph $G_h(V, E)$, which can be constructed as follows:

1. $V = \{s_h\} \cup \{t_h\} \cup \{v_i | i = 1, ..., n\}$, where v_i corresponds to a block, s_h is the source node representing left boundary, and t_h is the sink node representing right boundary;
2. $E = \{(s_h, v_i) | i = 1, ..., n\} \cup \{(v_i, t_h) | i = 1, ..., n\} \cup \{(v_i, v_j) |$ block i is to the left of block $j\}$;
3. Vertex weight = width of block i for vertex v_i, but 0 for s_h and t_h.

The vertical constraint graph $G_v(V, E)$ can be similarly constructed.

Both $G_h(V, E)$ and $G_v(V, E)$ are vertex weighted, directed, acyclic graphs, so a longest path algorithm can be applied to determine the x and y coordinates of each block. The coordinates of a block are the coordinates of the lower-left corner of the block. The construction of constraint graphs G_h and G_v takes $\Theta(n^2)$ time. The longest path computation can be done in $O(n + m)$ time, where $n = \#vertices$, and $m = \#edges$ in the graph. The overall time for translating a sequence pair to a floorplan is then $\Theta(n^2)$.

3. Fast Evaluation of Sequence Pair in $O(n \log \log n)$ Time

In this section, we first introduce the concept of longest common subsequence and its relation to the evaluation of sequence pair. Then, we describe an efficient data structure needed in our algorithm. Finally, the algorithm to evaluate a sequence pair in $O(n \log \log n)$ time is presented. Our algorithm is an improvement of the $O(n \log n)$ algorithm in [16].

3.1 Longest Common Subsequence for Weighted Sequence Pair

We now show that sequence pair evaluation is equivalent to longest common subsequence computation. To make the paper self-contained, we introduce the following concepts.

Definition 1. *A weighted sequence is a sequence whose elements are in a given set S, while every element $s_i \in S$ has a weight. Let $w(s_i)$ denote the weight of s_i.*

In this paper, we assume $\forall s_i \in S,\ w(s_i) \geq 0$.

Definition 2. *Given two weighted sequences X and Y, a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y. For example, <1 2> is the common subsequence of <1 5 2> and <4 1 2 5>.*

Definition 3. *The length of a common subsequence $Z =< z_1\ z_2\ ...\ z_n >$ is $\sum_{i=1}^{n} w(z_i)$.*

In other words, the longest common subsequence for a sequence pair is the common subsequence of the two sequences with maximum length. In the following, let **LCS** denote longest common subsequence and $lcs(X, Y)$ denote the length of the longest common subsequence of X and Y.

Given the block set $B = \{1, ..., n\}$ and sequence pair (X, Y), a path from s_h in horizontal constraint graph G_h corresponds to a common subsequence of (X, Y), and vice versa. Figure 2(a) illustrates a path in G_h corresponding to a common subsequence of (X, Y). Let X^R denote the reverse of X. Then a path from s_v in vertical constraint graph corresponds to a common subsequence of (X^R, Y), and vice versa. Figure 2(b) illustrates a path in G_v corresponding to a common subsequence of (X^R, Y).

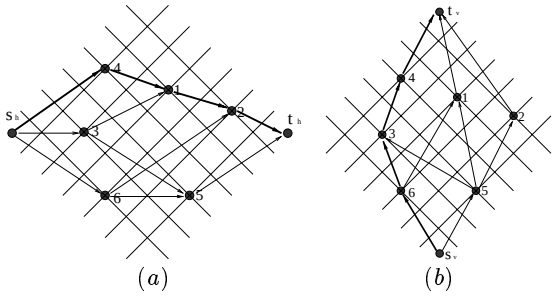

Figure 2: (a): in horizontal constraint graph $G_h(V, E)$, a path $s_h \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow t_h$ corresponds to <4 1 2>, a common subsequence of (X, Y)=(<4 3 1 6 2 5>, < 6 3 5 4 1 2>); (b): in vertical constraint graph $G_v(V, E)$, a path $s_v \rightarrow 6 \rightarrow 3 \rightarrow 4 \rightarrow t_v$ corresponds to <6 3 4>, a common subsequence of (X^R, Y)=(<5 2 6 1 3 4>, <6 3 5 4 1 2>).

Suppose there is a block b in the sequence pair (X, Y). Let $(X, Y) = (X_1 b X_2, Y_1 b Y_2)$. Then $(X^R, Y) = (X_2^R b X_1^R, Y_1 b Y_2)$. We can see a path from s_h to b in the horizontal constraint graph corresponds to a common subsequence of (X_1, Y_1). Similarly, a path from s_v to b in the vertical constraint graph corresponds to a common subsequence of (X_2^R, Y_1). Note that the coordinates of a block are the coordinates of the lower-left corner of the block. Thus, if $w(i)$= width of block i, $lcs(X_1, Y_1)$ is the x-coordinate of block b, and $lcs(X, Y)$ is the width of the block placement. Similarly, if $w(i)$ = height of block i, $lcs(X_2^R, Y_1)$ is the y-coordinate of block b, and $lcs(X^R, Y)$ is the height of the block placement.

3.2 Data Structure

We now describe a data structure which is needed in our algorithm. It is an efficient implementation of Priority Queue over the domain $\{1, ..., n\}$ [18]. The data structure is a complete binary tree with $n + 1$ leaves on the lowest level. The leaves of the binary tree on the lowest level are numbered consecutively from the left so that they corresponds to the index domain $\{0, 1, ..., n\}$. Each index therefore defines a unique path to the root of the tree. Only these leaves, called *bucket*, can contain elements. The non-empty buckets are kept in the *bucket list*, a doubly-linked list sorted on indices. Bucket 0 is used as the header for the bucket list, and is always present in the list. For convenience, we refer to the leaf corresponding to a non-empty bucket also as *index*. The space for the entire tree is allocated in the beginning. However, the "visible" part of the tree is dynamically changing, since as much of the tree as possible is left unconstructed. We shall construct the path for an index i whenever the corresponding bucket becomes non-empty. We shall also delete

that portion of the path which is no longer needed when the bucket becomes empty again. Thus, the bucket list is maintained as follows. Whenever a new bucket is inserted, its path would be constructed rootward until the new path intersects the path of some non-empty bucket inserted previously. Then this path would be followed leafward 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.

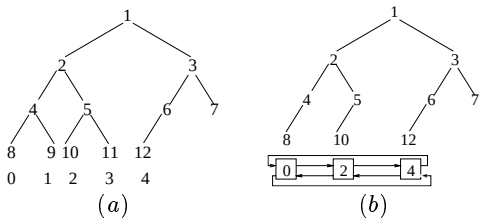

Figure 3: (a): host tree $H = \{1, ..., 12\}$ for the index domain $\{1, 2, 3, 4\}$, height $h = 3$; (b): a visible tree T for buckets $\{2, 4\}$ embedded in H. The bucket list is shown below T.

The complete binary tree (called *host tree*) is represented as $H = \{1, ..., 2^h + n\}$ where n is the size of the index domain and h is the height of the host tree, $h = \lceil \log(n+1) \rceil$. Each node q in $H - \{1\}$ is a child of $\lfloor q/2 \rfloor$. If q is even, q is the *left* child; if q is odd, q is the *right* child. Node 1 is the root of H. As an example, Figure 3(a) illustrates a host tree $H = \{1, ..., 12\}$ for the index domain $\{1, 2, 3, 4\}$. Buckets on the list comprise the leaves of a binary tree T (called *visible tree*). Visible tree T is then defined as the union of all paths $(1 \rightarrow f)$ in H for which leaf f corresponds to a bucket on the bucket list. Figure 3(b) illustrates a visible tree T embedded in H.

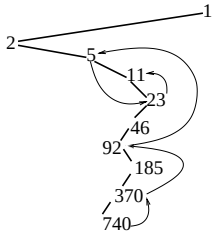

Figure 4: A chain of a binary search nodes from 740 to 11 on path $(1 \rightarrow 740)$ in a visible tree T with height $h = 9$.

Let D be the length of the smallest interval between the indices in the bucket list covering the index newly inserted or deleted, i.e., the distance between the two neighbors of the index newly inserted or deleted. Then we have $D \leq n$. Therefore, the length of the traversal path from the index newly inserted or deleted to its nearest index will be $O(\log D)$. During the operations of insertion and deletion, traversal or construction of entire new path segments will cost time proportional to the length of the path segment involved, which is $\Theta(\log D)$ in worst case. If we can traverse or construct only a logarithmic number of nodes in each new path segment, then the time for each operation will be reduced to $O(\log \log D)$. It is always possible since the host tree H is fixed hierarchically. A binary search on the path $(q \rightarrow f)$ from f for node q takes $O(\log k)$ time, where k is the length of the path. As an example, Figure 4 shows the binary search from node 740 for node 11 on path$(1 \rightarrow 740)$ in a tree T with height $h = 9$.

It was shown in [18] that insertion and deletion of an element on such a data structure take $O(\log \log D)$ time. With the help of the doubly-linked bucket list, the operations of finding *neighbor* (*predecessor* and *successor*) and extracting *MIN* and *MAX* take constant time.

3.3 Fast Evaluation Algorithm

In the section, we only focus on how to compute x coordinates for all the blocks, since computing y coordinates is similar to deal with the sequences (X^R, Y).

In order to describe the algorithm, we first introduce the involved data structures. Assume the blocks are $1...n$, and the input sequence pair is (X, Y). Both X and Y are then a permutation of $\{1...n\}$. Block position array $POS[b], b = 1...n$ is used to record the x or y coordinate of block b depending on the weight $w(b)$ equals to the width or height of block b respectively. The array $MATCH[b], b = 1...n$ is constructed to indicate the indices of b in X and Y, i.e. $MATCH[b].x = i$ and $MATCH[b].y = j$ if $b = X[i] = Y[j]$. Let the host tree be $H = \{1, ..., 2^h + n\}$ where $h = \lceil \log(n+1) \rceil$, the bucket list be $BUCKL$. Each bucket has an *index* and stores *value*. Let $BUCKL[index]$ denote its value. $BUCKL[index]$ records the length of candidates of the longest common subsequence. The algorithm is shown below.

Algorithm Eval-Seq(X,Y)
1. Initialize_Match_Array $MATCH$;
2. Initialize H, insert the initial index 0;
3. Initialize $BUCKL$ with $BUCKL[0] = 0$;
4. **FOR** $i = 1$ **TO** n **DO**
5. $b = X[i]$;
6. $p = MATCH[b].y$;
7. insert p to H and $BUCKL$;
8. $POS[p] = BUCKL[predecessor(p)]$;
9. $BUCKL[p] = POS[p] + w(b)$;
10. discard the successors of p from H and $BUCKL$ whose value $\leq BUCKL[p]$;
11. **RETURN** $BUCKL[index_{max}]$;

Theorem 1. *Algorithm Eval-Seq reports $lcs(X, Y)$ and records the position of each block in $O(n \log \log n)$ running time with $O(n)$ space requirement for a sequence pair (X, Y).*

Proof As described before, the position of block b is $lcs(X[1...(i-1)], Y[1...(j-1)])$, where $X[i] = Y[j] = b$. Assume the last element of the LCS of $(X[1...(i-1)], Y[1...(j-1)])$ is b'. Thus, $MATCH[b'].x \leq i-1$ and $MATCH[b'].y \leq j-1$. We get

$$\begin{aligned} & lcs(X[1...(i-1)], Y[1...(j-1)]) \\ = \; & lcs(X[1...(MATCH[b'].x - 1)], \\ & \quad Y[1...(MATCH[b'].y - 1)]) + w(b') \end{aligned}$$

The bucket list is kept in increasing order on index. The element in the bucket list with greater index but less value is useless to subsequent LCS computation. In line 10, the useless elements are deleted. Therefore, $predecessor(p)$ (now, $p = MATCH[b].y = j$) will be $MATCH[b'].y$ (otherwise, b' can not be the last element of the LCS). Induction on i would prove the correctness of recording the position of each block. $BUCKL[index_{max}]$ is clearly $lcs(X, Y)$.

Note that $H = \{1, ..., 2^h + n\}$ and $h = \lceil \log(n+1) \rceil$. Thus, $n + 1 \leq 2^h < 2(n+1)$, and then $|H| < 3n + 2$. The size of bucket list is $O(n)$. Thus, the space requirement is $O(n)$.

Now we use **amortized analysis** to prove that its running time is $O(n \log \log n)$. The initializations in line 1,2 and 3 therefore take $O(n)$ time. In line 4, the loop has n iterations. Line 5,6,8 and 9 take constant time each. As described before, the insertion in line 7 takes $O(\log \log D)$ time. As maintained, the bucket list is kept in increasing order on both index and value before the insertion of the new element. During the discarding in line 10 after the new element is inserted, we only need to check its successors. We keep checking until an element with greater value than $BUCKL[p]$ is found. As shown previously, discarding

one element from H and $BUCKL$ takes $O(\log\log D)$ time. Suppose there are d_k elements discarded during the kth iteration. Let D_{max} denote the maximum of all the D's. Thus, the total running time is

$$O(n\log\log D_{max}) + \sum_{k=1}^{n} d_k \cdot O(\log\log D_{max})$$

Since there are at most n elements discarded, $\sum_{k=1}^{n} d_k \leq n$. Therefore, the total running time is $O(n\log\log D_{max})$. Since $D_{max} \leq n$, this completes the proof. □

4. Placement with Constraints

In floorplanning, it is useful if users are allowed to specify some placement constraints in the final packing. In this section, we apply the fast method to evaluate sequence pair in block placement with constraints. The main constraints we are considering are pre-placed constraint, range constraint and boundary constraint.

4.1 Placement Constraints

The placement constraints can be formally defined as follows.

Definition 4. **Pre-placed constraint** *Given a block b with fixed orientation, and a point (x_1, y_1), block b must be placed with its lower-left corner at (x_1, y_1) in the final packing.*

Definition 5. **Range constraint** *Given a block b and rectangular region $R = \{(x,y)|x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$, block b must be placed inside R in the final packing.*

Definition 6. **Boundary constraint** *Given a block b, it must be placed on one of the four sides: on the left, on the right, at the bottom or at the top of the final packing.*

These constraints are useful in floorplanning. Floorplan with obstacles can be solved by treating the obstacles as pre-placed blocks. Floorplan with irregular boundaries can also be solved by treating the protruding parts along the boundaries as pre-placed blocks. Range constraint problem is a generalization of pre-placed constraint because any pre-placed constraint can be written as a range constraint by specifying the rectangular region to be of the same size as the block itself. Floorplanning is usually done hierarchically in which blocks are grouped into different units and floorplanning is done independently for each unit. If some blocks are constrained to be packed along the boundaries of the unit so that they can abut with some others in the neighboring units, then boundary constraint applies.

Since pre-placed constraint is a special case of range constraint, we will only consider range constraint in the following.

4.2 "Dummy" Block

In order to pack blocks with placement constraints, special blocks called *dummy blocks* are introduced. Let W be the width and H be the height of the chip. We use the notion $BP(W,H)$ to denote the problem of placing the given blocks onto the chip. Given a block b with range constraint $R = \{(x,y)|x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$ in problem $BP(W,H)$, four dummy blocks will be added: one is on the left of b, l_b with width x_1 and height 0; one is on the right of b, r_b with width $W - x_2$ and height 0; one is below b, w_b with width 0 and height y_1; the last one is above b, v_b with width 0 and height $H - y_2$. Figure 5(a) shows the 4 dummy blocks around a block with range constraint. Given a block b on the left boundary of packing, a dummy block r_b with width $W - width(b)$ and height 0 will be added on the right of b. Given a block b on the right boundary, a dummy block l_b with width $W - width(b)$ and height 0 will be added on the left of b. Given a block b on the lower boundary, a dummy block v_b with width 0 and height $H - height(b)$ will be added above b. Given a block b on the upper boundary, a dummy block w_b with width 0 and height $H - height(b)$ will be added below b. Figure 5(b) illustrates the dummy blocks for boundary constraint.

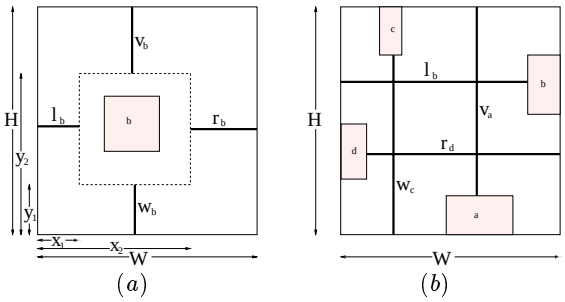

Figure 5: (a): 4 dummy blocks: l_b, r_b, w_b, v_b, around block b with a range constraint $R = \{(x,y)|x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$ in $BP(W,H)$; (b): dummy blocks for boundary constraint: block a, b, c and d will be placed on the lower, right, upper, and left side of the chip respectively.

Dummy blocks would not necessarily appear in sequence pair. It is equivalent to add additional edges in the constraint graph to meet range/boundary constraint. [6] introduced an additional edge from source to a pre-placed block (the edge is equivalent to l_b or w_b), followed by an adaption to handle pre-placed constraint. In the paper, we introduce 4 dummy blocks for a pre-placed/range constraint. Each dummy block is equivalent to an additional edge. r_b or v_b is equivalent to an edge from the block to the sink (t_h or t_v). For a block b with range constraint $R = \{(x,y)|x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$ and 4 dummy blocks: l_b, r_b, w_b and v_b, an edge (s_h, b) with weight $width(l_b)$ and an edge (b, t_h) with weight $width(r_b)$ are added into the horizontal constraint graph G_h; and an edge (s_v, b) with weight $height(w_b)$ and an edge (b, t_v) with weight $height(v_b)$ are added into the vertical constraint graph G_v. For blocks with boundary constraint, additional edges can be added similarly into G_h and G_v. As an example, Figure 6 shows G_h and G_v with a range constraint.

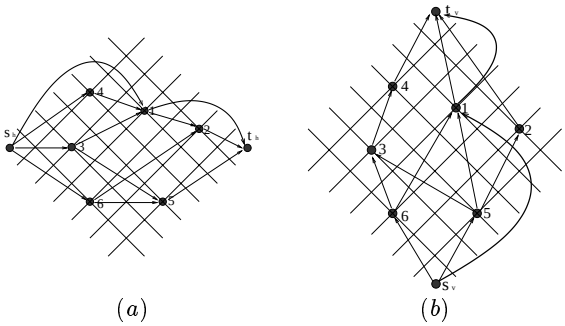

Figure 6: Sequence pair (X,Y)=(<4 3 1 6 2 5>, <6 3 5 4 1 2>), 1 is a block with a range constraint, (a): G_h with additional edges imposed by range constraint; (b): G_v with additional edges imposed by range constraint.

Given a placement satisfying the constraints, a sequence pair can be obtained by gridding operation in [5]. This observation is stated in the following lemma.

Lemma 1. *Any placement that satisfies the given placement constraints is represented by a sequence pair.*

4.3 Feasible Sequence Pair

When constraints are introduced into sequence pair, there may not exist packing for some sequence pairs. *Feasible sequence pair* can be defined as follows.

Definition 7. **Feasible sequence pair** *If there exists a packing which meets all the constraints imposed by the sequence pair, then the sequence pair is feasible. Otherwise, it is infeasible.*

We have the following theorem.

Theorem 2. *A sequence pair (X,Y) is feasible if and only if the length of the longest path from s_h to t_h in G_h is no more than W and the length of the longest path from s_v to t_v in G_v is no more than H.*

Proof If the length of the longest path from s_h to t_h in G_h is no more than W, then $\forall b$, the length of the longest path from s_h to b (denoted as $max|s_h \rightarrow b|$) is no more than $W - width(b)$. Similarly, $\forall b$, $max|s_v \rightarrow b| \leq H - height(b)$ in G_v. By longest path computations in both G_h and G_v, the coordinates of each block can be determined, which imposes a non-overlapping packing for all the blocks. Therefore, the sequence pair (X,Y) is feasible.

On the other hand, if $max|s_h \rightarrow t_h| > W$ in G_h or $max|s_v \rightarrow t_v| > H$ in G_v, then there is no non-overlapping packing for the blocks along the longest path to satisfy all the constraints. That means that all the blocks can not fit in the given frame $W \times H$ or some block with range constraint is placed out of its range or some block with boundary constraint is not placed on the boundary. Then, the sequence pair (X,Y) will be infeasible. □

4.4 Modified Algorithm

Our fast $O(n \log\log n)$ algorithm can be adapted to evaluate the sequence pair in block placement with such constraints without increasing its asymptotic complexity.

Note that the calculations of x and y coordinates of all the blocks are done independently by evaluating (X,Y) and (X^R,Y) respectively. The evaluation of (X,Y) can be modified as follows. A "sink" variable t is introduced to record the intermediate *lcs* imposed by dummy blocks in placement constraints (i.e., the longest path to t_h in G_h).

Algorithm Eval-Seq(X,Y)
1. Initialize_Match_Array $MATCH$;
2. Initialize H, insert the initial index 0;
3. Initialize $BUCKL$ with $BUCKL[0] = 0$;
4. $t = 0$;
5. **FOR** $i = 1$ **TO** n **DO**
6. $b = X[i]$;
7. $p = MATCH[b].y$;
8. insert p to H and $BUCKL$;
9. $POS[p] = BUCKL[predecessor(p)]$;
10. if l_b exists, $POS[p] = max(POS[p], width(l_b))$;
11. $BUCKL[p] = POS[p] + w(b)$;
12. if r_b exists, $t = max(t, BUCKL[p] + width(r_b))$;
13. discard the successors of p from H and $BUCKL$ whose value $\leq BUCKL[p]$;
14. **RETURN** $max(t, BUCKL[index_{max}])$;

The evaluation of (X^R,Y) can be modified similarly. It is clear that the modification does not increase the complexity.

Let $lcs'(X,Y)$ denote the return value of the modified algorithm, i.e. the length of the longest common subsequence with the presence of dummy blocks. The following claim is directly implied.

Corollary 1. *A sequence pair (X,Y) is feasible if and only if $lcs'(X,Y) \leq W$ and $lcs'(X^R,Y) \leq H$.*

If there is a pre-placed block, or a block with left/right boundary constraint, then clearly $lcs'(X,Y) \geq W$. Similarly, if there is a pre-placed block, or a block with lower/upper boundary constraint, $lcs'(X^R,Y) \geq H$. Therefore, we obtain the following result.

Corollary 2. *If there is a pre-placed block, or there exist a block with left/right boundary constraint and a block with lower/upper boundary constraint, then the necessary and sufficient condition of feasible sequence pair (X,Y) will be $lcs'(X,Y) = W$ and $lcs'(X^R,Y) = H$.*

4.5 Unified Cost Function

When placement constraints are introduced, some sequence pairs may be infeasible. An intuitive way to evaluate infeasible sequence pairs is to assign them an infinite cost. However, this will seriously affect the smoothness of stochastic search process. Note that if a sequence pair (X,Y) is infeasible then $lcs'(X,Y) > W$ or $lcs'(X^R,Y) > H$. Thus, we treat an infeasible sequence pair as if it is feasible, and regard the "area" as

$$A = lcs'(X,Y) \cdot lcs'(X^R,Y)$$

Therefore, we get a unified cost function

$$C = \alpha A + \beta W$$

where W is interconnect cost, and α and β are coefficients for balancing between A and W.

The benefits of the unified cost function are: (i) no need to accumulate the error for each constrained block; (ii) no need to add an additional penalty term into the cost function; and (iii) fixing an infeasible sequence pair (adaption) is not necessary.

5. Experimental Results

We have implemented a block placement tool FAST-SP based on our fast sequence pair evaluation algorithm in C language on a SUN Sparc Ultra 1 (166 MHz). FAST-SP uses the technique of simulated annealing to search for an optimal placement with a special annealing schedule where a very large number of temperatures is used but only a small number of moves are made within each temperature.

Experimental results show that our sequence pair evaluation algorithm is indeed very fast – we can easily evaluate a million sequence pairs within one minute for typical input size of placement problems. Compared to the $(n \log n)$ algorithm in [16], we achieved 7X speedup on average for all the MCNC benchmark placement problems. We also compared FAST-SP with block placement algorithms based on O-tree [12, 13] and B*-tree [14] on MCNC benchmark problems. (Note that O-tree and B*-tree have reported the best results for these benchmark problems.) Table 1 lists the runtime and chip area for O-tree, B*-tree and FAST-SP. All experiments were based on hard blocks and optimizing chip area only. The runtime and areas for O-tree and B*-tree are directly taken from [13, 14] respectively[1]. The results for FAST-SP are based on Sun Sparc Ultra 1(166MHz) with 128M memory, while B*-tree is on Sun Sparc Ultra 1(200MHz) with 256M memory and O-tree is on Sun Ultra 60 which is significantly faster. For all MCNC benchmark problems, FAST-SP has obtained the best results ever reported in the literature with significantly less runtime. For example, the best known result for ami49 (36.8 mm^2) was obtained by a B*-tree based algorithm using 4752 seconds, and FAST-SP obtained a better result (36.5 mm^2) in 31 seconds. Upon using longer runtime, we can still get a little better results for ami33 and ami49. For example, 36.46 mm^2 was obtained for ami49 in 77 seconds, and 1.195 mm^2 for ami33 in 74 seconds.

We also tested FAST-SP on problems with placement constraints. It should be noted that W and H need to

[1]We have been informed that there is a new implementation of B*-tree[19] improving the results in [14].

Table 1: Comparison of runtime and area among O-tree(in Sun Ultra60), B*-tree(in Sun Ultra1) and FAST-SP(in Sun Ultra1).

circuit	block	O-tree		B*-tree		FAST-SP	
		time(Ultra60) (s)	area (mm^2)	time(Ultra1) (s)	area (mm^2)	time(Ultra1) (s)	area (mm^2)
apte	9	11	46.92	7	46.92	1	46.92
xerox	10	38	20.21	25	19.83	14	19.80
hp	11	19	9.159	55	8.95	6	8.947
ami33	33	119	1.242	3417	1.27	20	1.205
ami49	49	526	37.73	4752	36.80	31	36.50

be determined initially in problem $BP(W, H)$. We choose W and H under some given aspect ratio such that $W \cdot H$ equals 150% of the sum of the areas of all the blocks. (In MCNC benchmarks, the width and height of initial floorplan is given.) Since we want to minimize the total area of the bounding box of final packing, we decrease W and H when a feasible sequence pair is met. Three kinds of decreasing schemes are randomly chosen in our experiments: decrease W only, or H only or both. FAST-SP performed well on all test problems. One of the test problems we used is derived from the MCNC benchmark ami49. 3 blocks (1,2,3) are selected to be assigned a range constraint, and 4 blocks are selected to be placed on the 4 sides of the chip (boundary constraint): 4 on the lower boundary, 5 on the upper boundary, 6 on the left and 7 on the right. Figure 7 displays the final packing. The dead space is 6.0%, and the running time is 58 seconds.

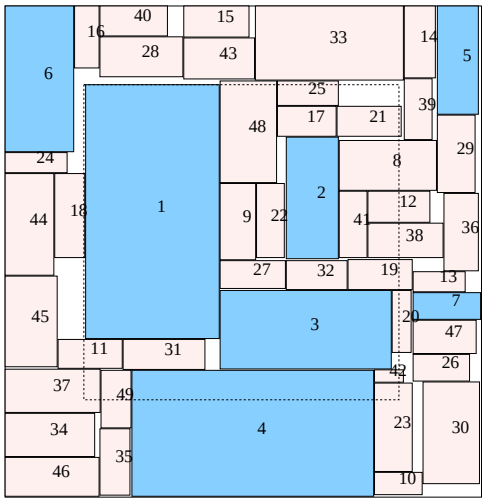

Figure 7: The result packing of ami49. Block 1,2 and 3 are constrained to be placed within the dashed rectangle, and block 4,5,6 and 7 are constrained on the 4 boundaries respectively.

6. Concluding Remarks

In this paper, we presented FAST-SP, a fast block placement algorithm based on sequence pair. FAST-SP has two significant improvements: 1) the runtime of evaluating a sequence pair is reduced to $O(n \log \log n)$. As a result, we can easily evaluate a million sequence pairs within one minute for typical input size of placement problems. 2) it can handle placement constraints such as pre-placed constraint, range constraint and boundary constraint in $O(n \log \log n)$ time. We presented the necessary and sufficient condition of feasible sequence pair and proved that any placement satisfying such constraints is represented by a sequence pair. Finally, a unified cost function was derived for the evaluation of both feasible and infeasible sequence pair. Experimental results showed that we obtained the best results ever reported in the literature with significantly less runtime.

7. References

[1] Semiconductor Industry Association. National Technology Roadmap for Semiconductors, 1997.

[2] N. Sherwani. Algorithms for VLSI Physical Design Automation, Kluwer Academic Publishers, Boston, 1995.

[3] M. Sarrafzadeh and C.K. Wong. An Introduction to VLSI Physical Design, McGraw Hill, 1996.

[4] J. Parkhurst, N. Sherwani, S. Maturi, D. Ahrams, and E. Chiprout. "SRC physical design top ten problems", ISPD-99, pp. 55-58, 1999.

[5] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. "VLSI module placement based on rectangle-packing by the sequence pair", *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, vol. 15:12, pp. 1518-1524, 1996.

[6] H. Murata, K. Fujiyoshi, and M. Kaneko. "VLSI/PCB placement with obstacles based on sequence pair", ISPD-97, pp. 26-31, 1997.

[7] H. Murata, and E.S. Kuh. "Sequence-pair based placement method for hard/soft/pre-placed modules", ISPD-98, pp. 167-172, 1998.

[8] J. Xu, P.N. Guo, and C.K. Cheng. "Rectilinear block placement using sequence pair", ISPD-98, pp. 173-178, 1998.

[9] M. Kang, and W.W.M. Dai. "Topology constrained rectilinear block packing for layout reuse", ISPD-98, pp. 179-186, 1998.

[10] K. Fujiyoshi, and H. Murata. "Arbitrary convex and concave rectilinear block packing using sequence pair", ISPD-99, pp. 103-110, 1999.

[11] F. Balasa, and K. Lampaert, "Module placement for analog layout using the sequence pair representation", DAC-99, pp. 274-279, 1999.

[12] P.N. Guo, C.K. Cheng, and T. Yoshimura, "An O-tree representation of non-slicing floorplans and its applications", DAC-99, pp. 268-273, 1999.

[13] Y. Pang, C.K. Cheng, and T. Yoshimura, "An enhanced perturbing algorithm for floorplan design using the O-tree representation", ISPD-2000, pp. 168-173, 2000.

[14] Y.C. Chang, Y.W. Chang, G.M. Wu, and S.W. Wu. " B*-trees: a new representation for non-slicing floorplans", DAC-2000, pp. 458-463, 2000.

[15] T. Takahashi. "An algorithm for finding a maximum-weight decreasing sequence in a permutation, motivated by rectangle packing problem ", *Technical Report of IEICE, VLD96*, vol. VLD96, No. 201, pp. 31-35, 1996.

[16] X. Tang, R. Tian, and D.F. Wong. "Fast evaluation of sequence pair in block placement by longest common subsequence computation", DATE-2000, pp. 106-111, 2000.

[17] P. Van Emde Boas, R. Kaas, and Z. Zijlstra. "Design and implementation of an efficient priority queue", *Mathematical Systems Theory* 10, pp. 99-127, 1977.

[18] D.B. Johnson. "A priority queue in which initialization and queue operations take $O(\log \log D)$ time", *Mathematical Systems Theory* 15, pp. 295-309, 1982.

[19] Y.W. Chang. Personal communication.