# Architecture and Details of a High Quality, Large-Scale Analytical Placer

Andrew B. Kahng Sherief Reda Qinke Wang
Computer Science and Engineering Dept.
Univ. of CA, San Diego
La Jolla, CA 92093
Email:{abk@ucsd.edu, sreda@cs.ucsd.edu, qiwang@cs.ucsd.edu}

**Abstract**

Modern design requirements have brought additional complexities to netlists and layouts. Millions of components, whitespace resources, and fixed/movable blocks are just a few to mention in the list of complexities. With these complexities in mind, placers are faced with the burden of finding an arrangement of placeable objects under strict wirelength, timing, and power constraints. In this paper we describe the architecture and novel details of our high quality, large-scale analytical placer. The performance of our placer has been recently recognized in the recent ISPD-2005 placement contest, and in this paper we disclose many of the technical details that we believe are key factors to its performance. We describe (i) a new clustering architecture, (ii) a dynamically adaptive analytical solver, and (iii) better legalization schemes and novel detailed placement methods. We also provide extensive experimental results on a number of benchmark sets. On average, our results are better than the best published results by 3%, 14%, and 6% for the IBM ISPD'04, ICCAD'04, and ISPD'05 benchmark sets respectively. One of the goals of this paper is to also provide enough details to enable possible future replication of our methods.

## 1 Introduction

Beside enormous sizes, modern VLSI circuits exhibit a wide range of features that require careful handling by physical design tools. These features include thousands of fixed as well as movable blocks, a large number of I/O pads that are not necessary on the peripheral of the layout, millions of standard cells, and a large amount of whitespace for routing and timing requirements. These features are challenging to handle with existing placers, as has been demonstrated in the recent ISPD-2005 placement contest [28].

To tackle these challenges, modern placers combine a wide range of techniques and components. For instance, (i) clustering is routinely used to cut down runtime and enable scalable implementations [33, 19, 5], (ii) core placement engines are based on min-cut [10, 39, 4] and/or analytical solvers [37, 16, 11], (iii) legalization component [27, 7, 21] (iv) iterative improvement heuristics [17, 32], and (v) detailed placers and whitespace distributers [8, 15, 38, 24, 21]. All these techniques must readily handle the presence of blocks - whether fixed or movable - and whitespace. Furthermore, all components have to be carefully tuned to squeeze out every possible increment of Quality of Result (QOR).

In this paper we describe the architecture and details of our placer APLACE for the ISPD-2005 contest [28]. We believe this requires an additional treatment for three reasons: (1) the ideas and changes to our original placer are numerous and require a separate treatment, (2) many of the these ideas were conceived after the submission of the three-page brief description at ISPD and during the two months run-up before the contest, and (3) the results of our new placer on existing benchmark sets are some of the best unpublished. The performance of our placer has been recently recognized in the ISPD-2005 contest: our placer was superior to all other entries of the contest, on all benchmarks[1]. Given these reasons, we have decided to cover all the architectural aspects as well technical details of our placer in this paper. We also provide extensive experimental results on most recent benchmark sets.

The outline architecture of our placement tool is given in Figure 1. Clustering is used as an initial pre-processing step to condense the netlist in a multi-level fashion to just around a couple of thousand components. Global placement works on the clustered netlist until a "decent" spreading is achieved. At that point, unclustering breaks down the clusters to reveal the next level of clustering. The process of global placement and unclustering is iterated until the original flat netlist is well spread. Then, legalization assigns valid positions for all movable components with no overlaps. Legalization typically incurs an increase in wirelength of the placement. The ensuing detailed placement phase attempts to recover any loss of quality due to legalization. Detailed placement is comprised of three phases: (i) global moving where cells are moved globally to reduce wirelength, (ii) whitespace distribution where whitespace is optimally distributed to minimize wirelength while maintaining the relative cell ordering in every layout row, and (iii) cell order polishing where successive small windows of cells are optimally re-ordered. The three phases of detailed placement may be iterated until negligible improvements in wirelength are observed.

The organization of this paper is as follows. Section 2 gives the details of the clustering and unclustering phases. Section 3 discusses various global placement ideas and details. Section 4 provides the details of our legalization scheme and various phases of detailed placement. Section 5 gives experimental results for the IBM ISPD'04, ICCAD'04, and the ISPD'05 benchmark sets. Finally, Section 6 summarizes and highlights the contributions of this work.

## 2 Clustering and Unclustering

Executing an analytical global placer on a flat cell design might give the best placement results - depending on how scalable the placer is - but nevertheless can incur extremely long runtimes. Clustering offers an attractive choice to reduce runtime, and with careful tuning this can have no impact on

[1]Dr. Gi-Joon Nam, the contest organizer, noted in a published interview that "these designs represent the real physical design challenges of today and tomorrow".

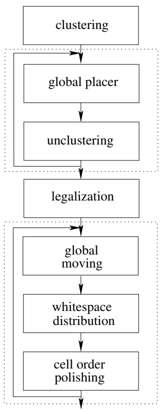

Figure 1: The outline of our placement flow.

placement quality. Our clustering approach can be viewed as a middle-ground between the top-down multi-level paradigm of MLPart [6] and hMetis[25] on one side, and fine-grain clustering [19] and semi-persistent clustering [5] on the other side.

Our clustering pre-processes the input netlist to reduce its size to only a couple of thousand clusters. However, this clustering is executed in a multi-level paradigm, where each clustering level is about tenth the size of the previous clustering level. For example, if the input size is around two million objects - roughly the size of the largest circuit in the IBM ISPD'05 benchmark set - then the clustering hierarchy is around four levels with vertex cardinalities: 2M, 200k, 20k, and 2k. After clustering, the pre-processed netlist is given to the global analytical placer to operate on. The global placer keeps on solving the netlist until it achieves a non-overlapping, or a "sufficiently" small overlapping placement. At this point, unclustering is triggered and the components of the next clustering hierarchy level replace the existing components. The components of an unclustered object are initially placed at the center location of their component with a slight random perturbation.

To move from one clustering level to the next during the initial pre-processing step, we use the *best choice* heuristic [5] with tight control on cluster area and using lazy updates. This can be summarized as follows. Initially, the *affinity* of every object $u$ to its neighbors is calculated and the neighbor object with largest affinity is declared the *closest*; its affinity becomes the *score* of node $u$. The affinity between a pair of objects $u$ and $v$ is the total weight of the hyperedges joining them divided by their area (similar to first choice clustering [26]), where the weight of a hyperedge is inversely proportional to its cardinality. After the scores of all nodes are calculated, they are inserted in a priority queue that is sorted in a descending order. Clustering then proceeds as follows: (i) cluster the best node - essentially the one with highest score - with its closest neighbor, unless the node is marked "invalid" or clustering violates the area constraints, (ii) update the netlist and insert the score of the new clustered node in the proper position in the priority queue, and (iii) mark the neighbors of the new node as invalid. The clustering algorithm is given in Figure 2.

The interaction between the best choice clustering heuristic with the multi-level paradigm in the presence of different cell areas creates an unbalanced cluster hierarchy at each level, and the boundaries of such a hierarchy must be clearly marked to allow correct unclustering. For example, Figure 3 gives a possible clustering hierarchy, where we label each node with its level. We can clearly see that a node might take part in a

**Input:** Flat netlist.
**Output:** Clustered netlist.

```
until number of clusters < 2000:
  target number of clusters = $\frac{\text{current number of clusters}}{\text{clustering ratio=10}}$.
  target cluster area (CA) = $\frac{\text{total cell area}}{\text{target number of clusters}} * 1.5$.
  for each object $u$:
    calculate the most affine neighbor to $u$ and $u$'s score.
  sort all objects by their score descendingly using a heap.
  until the target number of clusters is met:
    if (i) top of the heap $u$ is not marked invalid and
        (ii) clustering does not violate CA
        then cluster $u$ with its most affine neighbor.
    else if $u$ is marked invalid
        then recalculate its score, insert in heap and mark valid.
    else remove $u$ from the heap and continue.
    update netlist and calculate the new clustered object score.
    insert the new object into the heap.
    mark the neighbors of the new object invalid.
```

Figure 2: Clustering algorithm.

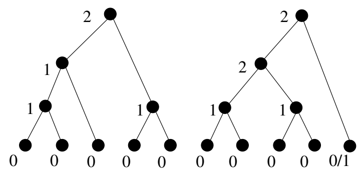

Figure 3: A multi-level clustering hierarchy, with a clustering ratio of 2 and a required final target number of clusters equal to 2. Each node is labelled with its position in the clustering hierarchy.

number of clusterings during the same level - as long as area constraints are not violated. Thus, during clustering it is important to remember the boundaries of the clustering hierarchy to allow exact reversal of the clustering process.

Another concern during clustering is what we call *clustering saturation*. In netlists with large numbers of fixed blocks and I/O pads, it is possible that clustering is not able to meet its final target number of clusters since a large number of clusters are just connected to fixed components. These fixed components slow down the clustering, causing saturation. In this case, we can *bypass* the fixed objects - especially the small ones - to allow further clustering. This bypassing can be achieved by adding an artificial net connecting all the neighbors of a fixed object together.

## 3 Global Placement

### 3.1 Constrained Minimization Formulation

We regard global placement as a *constrained nonlinear optimization problem*: We divide the placement area into uniform grids, and seek to minimize total half-perimeter wirelength (HPWL) under the constraint that total module area in every grid is equalized. The problem is expressed using the following formulation:

$$\begin{array}{ll} min & HPWL(\mathbf{x},\mathbf{y}) \\ s.t. & D_g(\mathbf{x},\mathbf{y}) = D \quad \text{for each grid } g \end{array} \tag{1}$$

where $(\mathbf{x},\mathbf{y})$ is the center coordinates of modules, $HPWL(\mathbf{x},\mathbf{y})$ is the total HPWL of the current placement, $D_g(\mathbf{x},\mathbf{y})$ is a density function that equals the total module area in grid $g$ and $D$ is a constant denoting the average module area over all grids.

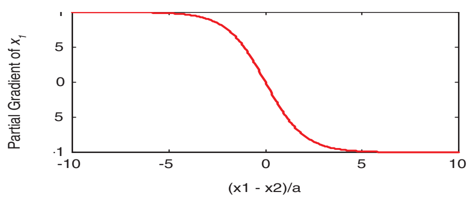

Figure 4: Partial wirelength gradient for $x_1$ as a function of $(x_1 - x_2)/\alpha$.

To solve the problem using nonlinear optimization techniques, first we need to have smooth wirelength and density functions.

### 3.1.1 LOG-SUM-EXP Wirelength Function

While wirelength and overall placement quality is typically evaluated according to HPWL, this "linear wirelength" function can not be efficiently minimized. In our placer, we use a **log-sum-exp** method to capture the linear HPWL while simultaneously obtaining the desirable characteristic of continuous differentiability. The log-sum-exp formula picks the most dominant terms among pin coordinates; it is proposed for wirelength approximation in [29] and applied in recent academic placers [22], [23], [13]. For a net $e$ with pin coordinates $\{(x_1,y_1),\ (x_2,y_2),\ \dots\ (x_n,y_n)\}$, the smooth wirelength function is

$$\begin{aligned} WL(e) = \quad & \alpha \cdot (log(\sum e^{x_i/\alpha}) + log(\sum e^{-x_i/\alpha})) + \\ & \alpha \cdot (log(\sum e^{y_i/\alpha}) + log(\sum e^{-y_i/\alpha})) \end{aligned} \tag{2}$$

where $\alpha$ is a smoothing parameter. $WL(e)$ is strictly convex, continuously differentiable and converges to $HPWL(e)$ as $\alpha$ converges to 0 [29].

Intuitively, for the overall placement problem, the smoothing parameter $\alpha$ can be regarded as a "significance criterion" for choosing nets with large wirelength to minimize. For example, for a two-pin net with pin coordinates $\{(x_1,y_1),\ (x_2,y_2)\}$, the partial gradient of the wirelength function $WL$ for $x_1$ is

$$\frac{\partial WL}{\partial x_1} = 1/(1+e^{(x_1-x_2)/\alpha}) - 1/(1+e^{(x_2-x_1)/\alpha}) \tag{3}$$

As shown in Figure 4, when the net length $|x_1 - x_2|$ is relatively small compared to $\alpha$, the partial gradient is close to 0; otherwise, the gradient is close to 1 or -1. It means that the length of long nets (relative to $\alpha$) will be minimized more efficiently than short nets when optimizing the wirelength function for the whole netlist. Our placer uses this important characteristic to facilitate the multi-level algorithm that will be described below in Section 3.3.

### 3.1.2 Bell-Shaped Potential Function

The density function $D_g$ in Equation 1 is also not smooth or differentiable. Function $D_g$ can be expressed as the following form:

$$D_g(\mathbf{x},\mathbf{y}) = \sum_v P_x(g,v) \cdot P_y(g,v) \tag{4}$$

where functions $P_x(g,v)$ and $P_y(g,v)$ denote the overlap between the grid $g$ and module $v$ along the $x$ and $y$ directions, respectively. For example, suppose we have a grid $g$ with width $w_g$ and a standard cell $c$. Since the size of cell $c$ is usually small relative to the grid size, we ignore the cell size and assume it to be a dot with unit area. Then function $P_x(g,c)$ is a 0/1 function as shown in Figure 5(a): $P_x(g,c)$ is 1 when the horizontal distance between grid $g$ and cell $c$, $d_x = |x_c - x_g|$ is less than $w_g/2$, and is 0, otherwise.

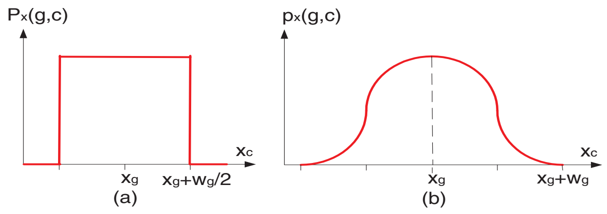

Figure 5: (a) "Rectangle-shaped" function $P_x(g,v)$; and (b) "Bell-shaped" smooth function $p_x(g,v)$.

Naylor et al. [29] propose to replace the above "rectangle-shaped" function with a "bell-shaped" function $p_x(g,c)$ as shown in Figure 5(b):

$$p_x(g,c) = \begin{cases} 1 - 0.5d_x^2/w_g^2 & (0 \le d_x \le w_g) \\ 0.5(d_x - 2w_g)^2/w_g^2 & (w_g \le d_x \le 2w_g) \end{cases} \tag{5}$$

This function is implemented in our original placer and has been proved effective. Since the function decides an "area potential" exerted by a cell to its nearby grids, we call it *area potential function*.

We follow the above idea and apply a similar "bell-shaped" area potential function in our current placer. Unlike the above potential function, our potential function also takes care of large blocks, as well as standard cells, and extends the scope of area potential according to the block size so that a larger block will have non-zero potential with respect to more nearby grids.

Suppose a module $v$ has a large width $w_v$. The scope of this module's $x$-potential is $w_v/2 + 2w_g$, i.e., every grid within horizontal distance of $w_v/2 + 2w_g$ from the module's center has a non-zero $x$-potential from this module. Therefore, the area potential function for the $x$-direction $p_x(g,v)$ becomes

$$p_x(g,v) = \begin{cases} 1 - a * d_x^2 & (0 \le d_x \le w_v/2 + w_g) \\ b * (d_x - 2w_g)^2 & (w_v/2 + w_g \le d \le w_v/2 + 2w_g) \end{cases} \tag{6}$$

where

$$\begin{aligned} a &= 4(w_v + 4w_g^2)/((w_v + 8w_g^2)(w_v + 2w_g)^2) \\ b &= 4/(w_v + 8w_g^2) \end{aligned} \tag{7}$$

so that the function is continuous when $d_x = w_v/2 + w_g$.

Similarly, we define a smooth $y$-potential function $p_y(g,v)$ and the non-smooth function $D_g$ in Equation 1 is replaced by a continuous function:

$$SD_g(\mathbf{x},\mathbf{y}) = \sum_v C_v \cdot p_x(g,v) \cdot p_y(g,v) \tag{8}$$

where $C_v$ is a normalization factor so that $\sum_g C_v \cdot p_x(g,v) \cdot p_y(g,v) = A_v$, i.e., each module $v$ has a total area potential equal to its area $A_v$.

## 3.2 Quadratic Penalty Method and Conjugate Gradient Solver

In the current version of our placer, we solve the constrained optimization problem in Equation 1 using the simple *quadratic penalty method*. That is, we solve a sequence of unconstrained minimization problems of the form

$$min \quad WL(\mathbf{x},\mathbf{y}) + \tfrac{1}{2\mu} \sum_g (SD_g(\mathbf{x},\mathbf{y}) - D)^2 \tag{9}$$

```
Conjugate Gradient Algorithm
Input:
  A high dimensional function $f(x)$
  Initial solution $x_0$
  Minimum step length $\varepsilon$
  Initial maximum step length $\gamma_0$
  Maximum number of iterations $N$
Output:
  Local minimum $x^*$
Algorithm:
  01. Initialize # iterations $k = 1$, step length $\alpha_0 = \infty$
      gradients $g_0 = 0$ and conjugate directions $d_0 = 0$
  02. For ($k < N$ and step length $\alpha_{k-1} > \varepsilon$)
  03.       Compute gradients $g_k = \nabla f(x_k)$
  04.       Compute Polak-Ribiere parameter $\beta_k = \frac{g_k^T(g_k - g_{k-1})}{||g_{k-1}||^2}$
  05.       Compute conjugate directions $d_k = -g_k + \beta_k d_{k-1}$
  06.       Compute step length $\alpha_k$ within $\gamma_{k-1}$
                 using Golden Section line search algorithm
  07.       Update new solution $x_k = x_{k-1} + \alpha_k d_k$
  08.       Update maximum step length $\gamma_k$ =MAX$\{\gamma_0, 2\gamma_{k-1}\}$
  17. Return minimum $x^* = x_k$
```

Figure 6: Conjugate Gradient Algorithm

for a sequence of values $\mu = \mu_k \downarrow 0$ and use the solution of the previous unconstrained problem as an initial guess for the next one.

Empirical studies show that the values of $\mu$ is very important to the solution quality. Theoretically, when the optimal solution of the unconstrained problem in Equation 9 is reached, the gradients derived from the wirelength term are opposite to those derived from the density penalty term. Therefore, we decide the initial $\mu$ according to the absolute values of wirelength and density gradients:

$$\mu_0 = \frac{1}{2} \frac{\sum_{x_i, y_j} \sum_{g} |SD_g - D| \cdot (|\frac{\partial SD_g}{\partial x_i}| + |\frac{\partial SD_g}{\partial y_j}|)}{\sum_{x_i, y_j} (|\frac{\partial WL}{\partial x_i}| + |\frac{\partial WL}{\partial y_j}|)} \tag{10}$$

After that, $\mu$ decreases by half: $\mu_{k+1} = 0.5\mu_k$

We solve the unconstrained problem in Equation 9 using the *Conjugate Gradient* (CG) method, as shown in Figure 6. The conjugate gradient method is quite useful in finding an unconstrained minimum of a high-dimensional function, even though the function is not convex. Also the memory required is only linear in the problem size, which makes it adaptable to large-scale placement problems.

## 3.3 Multi-Level Algorithm

Our placer applies a top-down multi-level algorithm to improve scalability. We use two different multi-level methods in the placer: (1) multiple levels of placeable objects and (2) multiple levels of grids.

### 3.3.1 Multiple Levels of Clusters

Before global placement, our placer builds up a hierarchy of clusters as described in Section 2, performs placement for each level of clusters and use the solution of the current level cluster placement as an initial guess for the next level placement problem.

Clustering reduces the number of placeable objects and thus speeds up the calculation of density penalty. For each level in the cluster hierarchy, we compute the density penalty by regarding a cluster as a square block with area equal to the total module area of the cluster. Moreover, the decrease of the number of variables also greatly reduces the number of conjugate gradient iterations required to obtain a good solution of the unconstrained optimization problem. For wirelength calculation, we assume modules to be located at the center of the cluster and only consider the inter-cluster parts of nets, which speeds up the wirelength calculation.

### 3.3.2 Multiple Levels of Grids

Beside the commonly used method of multiple cluster levels, our placer also employs multiple levels of grids to achieve better scalability and global optimization.

Various grid sizes provide different levels of relaxation for the constrained wirelength minimization problem in Equation 1. For example, in the optimal solution of the constrained minimization problem with a larger grid size, modules in the same grid are expected to cluster together instead of spread evenly over the grid in order to reduce total wirelength, although total module area in each grid is equal. However, this solution can be used as the initial solution for the placement problem constrained with finer grids, to obtain a more even module placement.

We adaptively modify the smoothing parameter $\alpha$ according to the grid size, instead of using a small constant value. For a wirelength minimization problem constrained with coarser grids, minimization of short nets (relative to the grid size) leads to undesirably clustered cells. Therefore, the value of $\alpha$ should be comparable to the grid size, so that only long nets (probably connecting modules in different grids) are "chosen" to be minimized and short nets (probably connecting modules in the same grid) are "ignored". Empirical studies show that better placement quality is obtained by setting $\alpha$ to half of the grid size.

Using an initial larger grid size and wirelength smoothing parameter in our placer not only leads to better global optimization, but also greatly speeds up the placer. As shown in Equation 6, the scope of modules' potential is proportional to the grid size. Therefore, a larger grid size helps to spread cells faster than a smaller grid size.

### 3.3.3 Top-Down Multi-Level Algorithm

Combining the two methods discussed in Sections 3.3.1 and 3.3.2, our top-down multi-level algorithm is described in Figure 7. Notations used are summarized as follows:

| | |
|---|---|
| $\alpha$ | wirelength smoothing parameter |
| $\varepsilon$ | minimum step length of CG solver |
| $f$ | unconstrained objective function |
| $N_l$ | the number of clusters at level $l$ |
| $L$ | the number of cluster levels |
| $\{Gradient_l(i)\}$ | a vector of conjugate gradients |
| $\{ClusterPosition_l(i)\}$ | a vector of cluster positions |

Subscript ranges, where not explicit, are: $l = 0, \ldots, L$; and $i = 1, \ldots, N_l$.

Initially, the global placements of all modules is initialized to be at the center of the placement area. Unlike most analytical placers, our placer can also place circuits without fixed pads, or simultaneously place modules and peripheral/area I/O pads. In this case, the placeable objects are initially placed randomly close to the center.

For each level in the cluster hierarchy, the grid size is determined according to the number of clusters, assuming the total module area of each cluster is similar. We then decide most important control parameters according to the grid size.

After that, the global placer basically uses the CG optimizer to solve the constrained wirlength minimization prob-

```
Top-Down Multi-Level Algorithm
Input:
  User-defined target density discrepancy TargetDisc
  User-defined max #iterations per optimization MaxIters
Output:
  Global placement
Algorithm:
  01. Construct a hierarchy of clusters
  02. For (each cluster level $l$ from top to down)
  03.     Set initial placement $\{ClusterPosition_l(i)\}$
  04.     $GridSize \propto 1/sqrt(N_l)$
  05.     $\alpha = 0.5 \cdot GridSize$
  06.     $\varepsilon = 0.1 \cdot GridSize$
  07.     $\mu = 0.5 \frac{TotalAbsoluteDensityGradient}{TotalAbsoluteWirelengthGradient}$
  08.     While ( $Discrepancy > TargetDisc$)
  09.       While ( $\#Iter < MaxIters$)
  10.         $f = WL + \frac{1}{2\mu} \cdot QuadraticPenalty$
  11.         Compute conjugate gradients $Gradient_l$
  12.         $StepLength = LineSearch(f, Gradient_l)$
  13.         $ClusterPosition_l += StepLength * Gradient_l$
  14.         If ($StepLength < \varepsilon$)
  15.           $\mu = 0.5\mu$
  16.           break
  17. Return module placement $\{ClusterPosition_0(i)\}$
```

Figure 7: Top-Down Multi-Level Algorithm

lem. Note that when $\alpha$ is small, the wirelength approximation in Section 3.1.1 is close to the HPWL. Thus for flat placement, we use the actual HPWL instead of the approximation in the line search algorithm, in order to reduce runtime.

**Definition 1** Discrepancy within area $A_w$ *is defined as the maximum ratio of total module area to the window area over all windows with area* $A_w$.

Our placer uses *discrepancy* within 1% of the total placement area to measure evenness of module distribution. The global placer stops when the discrepancy is less than a user specified target value, which is 1.0 at default.

## 4 Legalization and Detailed Placement

Our legalization scheme is based on the schemes of [18, 27] with few modifications. In the basic scheme, cells are first sorted according to their horizontal locations, and then they are processed in order from left to right, where each cell is assigned to the closest available position. We then repeat the above procedure except that cells are processed in the reverse order from right to left this time. We pick the better of the two legalization results.

Detailed placement is composed of three phases that can be iterated a number of times until a negligible threshold of improvement is attained. The three stages are (i) global cell moving, (ii) whitespace distribution, and (iii) cell order polishing. We start by describing global cell moving.

### 4.1 Global Moving

The objective of global moving is to move each cell to the optimal location among available whitespace without changing other modules' positions. Global moving is applied in our placer to improve placement quality for designs with a low utilization ratio. However, for designs without plenty of whitespace, since the global placer is already quite strong, the effect of global moving could be negligible.

We design an efficient heuristic to find a suboptimal available location for each cell. For each cell, we first traverse all the nets connected to the cell, and decide the optimal region for the cell's placement based on the median idea of [17]. Then we search for an available location in the optimal region, if the current placement is not already in it.

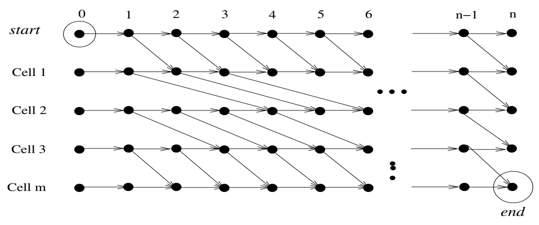

Figure 8: The directed acyclic graph $G$ for finding the optimal whitespace distribution.

If the optimal region is already filled up, we divide the placement area into uniform bins, choose a "best" bin according to available whitespace in the bin and the cost (wirelength difference) of moving the module to the center of the bin, and then search for a best available location in the candidate bin. To quickly estimate if it is possible for a bin to have a continuous whitespace wider than the cell, we assume a normal distribution of whitespace with respect to its width, and obtain the average $\mu$ and standard deviation $\sigma$ at the beginning of global moving. Therefore the number of whitespaces in a bin with total whitespace $s$ that can hold a cell with width $w$ is $s \cdot \frac{1}{2} \cdot erfc(\frac{w-u}{\sqrt{2}\sigma})$.

### 4.2 Whitespace Distribution

The objective of whitespace distribution is to optimally place the whitespace within each row to minimize the wirelength while reserving the cell order [24, 21]. We briefly sketch our procedure[2]. We define a *subrow* $S_i$ as sequence of ordered sites $S_i = \{s_1, s_2, \ldots, s_n\}$ starting from a left fixed boundary - layout periphery or a fixed object - and ending at a right fixed boundary. Let $C_i = \{c_1, c_2, \ldots, c_m\}$ denote the set of $m$ cells residing at subrow $S_i$, $x(.)$ indicates the leftmost site occupied by a cell and $w(\cdot)$ the width of a cell. To optimally redistribute the whitespace in subrow $S_i$, we construct a directed acyclic graph $G = (V, E)$ as shown in Figure 8 with vertex set $V = \{0, \ldots, n\} \times \{0, \ldots, m\}$, and edge set

$$E = \{(j, k-1) \rightarrow (j, k) \mid 0 \leq j \leq m, 1 \leq k \leq n\} \cup \{(j-1, k) \rightarrow (j, k + w(c_j)) \mid 0 \leq j \leq m, 1 \leq k \leq n - w(c_j)\}.$$

The set of edges $E$ is composed of the union of horizontal edges and diagonal edges in $G$. A diagonal edge indicates the placement of a cell at its tail, while a horizontal edge indicates no placement. Thus to minimize HPWL, each diagonal edge starting at $(j-1, k)$ is labelled by the cost (wirelength difference) of placing a cell $c_j$ in position $k$, and all horizontal edges are labelled by zero. Finding an arrangement of the cells that optimally distributes whitespace corresponds to calculating the shortest path in $G$ from the *start* node to the *end* node. Since $G$ is a directed acyclic graph, the shortest path can be calculated using topological traversal of $G$ in $O(mn)$ steps.

A dynamic programming algorithm is applied to find the shortest path from the *start* node to the *end* node in $G$ as shown in Figure 8. The algorithm uses a table of size $mn$, computes

[2]While it is possible to use faster methods such as the CLUMPING algorithm [24], we use the method described since it is more convenient with cell order polishing described in next subsection.

the shortest distance from the *start* node to the node $(j,k)$ row by row from left to right and marks for each node whether the shortest path comes from the left node $(j,k-1)$ in the same row or the node $(j-1,k-w(c_j))$ in the upper row. In the table, the element at $(j,k)$ is the minimum cost (total HPWL difference) of placing the first $j$ cells in the first $k$ sites. After the calculation is finished for all the nodes, the shortest path can be traced from the *end* node back to the *start* node, and the optimal placement is obtained.

We also speed up the algorithm by constantly comparing the total size of non-placed cells to available whitespace. Suppose the algorithm is currently computing the element $(j,k)$. Let $t$ be the total size of the remaining $m-j$ cells. If $t < n-k$, i.e., there is no enough whitespace left for placing the rest of the cells, the dynamic programming algorithm stops calculating the remaining elements for the $j^{th}$ row of the table, meaning that the remaining $n-k$ sites cannot be occupied by the first $j$ cells in a valid placement.

### 4.3 Cell Order Polishing

The idea of cell order polishing is to permute a small window of cells in order to improve wirelength. Similar techniques are commonly applied in academic placers. For example, Capo applies a detailed placement improvement technique based on the optimal placement [9], RowIroning, which permutes several cells in one row assuming equal whitespace distribution between cells. FengShui's cell ordering technique [2] permutes six objects in one or more rows regarding whitespace as pseudo cells.

In this section, we present a branch-and-bound algorithm that permutes the order of a few nearby cells in one row or multiple rows, and simultaneously considers the optimal placement for each permutation in a small window. Thus, our algorithm allows more accurate, overlap-free permutations and does not have to shift other cells.

#### 4.3.1 Ordering for Intra-Row Cells

For a few consecutively placed cells in one row, we define a small window of available sites, which includes all the sites occupied by the cells and neighboring whitespace. The algorithm permutes the cells, finds the optimal placement for each permutation within the window using the dynamic programming algorithm described in Section 4.2, and selects the best permutation and the correspondent optimal placement as the solution.

An important fact is that the cost of placing the first $j$ cells of a permutation is not related to the order of the rest of the cells, because they will be definitely placed to the right of the first $j$ cells. E.g., given a net incidental to the $j^{th}$ cell, it is clear whether the cell is the rightmost or leftmost terminal of the net and the cost in $x$-wirelength can be accurately calculated.

Therefore, the dynamic programming algorithm can be easily combined with permutation of cells to speed up the process. We construct the permutations of cells in lexicographic order, so that the next permutation has a same prefix as the previous one, and thus the beginning rows in the table calculated by the dynamic programming algorithm for the previous permutation can be reused for the next one as possible. When constructing a new permutation that has the same prefix of $j-1$ cells as the previous one and selecting a different $j^{th}$ cell, we keep the first $j-1$ rows of the table in the dynamic programming algorithm and recompute the $j^{th}$ row for the costs of placing the new cell.

Note that in the dynamic programming algorithm, after we select the first $j$ cells for a permutation, we already know the minimum cost of placing the first $j$ cells within the window. Therefore, if the best placement for the first $j$ cells of the permutation is already worse than the current best solution of the cell order polishing problem, we will discard all the permutations that have the same prefix of length $j$. This can be easily implemented in our algorithm.

#### 4.3.2 Ordering for Inter-Row Cells

Similarly, we can design a dynamic programming algorithm for permutating a few nearby cells in multiple rows. Here, the window of available sites will include neighbor sites among multiple rows. The algorithm first decides how many cells will be assigned to each row from up to down, and then finds the optimal placement within the window for each permutation of cells, which also tells which row that each cell is assigned to.

We use the same method as in Section 4.3.1 to combine the dynamic programming algorithm with the lexicographic way of constructing permutations. Note that when we construct a new permutation that has the same prefix of $j-1$ cells as the previous one by selecting a $j^{th}$ cell, and compute the $j^{th}$ row of the table, we only know that the rest of the cells will be placed in the same row as the $j^{th}$ cell or in lower rows. However, whether the remaining cells will be placed to the left or right of the $j^{th}$ cell is not clear. Therefore, the cost in $y$-wirelength can be accurately calculated, and if there is a net connecting to the $j^{th}$ cell and another non-placed cell, the cost in $x$-wirelength for the net is inaccurate. Because of the exponential time complexity of the algorithm, in practice, cell order polishing can only be applied to a small subset of cells at one time, and empirical studies show that the method is still very effective.

## 5 Experimental Results

In this section we report the performance of our placer on three different benchmark sets: IBM ISPD'04, ICCAD'04, and the new ISPD'05 set. These benchmarks feature different characteristics, all of which are helpful in testing the placer's capabilities: IBM ISPD'04 benchmarks [35] are composed of standard cells and test basic placer performance without worry about other "extras" such as whitespace distribution, movable blocks and fixed blocks. IBM ICCAD'04 benchmarks [1] contain large movable macros and assess the performance of a placer in simultaneous floorplanning and standard cell placement. IBM ISPD'05 benchmarks [28] contain large amount of whitespace, and fixed blocks and I/Os, as well as designs with over two million components. These benchmarks are directly derived from industrial ASIC designs, and preserve the physical structure of the designs, unlike other benchmark suites. Thus, they test a placer's ability to handle modern layout features such as whitespace and fixed blocks, and represent the current and future physical design challenges.

In all of our experiments, we use a Linux machine with 1.6GHz CPU and 2GB of memory. In the first set of experiments, we report our results for the ISPD'05 benchmarks in Table 1. We report the results of other placers as published in the contest results [28]. The first column of the table shows the nine placers participating in the ISPD-2005 placement contest [28]. The HPWL for each of the six benchmarks obtained by

| Placer | Benchmark | | | | | | Average |
|---|---|---|---|---|---|---|---|
| | adaptec2 | adaptec4 | bigblue1 | bigblue2 | bigblue3 | bigblue4 | |
| Ours | 87.31 | 187.65 | 94.64 | 143.82 | 357.89 | 833.21 | 1.00 |
| mFAR [20] | 91.53 | 190.84 | 97.70 | 168.70 | 379.95 | 876.28 | 1.06 |
| Dragon [34] | 94.72 | 200.88 | 102.39 | 159.71 | 380.45 | 903.96 | 1.08 |
| mPL [12] | 97.11 | 200.94 | 98.31 | 173.22 | 369.66 | 904.19 | 1.09 |
| FastPlace [36] | 107.86 | 204.48 | 101.56 | 169.89 | 458.49 | 889.87 | 1.16 |
| Capo [31] | 99.71 | 211.25 | 108.21 | 172.30 | 382.63 | 1098.76 | 1.17 |
| NTUP [14] | 100.31 | 206.45 | 106.54 | 190.66 | 411.81 | 1154.15 | 1.21 |
| FengShui [3] | 122.99 | 337.22 | 114.57 | 285.43 | 471.15 | 1040.05 | 1.50 |
| Kraftwerk+Domino [30] | 157.65 | 352.01 | 149.44 | 322.22 | 656.19 | 1403.79 | 1.84 |

Table 1: Results of all placers on the ISPD-2005 contest benchmarks. The results of other placers are from [28].

the placers are shown in the next six columns[3]. We normalize each wirelength result based on the HPWL obtained by our placer. The last column in Table 1 shows the average normalized ratio for each placer. Our placer gives the best results on all six benchmarks and on average is better than the best of all other placers by 6%. The entire benchmark set takes 113.2 hours of runtime to complete. Executing Capo v9.1 [1] on our machines takes around 37.8 hours to complete. Thus, on the average, our placer is 3× slower than Capo.

In the second set of experiments, we evaluate the performance of our placer on the IBM ICCAD'04 mixed-size benchmarks [1]. These recent mixed-size circuits contain large movable blocks with non-ignorable aspect ratios and I/Os placed at the blocks' periphery, and thus are more realistic than the previously widely used IBM ISPD'02 mixed-size benchmarks. Our results are summarized in Table 2 and compared to FengShui and Capo. The first five columns of Table 2 show the HPWL after global placement, runtime of global placement, HPWL after legalization, HPWL after detailed placement and runtime of detailed placement, respectively, for our placer. According to the table, the legalization and detail placement steps reduce the wirelength by 4% on average, which indicates a strong global placement and effective post-processing. We report the results of FengShui v2.6 and Capo v9.0 as recently published in [1] in the last two columns. The last row in Table 2 shows the average normalized wirelength ratio based on FengShui's results. The results show that our placements are better than FengShui and Capo for all the circuits, and the average improvement is around 14% over FengShui and 19% over Capo. We also believe it is possible to further improve our results if cell flipping is applied - an improvement executed by Capo. The runtime of the entire benchmark set takes 18.0h of runtime. The total runtime of FengShui and Capo on a Linux machine with 2.4GHz CPU are 8.7h and 14.0h respectively, as reported in [1].

The focus of our third set of experiments is on the IBM ISPD'04 standard cell benchmark suite. Wirelength after global placement, wirelength after detail placement, and total runtime of our placer are shown in the second to fourth columns of Table 3. We also report the latest wirelength results of mPL v5.0 (in the fifth column), as well as the normalized wirelength ratio with respect to mPL's results of Capo v9.0, Dragon v3.01, FastPlace v1.0 and FengShui v5.0 (in the sixth to ninth columns respectively), as published in [13]. The last row in Table 3 shows the average normalized ratio with respect to mPL's results for each placer. The results show that our placements are better than other placers for most of the circuits. On average, our placer is better than the best of all other placers, mPL, by 3%, and better than Capo, Dragon, FastPlace and FengShui by 11%, 6%, 10% and 8% respectively. The entire benchmark set takes 17.4h to place. The total runtime of mPL5, Capo, Dragon, FastPlace and FengShui on a Linux machine with 2.4GHz CPU are 3.2h, 7.2h, 39.2h, 0.6h and 6.4h respectively, as reported in [13].

The excellent performance of our placer on all benchmark sets clearly show that our placement methods are: (1) scalable, (2) deliver high quality placements, and (3) capable of handling various netlist and layout features such as movable/fixed blocks and whitespace.

[3]Due to the contest setup, the results are obtained after five days of tuning of the placers with the circuits.

| bench | Placer | | | | | | |
|---|---|---|---|---|---|---|---|
| | Ours | | | | | FS | Capo |
| | gpWL (e6) | CPU (s) | legWL (e6) | dpWL (e6) | CPU (s) | dpWL (e6) | dpWL (e6) |
| ibm01 | 2.17 | 436 | 2.20 | 2.14 | 28 | 2.56 | 2.67 |
| ibm02 | 4.83 | 949 | 4.73 | 4.61 | 57 | 6.05 | 5.54 |
| ibm03 | 6.94 | 1078 | 6.93 | 6.72 | 67 | 8.77 | 8.67 |
| ibm04 | 7.70 | 1169 | 7.83 | 7.60 | 73 | 8.38 | 9.79 |
| ibm05 | 9.82 | 915 | 9.90 | 9.70 | 61 | 9.94 | 10.82 |
| ibm06 | 6.31 | 988 | 6.17 | 5.99 | 87 | 6.99 | 7.35 |
| ibm07 | 10.04 | 1445 | 10.35 | 10.02 | 124 | 11.37 | 11.23 |
| ibm08 | 12.65 | 1328 | 12.64 | 12.34 | 149 | 13.51 | 16.02 |
| ibm09 | 12.56 | 2515 | 12.63 | 12.15 | 170 | 14.12 | 15.51 |
| ibm10 | 30.32 | 3518 | 29.82 | 28.55 | 354 | 41.96 | 34.98 |
| ibm11 | 19.62 | 4253 | 19.41 | 18.67 | 236 | 21.19 | 22.31 |
| ibm12 | 34.51 | 3598 | 34.56 | 33.51 | 314 | 40.84 | 40.78 |
| ibm13 | 24.28 | 4869 | 24.07 | 23.03 | 308 | 25.45 | 28.70 |
| ibm14 | 37.51 | 4878 | 36.87 | 35.90 | 479 | 39.93 | 40.97 |
| ibm15 | 49.97 | 5337 | 48.93 | 46.82 | 708 | 51.96 | 59.19 |
| ibm16 | 57.15 | 6244 | 57.02 | 54.58 | 905 | 62.77 | 67.00 |
| ibm17 | 67.39 | 6495 | 69.01 | 66.49 | 834 | 69.38 | 78.78 |
| ibm18 | 44.40 | 9159 | 43.11 | 42.14 | 797 | 45.59 | 50.39 |
| Average | | | | 0.86 | | 1.00 | 1.05 |

Table 2: Results on the IBM ICCAD'04 mixed-size benchmarks. Results of Capo and FengShui are from [1].

| bench | Placer | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Ours | | | mPL5 | Capo | Dragon | FP | FS |
| | gpWL (e6) | dpWL (e6) | CPU (s) | dpWL (e6) | nWL | nWL | nWL | nWL |
| ibm01 | 1.60 | 1.63 | 333 | 1.67 | 1.08 | 1.02 | 1.09 | 1.08 |
| ibm02 | 3.54 | 3.48 | 649 | 3.62 | 1.09 | 1.02 | 1.06 | 1.02 |
| ibm03 | 4.46 | 4.51 | 874 | 4.57 | 1.10 | 1.05 | 1.12 | 1.03 |
| ibm04 | 5.56 | 5.61 | 996 | 5.75 | 1.06 | 1.00 | 1.04 | 1.05 |
| ibm05 | 9.63 | 9.49 | 1245 | 9.92 | 1.02 | 0.98 | 1.05 | 1.00 |
| ibm06 | 4.73 | 4.78 | 951 | 5.10 | 1.11 | 0.98 | 1.04 | 1.02 |
| ibm07 | 7.97 | 7.90 | 1892 | 8.23 | 1.11 | 1.04 | 1.08 | 1.09 |
| ibm08 | 9.16 | 9.46 | 1296 | 9.38 | 1.05 | 0.96 | 1.02 | - |
| ibm09 | 8.84 | 8.93 | 2104 | 9.33 | 1.08 | 1.07 | 1.12 | 1.06 |
| ibm10 | 17.20 | 16.95 | 3089 | 17.3 | 1.10 | 1.04 | 1.07 | 1.07 |
| ibm11 | 13.22 | 13.38 | 2936 | 14.0 | 1.09 | 1.03 | 1.09 | 1.04 |
| ibm12 | 21.83 | 21.47 | 3124 | 22.3 | 1.11 | 1.03 | 1.08 | 1.07 |
| ibm13 | 16.46 | 16.60 | 3702 | 16.6 | 1.10 | 1.05 | 1.11 | 1.09 |
| ibm14 | 30.55 | 30.76 | 4648 | 31.6 | 1.10 | 1.05 | 1.11 | 1.04 |
| ibm15 | 38.38 | 38.81 | 7364 | 38.5 | 1.09 | 1.04 | 1.13 | 1.07 |
| ibm16 | 41.36 | 41.32 | 7181 | 43.0 | 1.10 | 1.05 | 1.07 | 1.09 |
| ibm17 | 60.82 | 59.22 | 10261 | 61.3 | 1.09 | 1.08 | 1.08 | 1.08 |
| ibm18 | 39.32 | 38.98 | 10127 | 41.0 | 1.09 | 1.02 | 1.10 | 1.04 |
| Average | | 0.97 | | 1.00 | 1.09 | 1.03 | 1.08 | 1.06 |

Table 3: Results on the IBM ISPD'04 benchmarks. Results of Capo, Dragon, FastPlace, FengShui and mPL are from [13].

## 6 Summary and Conclusions

In this paper we have presented the architecture and details of the newest version of our placer. The major algorithmic changes with respect to our earlier published placement flow are: (1) a new clustering algorithm, (2) a dynamic scalable analytical engine, and (3) improved detailed placement algorithms. The experimental results clearly show the superiority of our placer in comparison to other academic placers on three recent benchmark suites. On average, our placements are better than the best published results by 3%, 14%, and 6% for the IBM ISPD'04, ICCAD'04, and ISPD'05 benchmark sets respectively. Reduction in placed HPWL reduces routing congestion and we believe that further improvement in routed wirelength is expected.

Our future work includes improving our placer runtime, handling cell flipping, and better and faster mechanisms for handling fixed objects.

## References

[1] S. N. Adya, S. Chaturvedi, J. A. Roy, D. A. Papa and I. L. Markov, "Unification of Partitioning, Placement and Floorplanning," in *Proc. IEEE International Conference on Computer Aided Design*, 2004, pp. 550–557.

[2] A. R. Agnihotri, S. Ono, C. Li, M. C. Yildiz, A. Khatkhate, C.-K. Koh and P. H. Madden, "Mixed Block Placement via Fractional Cut Recursive Bisection," *IEEE Transactions on Computer-Aided Design*, to appear.

[3] A. Agnihotri, S. Ono and P. Madden, "Recursive Bisection Placement: Feng Shui 5.0 Implementation Details," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 230–232.

[4] A. Agnihotri, M. Yildiz, A. Khatkhate, A. Mathur, S. Ono and P. Madden, "Fractional Cut: Improved Recursive Bisection Placement," in *Proc. IEEE International Conference on Computer Aided Design*, 2003, pp. 307–310.

[5] C. Alpert, A. Kahng, G.-J. Nam, S. Reda and P. Villarrubia, "A Semi-Persistent Clustering Technique for VLSI Circuit Placement," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 200–207.

[6] C. J. Alpert, J. H. Huang and A. B. Kahng, "Multilevel Circuit Partitioning," in *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 530–533.

[7] U. Brenner, A. Pauli and J. Vygen, "Almost Optimum Placement Legalization by Minimum Cost Flow and Dynamic Programming," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2004, pp. 2–9.

[8] U. Brenner and J. Vygen, "Faster Optimal Single-Row Placement with Fixed Ordering," in *Proc. Design, Automation and Test in Europe*, 2000, pp. 117–121.

[9] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Optimal Partitioners and End-Case Placers for Standard-Cell Layout," in *Proc. Proc. ACM/IEEE International Symposium on Physical Design*, 1999, pp. 90–96.

[10] ——, "Can Recursive Bisection Alone Produce Routable Placements?" in *Proc. ACM/IEEE Design Automation Conference*, 2000, pp. 477–482.

[11] T. F. Chan, J. Cong, T. Kong and J. R. Shinnerl, "Multilevel Optimization for Large-Scale Circuit Placement," in *Proc. IEEE International Conference on Computer Aided Design*, 2000, pp. 171–176.

[12] T. F. Chan, J. Cong, M. Romesis, J. R. Shinnerl, K. Sze and M. Xie, "mPL6: A Robust Multilevel Mixed-Size Placement Engine," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 227–229.

[13] T. F. Chan, J. Cong and K. Sze, "Multilevel Generalized Force-directed Method for Circuit Placement," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 185–192.

[14] T.-C. Chen, T.-C. Hsu, Z.-W. Jiang and Y.-W. Chang, "NTUplace: A Ratio Partitioning Based Placement Algorithm for Large-Scale Mixed-Size Designs," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 236–238.

[15] K. Doll, F. Johannes and K. Antreich, "Iterative Placement Improvement by Network Flow Methods," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13(10), pp. 1189–1200, 1994.

[16] H. Eisenmann and F. M. Johannes, "Generic Global Placement and Floorplanning," in *Proc. ACM/IEEE Design Automation Conference*, 1998, pp. 269–274.

[17] S. Goto, "An Efficient Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Layout," *IEEE Transactions on Circuits and Systems*, vol. 28(1), pp. 12–18, 1981.

[18] D. Hill, "Method and System for High Speed Detailed Placement of Cells Within an Integrated Circuit Design," *US Patent 6370673*, 2001.

[19] B. Hu and M. Marek-Sadowska, "Fine Granularity Clustering-Based Placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23(4), pp. 527–536, 2004.

[20] B. Hu, Y. Zeng and M. Marek-Sadowska, "mFAR: Fixed-Points-Addtion-Based VLSI Placement Algorithm," *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 239–241.

[21] A. B. Kahng, I. Markov and S. Reda, "On Legalization of Row-Based Placements," in *Proc. IEEE Great Lakes Symposium on VLSI*, 2004, pp. 214–219.

[22] A. B. Kahng and Q. Wang, "Implementation and Extensibility of an Analytic Placer," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2004, pp. 18-25.

[23] ——, "An Analytic Placer for Mixed-Size Placement and Timing-Driven Placement," in *Proc. IEEE International Conference on Computer Aided Design*, 2004, pp. 565-572.

[24] A. B. Kahng, P. Tucker and A. Zelikovsky, "Optimization of Linear Placements for Wirelength Minimization with Free Sites," in *Proc. IEEE Asia and South Pacific Design Automation Conference*, 1999, pp. 241–244.

[25] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar, "Multilevel hypergraph partitioning: Application in VLSI domain," in *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 526–529.

[26] G. Karypis and V. Kumar, "Multilevel *k*-way hypergraph partitioning," in *Proc. ACM/IEEE Design Automation Conference*, 1999, pp. 343–348.

[27] A. Khatkhate, C. Li, A. R. Agnihotri, M. C. Yildiz, S. Ono, C.-K. Koh and P. H. Madden, "Recursive Bisection Based Mixed Block Placement," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2004, pp. 84–89.

[28] G.-J. Nam, C. Alpert, P. Villarrubia, B. Winter and M. Yildiz, "The ISPD2005 Placement Contest and Benchmark Suite," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 216–219.

[29] W. Naylor, "Non-Linear Optimization System and Method for Wire Length and Delay Optimization for an Automatic Electric Circuit Placer," *US Patent 6301693*, 2001.

[30] B. Obermeier, H. Ranke and F. M. Johannes, "Kraftwerk - A Versatile Placement Approach," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 242–244.

[31] J. A. Roy, D. A. Papa, S. N. Adya, H. H. Chan A. N. Ng, J. F. Lu and I. L. Markov, "Capo: Robust and Scalable Open-Source Min-Cut Floorplacer," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 224–226.

[32] C. Sechen and K. W. Lee, "An Improved Simulated Annealing Algorithm for Row-Based Placement," in *Proc. IEEE International Conference on Computer Aided Design*, 1987, pp. 478–481.

[33] W.-J. Sun and C. Sechen, "Efficient and Effective Placement for Very Large Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14(5), pp. 349–359, 1995.

[34] T. Taghavi, X. Yang, B. K. Choi, M. Wang and M. Sarrafzadeh, "DRAGON2005: Large-Scale Mixed-Size Placement Tool," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2001, pp. 245–247.

[35] N. Viswanathan and C. Chu, "FastPlace: Efficient Analytical Placement Using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2004, pp. 26–33.

[36] ——, "FastPlace: An Analytical Placer for Mixed-Mode Designs," in *Proc. ACM/IEEE International Symposium on Physical Design*, 2005, pp. 221–223.

[37] J. Vygen, "Algorithms for Large-Scale Flat Placement," in *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 746–751.

[38] ——, "Algorithms for Detailed Placement of Standard Cells," in *Design, Automation and Test in Europe*, 1998, pp. 321–324.

[39] M. Wang, X. Yang and M. Sarrafzadeh, "DRAGON2000: Standard-Cell Placement Tool for Large Industry Circuits," in *Proc. IEEE International Conference on Computer Aided Design*, 2001, pp. 260–263.