Optimization of Linear Placements for Wirelength Minimization with Free Sites*

Andrew B. Kahng, Paul Tucker† and Alex Zelikovsky

UCLA CS Department, Los Angeles, CA 90095-1596
†UCSD CSE Department, La Jolla, CA, 92093-0114

Abstract

We study a type of linear placement problem arising in detailed placement optimization of a given cell row in the presence of white-space (extra sites). In this *single-row placement problem*, the cell order is fixed within the row; all cells in other rows are also fixed. We give the first solutions to the single-row problem: (i) a dynamic programming technique with time complexity $O(m^2)$ where m is the number of nets incident to cells in the given row, and (ii) an $O(m \log m)$ technique that exploits the convexity of the wirelength objective. We also propose an iterative heuristic for improving cell ordering within a row; this can be run optionally before applying either (i) or (ii). Experimental results show an average of 6.5% wirelength improvement on industry test cases when our methods are applied to the final output of a leading industry placement tool.

1 Introduction

The *linear placement problem* (LPP) is well-studied in the VLSI physical design literature, where it has appeared in various guises [15, 10, 11, 6, 2, 9, 13]. Traditionally, linear placement seeks to arrange a number of interconnected circuit modules within a row of locations, to minimize some objective. Applications of LPP are reported in, e.g., [10, 4, 5]. The LPP problem is NP-hard [8], and so most existing methods are heuristics (see [12] for a comprehensive review).

In this work, we address a variant of the linear placement problem that arises during detailed placement optimization of standard-cell layouts. Our context is a "successive-approximation" placement methodology, e.g., (1) mixed analytic (quadratic) and partitioning-based top-down global placement (cf. most leading EDA vendor tools), (2) row balancing and legalization, (3) detailed optimization of row assignments (cf., for example, the TimberWolf placement tool [14]), and (4) detailed routability optimization ("white-space" management and pin alignment) within individual rows. Step (4) in this methodology arises because of routing hot-spots and limited porosity of the cell library: even with deep-submicron multilayer interconnect processes, placers must leave extra space within cell rows so that the layout is routable.

Our problem differs from the traditional LPP formulation in that (i) cells of a given row can be placed with gaps between them, i.e., the number of legal locations is more than the number of cells, and the layout has "white-space"; and (ii) fixed cells from other rows participate in the net wirelength (cost) estimation for a given row.

Our contributions include the following.

- We present the first *optimal* algorithms for single-row cell placement with free sites (white-space), where the order of the cells in the row is fixed, and positions of cells in all other rows are fixed. Our algorithms are based on dynamic programming, and on exploiting convexity of the objective.
- We present a new iterative algorithm to improve the cell ordering within a given row.
- We present an iterative row-based placement algorithm that applies single-row cell placement to each row in turn, with optional cell ordering improvement in the given row.
- We achieve an average of 6.5% improvement in the wirelength objective[1] on five industry test cases, starting from the final output (i.e., after global and detailed placement) of an industry placement tool.

The remainder of this paper is organized as follows. In Section 2, we develop notation and terminology. Section 3 analyzes the *cell cost function*, which captures the dependency of the wirelength objective on the position of a movable cell in the given row. Section 4 then proposes a dynamic programming approach. We improve the dynamic programming approach in Section 5 using the piecewise linearity of the cost dependence on cell position, and the monotonicity of a so-called *prefix cost function* derived from the cell cost function. In Section 6, we develop another, more efficient solution that exploits the convexity of the wirelength objective. Section 7 describes our heuristic for improving the order of cells within a row, and Section 8 concludes with experimental results and directions for future work.

2 Notation and Problem Statement

The Single-Row Problem. We are given a single cell row with n *movable* cells C_i, $i = 1, \ldots, n$ whose left-to-right order is fixed, but whose positions are variable. All cells in all other rows are *fixed cells*, i.e., they have fixed positions. We seek a non-overlapping placement of movable cells such that the total bounding-box half-perimeter of all m signal nets N_j, $j = 1, \ldots, m$ is minimized. This is a one-dimensional problem: we minimize the total x-span of all nets. All cells have integer widths, and must be placed on an integer lattice of *sites*.[2]

We use the following notation (see Figure 1).

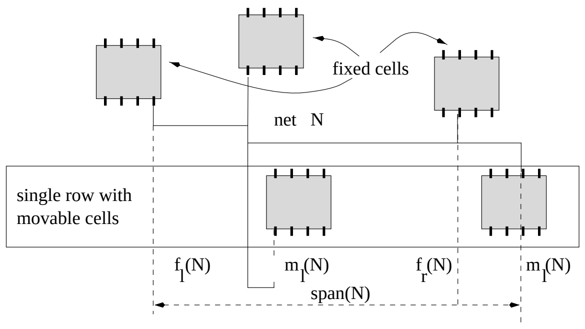

Figure 1: A net N with fixed and movable cells.

- The *position* of a cell C_i is the x-coordinate of the left edge of the cell. In general, all positions are x-coordinates.
- The entire row with movable cells consists of a sequence of k legal positions of cells ("sites") $s_1, \ldots, s_k$. A given cell C_i occupies w_i consecutive sites when placed.
- The *range* S_i of a cell C_i denotes the set of sites at which C_i can be placed. We use $k_i = |S_i|$ to denote the number of such sites.

* Research supported by a grant from Cadence Design Systems, Inc.

[1] We use the traditional objective: sum of net bounding box half-perimeters. Since row assignments are assumed fixed at this stage of the placement process, the objective becomes total horizontal span of all nets.

[2] It is possible to extend our approaches to address *flipping* (mirroring about the y-axis) of cells, but we will not discuss this here.

- The position of the leftmost (respectively, rightmost) *fixed pin* of net N is denoted by $f_l(N)$ (respectively, $f_r(N)$).
- The leftmost (respectively, rightmost) movable cell on N is denoted by $L(N)$ (respectively, $R(N)$); these are known because the order of movable cells is fixed.
- The leftmost (respectively, rightmost) pin of net N on cell C_i is denoted by $l_i(N)$ (respectively, $r_i(N)$).
- The position of the leftmost (respectively, rightmost) *movable pin* of net N is denoted by $m_l(N)$ (respectively, $m_r(N)$).
- The *span* of a net N, denoted $span(N)$, is

$$span(N) = \max\{f_r(N), m_r(N)\} - \min\{f_l(N), m_l(N)\}$$

- The *fixed span* of a net N, denoted $fix\_span(N)$, is

$$fix\_span(N) = f_r(N) - f_l(N)$$

- A *placement* is a subsequence $P = \{p_1, \ldots, p_n\}$ of the sequence of sites, where p_i is the placed position of the cell C_i. Nonoverlapping placement implies $p_i + w_i \leq p_{i+1}$, $i = 1, \ldots, n-1$.
- A *prefix placement*, denoted P_i, is a placement of the first i cells $C_1, \ldots, C_i$.

3 Cost Contribution of a Movable Cell

The *cell cost function*, denoted $cost_i(x)$, of a movable cell C_i is the sum over all nets N of the contribution of cell C_i to $span(N) - fix\_span(N)$. In other words, for a given position x of the cell C_i,

$$\begin{aligned} cost_i(x) &= \sum_{C_i = R(N)} \max\{m_r(N) - f_r(N), 0\} \\ &+ \sum_{C_i = L(N)} \max\{f_l(N) - m_l(N), 0\} \end{aligned}$$

The cost of a prefix placement P_i is given by $cost(P_i) = \sum_{j=1}^{i} cost_j(p_j)$, i.e., the sum of the cell costs when each cell in the prefix is placed by P_i.

The cell cost function $cost_i(x)$ is seen to be piecewise linear and convex, as follows. Create a sorted list containing all $f_r(N)$ positions for nets with $C_i = R(N)$ and all $f_l(N)$ positions for nets with $C_i = L(N)$. Between any two consecutive positions in this sorted list, $cost_i(x)$ is linear and decreasing (respectively, constant or increasing) if the number of $f_r(N)$ positions that are to the left of their corresponding $l_i(N)$'s is smaller than (respectively, equal to or greater than) the number of $f_l(N)$ positions that are to the right of their corresponding $r_i(N)$'s (see Figure 2). This means that the piecewise linear function $cost_i(x)$ is convex, with local minimum either a point or a segment.[3] The number of linear pieces in the function $cost_i(x)$ will be denoted $Bound_i$. Note that $Bound_i$ is at most twice the number of pins on C_i since C_i can be both $L(N)$ and $R(N)$ for the same net N. The total number of pins on the movable cells that can be the leftmost or rightmost for some net will be denoted

$$B = \sum_{i=1}^{n} Bound_i \leq 2m, \tag{1}$$

where m is the number of signal nets (we should take in account only nets containing movable cells), because each net contains exactly one leftmost and one rightmost pin.

Observe that for any given placement P

$$\sum_{i=1}^{n} cost_i(p_i) = \sum_{j=1}^{m} span(N_j) - \sum_{j=1}^{m} fix\_span(N_j)$$

[3]This description remains valid when there are different pin positions on a given cell.

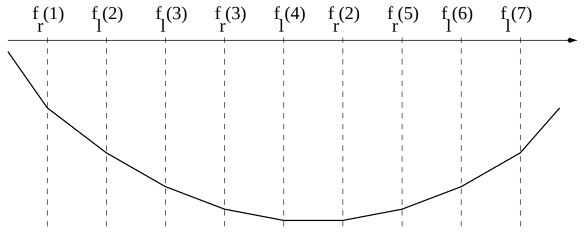

Figure 2: The cell cost function for a multi-pin cell.

4 Dynamic Programming Algorithm

We propose the following dynamic programming solution for the Single-Row Problem. Let $C_1, \ldots, C_n$ be the fixed left-to-right order of the movable cells. For each legal position s_j of each movable cell C_i, we will compute the minimum-cost *constrained* prefix placement of the cells $C_1, \ldots, C_i$, subject to the position of C_i being at or to the left of s_j (i.e., $p_i \leq j$). We denote this constrained prefix placement by $P_i(s_j)$.

The optimum constrained prefix placement $P_i(s_j)$ can be computed from the cost function $cost_i()$, prefix placements $P_{i-1}()$ for the preceding cell, and prefix placements $P_i(s_h)$ for $h < j$, as follows. Consider that in the prefix placement $P_i(s_j)$, the position of cell C_i is either strictly to the left of s_j or exactly at s_j. In the latter case, the preceding cell C_{i-1} cannot be placed to the right of the site $s_{j-w_{i-1}}$ because it must not overlap with C_i. Thus, the minimum-cost placement $P_i(s_j)$ is selected from two alternatives: (i) $P_i(s_{j-1})$, and (ii) $P_{i-1}(s_{j-w_{i-1}})$ extended by $p_i = s_j$. Notice that extending the prefix placement by $p_i = s_j$ simply means adding the cell cost $cost_i(s_j)$.

The dynamic programming principle is applied by using the precomputed solutions for $P_{i-1}()$ to compute $P_i()$. The outer loop of the algorithm is over cell indices i, and the inner loop is over corresponding ranges of positions j. In the inner loop we can walk along the values $s_j \in S_i$, i.e., the domain of cell cost function $cost_i()$, so that evaluating (ii) requires only constant time per each s_j. Then, we can choose the better option in $O(1)$ time. The total runtime will be of order

$$\sum_{i=1}^{n} k_i \approx n \cdot (k - \sum_{i=1}^{n} w_i) \tag{2}$$

Typically, $k - \sum_{i=1}^{n} w_i \approx n$, therefore the runtime is $O(n^2)$.

5 Exploiting Piecewise Linearity

The *optimal prefix cost function* gives the minimum possible cost of a prefix placement $P_i = \{p_1, \ldots, p_i\}$, subject to cell C_i being placed at or to the left of position x. We write this as $pcost_i(x) = min_{p_i \leq x} cost(P_i)$. Note that the optimal prefix cost function is monotonically nonincreasing[4] and piecewise linear. Piecewise linearity follows from (i) and (ii) in the dynamic programming description above. In intervals where (i) is preferable to (ii), $pcost_i(x)$ will remain constant. In intervals where (ii) is preferable to (i), the cell cost function $cost_i$ is decreasing and will be added pointwise to $pcost_{i-1}$. More precisely, we have $pcost_i(x) = pcost_{i-1}(x - w_{i-1}) + cost_i(x)$ while this sum decreases. If the slope of $pcost_i()$ reaches 0 at some $s_j \in S_i$, then $pcost_i()$ remains unchanged thereafter.

As stated above, the DP algorithm essentially tests each feasible position of each cell. However, we can exploit the piecewise linearity of the optimal prefix cost function in recursively constructing $pcost_i$ from $pcost_{i-1}$ and $cost_i$, so that the total number of points explicitly examined may be far less than $\sum_{i=1}^{n} k_i$. The key insight is that the functions $pcost_{i-1}$ and $cost_i$ can be represented, e.g., as sequences of

[4]This is because increasing the x argument relaxes the constraints on the prefix placement.

maximal line segments (tuples with minimum coordinate, maximum coordinate, slope and y-intercept). Then, the sites bounding maximal line segments of $pcost_i$ will be a subset of the sites bounding maximal line segments in $pcost_{i-1}$ (+ w_{i-1}) and $cost_i$. The representation of $pcost_i$ is computed by simultaneously walking these two component sequences.

Recall that the number of linear segments in $cost_i$ is at most the number of nets for which a pin on C_i may be the leftmost or rightmost pin, and that we use $Bound_i$ to denote this number. Given that $pcost_{i-1}$ and $cost_i$ are represented as sequences of linear segments that are represented by tuples $\langle a, b, x_{\min}, x_{\max}\rangle$ (indicating that $pcost_{i-1}(s_j) = ax + b$ for $x_{\min} \le x < x_{\max}$ being the x coordinate of s_j), the next function $pcost_i$ can be constructed as follows, in $O(Bound_{i-1} + Bound_i)$ time.

Prefix Algorithm

```
1. Create a sequence of tuples that is a shifted, extended
   copy of $pcost_{i-1}$
1.1.   for $x \in [\min(S_i), \max(S_{i-1}) + w_{i-1}]$,
       $pcost_i(x) = pcost_{i-1}(x - w_{i-1})$
1.2.   for $x \in [\max(S_{i-1}) + w_{i-1}, \max(S_i]$,
       $pcost_i(x) = pcost_{i-1}(\max(S_{i-1}))$
2. Iterate over the tuples in $cost_i$, in order,
   and sum them into the function representation of $pcost_i$.
```

Note that the slope of $pcost_i$ never rises above 0, so the tail of $pcost_i$ may consist of a single long segment where the tails of $pcost_{i-1}$ and $cost_i$ may contain multiple segments. Consequently, the time to compute $pcost_i$ for all of the cells in the row (by the algorithm above) is upper bounded by $\sum_{j=1}^{i} Bound_j$. In practice, we expect a time complexity closer to $O(n)$ because $Bound_i$ is likely to be upper bounded by a small constant.

Basically, computing $pcost_{i+1}$ from $pcost_i$ and $cost_{i+1}$ is proportional to the time to merge sorted sequences of linear segments of length at most $\sum_{j=1}^{i} Bound_j$ and $Bound_i$. We can use merge sort, which is proportional to $\sum_{j=1}^{i+1} Bound_j$. We conclude that the running time of the Prefix Algorithm is $O(B^2) = O(m^2)$.

6 Clumping Algorithm

The nature of the dynamic programming solutions in Sections 4 and 5 does not exploit the convexity of the function $cost_i$. We now describe an approach, called the *Clumping Algorithm*, based on the convexity of $cost_i$. We then show how to speed up this algorithm using advanced data structures such as red-black search trees.

Below we assume that any cell C_i has pins $\{pin_1^i, \ldots, pin_{r_i}^i\}$ sorted from left to right. Let us shift C_i from the site s_j to the right by some small $\varepsilon > 0$. We set $f_j(pin_l^i) = -1$ if pin_l^i is the rightmost pin on some net N and set $f_j(pin_l^i) = 1$ if pin_l^i is the leftmost pin for some net N. In all other cases we set $f_j(pin_l^i) = 0$.

Lemma 1 *The right slope of the function $cost_i$ at the site s_j equals*

$$slope_i(s_j) = \sum_{l=1}^{r_i} f_j(pin_l^i)$$

Proof. Sliding C_i to the right by ε increases $cost_i$ by $(L - R) \cdot \varepsilon$, where L is the number of nets with the rightmost pin on C_i and R is the number of nets with the leftmost pin on C_i. □

The minimum interval $[l(C_i), r(C_i)]$ of a cell C_i is the interval where the slope $slope_i$ equals 0.

The main step of the Clumping Algorithm, which is an application of the convexity of $cost_i$, involves the collapsing of two neighboring cells. By *collapsing* of two neighboring cells C_i and C_{i+1} we mean replacing them with a new cell C_i'. The new cell C_i' has the set of pins $\{pin_1^i, \ldots, pin_{r_i}^i, pin_1^{i+1}, \ldots, pin_{r_{l+1}}^{i+1}\}$ (sorted from left to right) and width $w(C_i) + w(C_{i+1})$.

We now present our Clumping Algorithm based on collapsing of cells. The set of cells is represented as a double list denoted $CELL$. The function $cost_i(x)$ is represented as a sorted list of positions where the function $cost_i(x)$ changes slope, i.e., where a net has its fixed leftmost/rightmost pin just over the corresponding pin of C_i. Each such position is marked with -1 if it is the rightmost pin of a net and $+1$ if it is the leftmost pin of a net. Then $slope_i(x)$ equals the sum of -1's to the left of x and $+1$'s to the right of x.

Clumping Algorithm

```
1. For $i = 1,..,n$ find the list $cost_i$ and two endpoints $[l_i, r_i]$ of
   the minimum interval, i.e., where $cost_i$ is minimum.
2. $x(C_0) \leftarrow 0$, $w(C_0) \leftarrow 0$, $C' \leftarrow C_0$
 // first cell $C_0$ with 0-position and 0-width
3. Going along the list $CELL$ apply $PLACE(C_i)$
 // i.e., for $i = 1..n$
4. Output positions of original cells using positions of collapsed
   cells and widths of the original cells

Procedure $PLACE(C_i)$
  if $p(C_{i-1}) + w(C_{i-1}) > r(C_i)$
    // if $C_{i-1}$, $C_i$ cannot both be placed in their
    // minimum intervals, $p(C_i) =$ position of $C_i$
  then $COLLAPSE(C_{i-1}, C_i)$ and $PLACE(C_{i-1})$
  else place $C_i$ at the position $p(C_i)$ which is either the
         leftmost minimum available position $p(C_{i-1}) + w(C_{i-1})$
         or the leftmost minimum position $l(C_i)$
         // choose the rightmost of them

Procedure $COLLAPSE(C_{i-1}, C_i)$
   Shift positions from the list of $C_i$ by $w(C_{i-1})$
   Merge the list for $C_i$ with the list for $C_{i-1}$
   Find the minimum interval for the merged lists
   Set $w(C_{i-1}) = w(C_{i-1}) + w(C_i)$
   delete cell $C_i$
```

Theorem 1 *The Clumping Algorithm finds the optimal placement.*

Proof. By induction on n, the number of cells. If $n = 1$, the cell C_1 is placed in the minimum interval and the theorem is obviously true. Now assume that the theorem is true for $n - 1$ cells. We will show that the theorem is true for n cells.

If in the operator **if** in the procedure $PLACE(C_i)$ we have the case **else**, then the theorem is true since we optimally place $C_1, \ldots, C_{n-1}$ (by induction) and then optimally place C_n. Otherwise, if there is a conflict between cells C_{n-1} and C_n, then we collapse C_n with C_{n-1}. In this case, there is no gap between cells C_{n-1} and C_n in the optimal placement and we may find the optimal placement for the union of $C_{n-1} \cup C_n$. Then, by induction, the collapsed cell C_{n-1} is placed optimally. □

Now we will show that the runtime of the Clumping Algorithm is $O(B^2) = O(m^2)$. Indeed, collapsing of two cells C_i and C_{i+1} takes time of order $Bound_i + Bound_{i+1}$. In the worst case, we need to collapse each next cell with the previous cell representing all previous original cells. The runtime is

$$\sum_{i=1}^{n} Bound_i = O(B^2) = O(m^2)$$

Unlike the Prefix Algorithm, the Clumping Algorithm can be implemented to run faster using advanced data structures. We show how to speed up the Clumping Algorithm to $O(m \log m)$ using red-black search trees (RBST).[5] We use RBSTs to represent lists corresponding to each function $cost_i(x)$ for a cell C_i, as follows. For each cell C_i, let Set_i be a set of coordinates at which the function $cost_i$ changes slope. We supply each element s of Set_i with the *net weight* $nw(s) = 1$ if $s = f_r(N)$ and $nw(s) = -1$ if $s = f_l(N)$.[6] Each set Set_i is supplied with pointers to the left and right ends of the minimum interval $[l_i = l(C_i), r_i = r(C_i)]$ $(slope_i(l_i) = 0)$.

[5]The red-black search tree is a data structure which allows insertion, deletion and finding of predecessor and successor (i.e. previous or next in the sorted list) in a set of n numbers in time $O(\log n)$ (see, e.g., Chapter 14 in [7]).

[6]Note that if we have some weights on the nets (e.g. 1/#(cells)) we can also incorporate them in the definition of the net weight.

Now we analyze how fast we can run procedures $PLACE(C_i)$ and $COLLAPSE(C_{i-1}, C_i)$. The procedure $PLACE(C_i)$ needs constant time because we can use pointers to $l_i = l(C_i)$ and $r_i = r(C_i)$. The implementation of $COLLAPSE(C_{i-1}, C_i)$ is more tricky. We first find which cell has the longer list (C_{i-1} or C_i) and then the positions from the shorter list are shifted left or right by the width of the "longer" cell depending on the relative position of the "longer" cell (left or right). Next, we insert shifted positions of the shorter list into the longer list. Each time a new position is inserted we update the minimum interval according to the net weight of that position. Note that the RBST implementation of all lists is very fast. We conclude that the runtime of the procedure $COLLAPSE(C_{i-1}, C_i)$ implemented with RBSTs is $O(\min\{M_i, M_{i-1}\} \cdot \log(\max\{M_i, M_{i-1}\}))$.

The number of entries in all Set_i's is $2m$. Thus the runtime of CA is $O(m \log m)$ since

$$\sum_{i=1}^{n}(min \cdot \log max) \leq \log(2m) \cdot \sum_{i=1}^{n} min \leq 2m \log(2m)$$

7 Heuristics for Improving the Cell Ordering

In this section we deal with the problem of finding an improved (sub)optimal ordering of cells in a single row when all cells in all other rows are fixed. As noted above, this is an instance of a well-known NP-hard problem [8].

Cell Ordering Problem = the Single-Row Problem where the left-to-right order of cells is not fixed.

We have implemented several heuristics for the Cell Ordering Problem, e.g., based on the order of middle points of the minimum intervals of cells. In our experience, initial cell orderings found in standard-cell placements by industry tools are quite good, and difficult to improve. The Swapping Heuristic below is the most successful approach we have found so far.[7]

```
Swapping Heuristic
For each cell, compute its cost curve and cost curve min point.
  (treat all cells in other rows as fixed)
Repeatedly iterate down the row doing:
  **For** every adjacent pair of cells that would overlap or cross
        when placed at their min points **do**
    **If** a superior relative placement is possible when swapped
    **then** swap the order of cells in the pair
    **else** don't swap, and remember not to try this pair again
Repeat previous step until no pairs would swap
```

8 Experimental Results and Future Directions

We have implemented both the Prefix and Clumping algorithms for the Single-Row Problem, as well as the swapping heuristic for improving the row ordering. Our testbed consists of industry standard-cell designs in Cadence LEF/DEF format, which we place with a leading industry placement tool.

Given the final output of the industry placer, we apply either the Prefix or the Clumping algorithm to optimize wirelength incident to each single row of the design. An iteration of our approach processes all rows of the design, one row at a time;[8] we continue iterations until improvement from an iteration is less than 0.001%. We have also studied our exact single-row algorithms in combination with the swapping heuristic (i.e., the swapping heuristic is applied to reorder cells in a given row, just before the wirelength minimization is applied). Table 1 shows the percentage wirelength improvement and CPU times on a 140MHz Sun Ultra-1 workstation for five industry test cases. We can see that surprisingly significant wirelength reductions (an average of 6.58%) are obtained in very little time.[9] The swapping heuristic adds slight further reduction in wirelength.

Our results indicate that large wirelength reductions are possible, even beyond the results of industry placement tools, when optimal single-row placement with extra sites (white-space) is applied iteratively to the rows of a standard-cell layout. Our current work seeks to verify the routability of the resulting improved placements, and to define more detailed (e.g., routability-driven) single-row placement objectives that are also amenable to efficient optimal solutions. More generally, we are investigating the utility of optimal solution approaches at a number of phases within a "successive approximation" methodology for standard-cell placement.

Benchmarks			Algorithm	Prefix		Clumping	
# cells	# rows	# sites	Heuristic	None	Swap	None	Swap
4155	46	41610	improve %	9.39	13.38	9.49	13.46
			runtime (sec)	56.8	94.1	31.8	69.4
11471	42	78036	improve %	6.00	6.18	6.00	6.22
			runtime (sec)	182.9	289	68.9	198
12260	610	128893	improve %	1.13	1.38	1.15	1.40
			runtime (sec)	13.9	35.1	14.8	40.2
7309	56	47152	improve %	7.43	8.13	7.45	8.08
			runtime (sec)	46.5	125.7	35.1	95.9
8829	60	98760	improve %	3.70	3.54	3.76	3.77
			runtime (sec)	34.0	43.9	20.7	91.8

Table 1: Experimental results for five industry test cases.

Acknowledgments. We thank Igor L. Markov and Devendra Vidhani for valuable discussions.

References

[1] D.Adolphson and T. Hu, "Optimal Linear ordering," *SIAM J. Appl. Math.*, vol.25, pp. 403-423, 1973.

[2] I. Cederbaum, "Optimal backboard ordering through the shortest path algorithm," *IEEE Trans. Circuits Syst.*, vol. CAS-21, pp. 626-632, 1974.

[3] P. Chin and A. Vannelli, "Interior point methods for placement," *IEEE Int. Symp. on Circuits and Syst.*, 1994, pp.169-172.

[4] H. Cho and C. Kyung, "A heuristic cell placement algorithm using constrained multistage graph model." *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 1205-1214, 1988.

[5] S. Chowdhury, "Analytical approaches to the combinatorial optimization in linear placement," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 630-639, 1989.

[6] J. Cohoon and S. Sahni, "Heuristics for the board permutation problem," in *Proc. Int. Conf. Computer-Aided Design*, 1983, pp. 81-83.

[7] T. H. Cormen and C. E. Leiserson and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP Completeness.* W. H. Freeman, San Francisco, 1979.

[9] S. Goto, I. Cederbaum, and B. Ting, "Suboptimal solution of the backboard ordering with channel constraint," *IEEE Trans. Circuits Syst.*, vol. CAS-24, pp. 645-652, 1977.

[10] S. Kang, "Linear ordering and application to placement", in *Proc. 20th Design Automation Conf.*, 1983, pp. 457-464.

[11] Y.Saab and V. Rao, "Linear ordering by stochastic evolution", in *Proc. 4th CSI/IEEE Int. Symp. VLSI Design*, New Delhi, India, 1991, pp. 130-135.

[12] Y.Saab, "An improved linear placement algorithm using node compaction," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 952-958, 1996.

[13] A. Sangiovanni-Vincentelli and M. Santomauro, "A heuristic guided algorithm for optimal backboard ordering," in *Proc. 13th Ann. Allerton Conf. Circuit Syst. Theory*, 1975, pp. 916-921.

[14] TimberWolf Systems, Inc., http://www.twolf.com.

[15] S. Yamada, H. Okude, and T. Kasai, "A hierarchical algorithm for one-dimensional gate assignment based on contraction of nets," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 622-629, 1989.

[7] We are currently investigating methods inspired by LP-based heuristics for global placement ([3]). E.g., one approach is to apply LP for the single-row placement instead of the entire layout, and derive orderings from the result.

[8] Actually, we process each sub-row of the design in turn; a sub-row is typically delimited by vertical M2 power stripes. Cells must remain within their original subrows.

[9] Notice that the Prefix and Clumping results are slightly different, even though both are optimal algorithms for the same problem. This is because there are often multiple optimal placement solutions for a given cell ordering in a row; our two implementations are not guaranteed to return the same optimal solution, and thus their solution paths can diverge. Also, while the Clumping algorithm is faster, its runtimes can occasionally be greater because it makes more iterations.