An Efficient and Effective Detailed Placement Algorithm

Min Pan, Natarajan Viswanathan and Chris Chu
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011
Email: {panmin, nataraj, cnchu}@iastate.edu

Abstract— **In the past few years there has been a lot of research in the area of global placement. In comparison, not much attention has been paid to the detailed placement problem. Existing detailed placers either fail to improve upon the excellent solution quality enabled by good global placers or are very slow. To handle the above problems we focus on the detailed placement problem. We present an efficient and effective detailed placement algorithm to handle the wirelength minimization problem. The main contributions of our work are: (1) an efficient Global Swap technique to identify a pair of cells that can be swapped to reduce wirelength; (2) a flow that combines the Global Swap technique with other heuristics to produce very good wirelength; (3) an efficient Single-Segment Clustering technique to optimally shift cells within a segment to minimize wirelength.**

On legalized *mPL5* global placements on the IBM Standard-Cell benchmarks [1], our detailed placer can achieve 19.0%, 13.2% and 0.5% more wirelength reduction compared to *Fengshui5.0*, *rowIroning* and *Domino* respectively. Correspondingly we are 3.6×, 2.8× and 15× faster. On the ISPD05 benchmarks [19], we achieve 8.1% and 9.1% more wirelength reduction compared to *Fengshui5.0* and *rowIroning* respectively. Correspondingly we are 3.1× and 2.3× faster.

I. Introduction

In recent years, the role of placement in the physical design of large chips has become very critical. Placement tools are not only used to place the cells for the subsequent routing step, but are also used to guide synthesis and floorplanning stages. Placement is no longer a point tool in the current physical synthesis flow [21]. It has become a major contributor to timing closure results.

Traditionally, placement is separated into two stages, *global* and *detailed* placement. The main purpose of global placement is to distribute the cells evenly over the placement region and optimize certain objectives such as wirelength. As we want to maintain a global view, some approximation has to be made to simplify the problem. Also, the global placement pays more attention to the relative positions among cells globally. Hence, it neglects some local problems. Detailed placement works on the legalized placement to further improve the solution quality. It is more constrained than global placement as it optimizes the objectives by transforming one legal placement solution into another. Because of this nature, more accurate models such as half-perimeter wirelength are used in detailed placement.

Acknowledgements: This work was partially supported by the Semiconductor Research Corporation under Task ID 1206.

Previous literature has mainly focused on the problem of global placement. These algorithms apply various approaches including analytical placement [7], [9], [11], [14], [16], [18], [22], simulated annealing [20], [24], and partitioning / clustering [4], [6], [26]. Recently, there have been significant improvements in terms of both solution quality and runtime. On a set of IBM benchmarks, [7], [16] reported very good wirelength and *FastPlace* [22] achieved runtimes many times faster than other state-of-the-art placement algorithms.

However, compared to global placement, there has been much less work in terms of detailed placement. [2], [3], [5] employed a window-based branch-and-bound method for detailed placement. Alternatively, *Dragon* [24] used a greedy cell exchange algorithm. *Domino* [8] transformed the placement problem into a transportation problem that was solved using a network flow algorithm. Kahng et al. [15] employed combinatorial techniques to perform legalization and detailed placement based on several different objectives. In [17], the single-row problem was solved optimally using a dynamic programming approach. In [13], Hur and Lillis proposed a technique called optimal interleaving and also incorporated the dynamic clustering technique [12].

Current detailed placement techniques are either not very effective or too slow. The window-based technique is very local if the window size is small. If a big window is used, the runtime is not affordable. *Domino* is considered a very good detailed placer but it consumes a lot of runtime. In [23], it was observed that *Domino* can achieve an average wirelength reduction of 5.9% over *FastPlace* on the IBM benchmarks. Hence, we believe that significant improvements in terms of wirelength reduction can be made at the detailed placement stage. It was also observed that the *FastPlace+Domino* flow was on average 7.6× slower than *FastPlace*. Considering that current global placers can generate high-quality solutions in a very short time, it is necessary to have efficient detailed placers to further improve the solution quality of global placement.

In this paper, we present an efficient and effective detailed placement algorithm that can work on both row-based standard cell placement and placement in the presence of fixed macros. The main contributions of our work are:

- An efficient Global Swap technique to identify a good pair of cells to swap globally based on their optimal positions while all other cells are fixed.

- A Vertical Swap technique that swaps a cell with a nearby cell in the segment above or below so as to move it in the direction of its optimal position.
- A Local Re-ordering technique that re-orders consecutive standard cells locally to reduce the wirelength.
- A Single-Segment Clustering technique that places standard cells optimally within a segment. It solves the same problem as the Single-Row Problem in [17]. Compared with the dynamic programming method of [17], this technique can get the optimal solution much faster.

We compare our detailed placer with three detailed placers: postprocessing in *Fengshui5.0*, *rowIroning* from the *Capo9.1* package and *Domino* on two benchmark suites: IBM Standard-Cell benchmark suite [1], [22] and ISPD05 benchmark suite [19]. On the IBM benchmarks, on global placements generated by *mPL5* [7] and legalized by the *Placement Utility* from the *Capo9.1* package, our detailed placer can achieve 19.0%, 13.2% and 0.5% more wirelength reduction compared to *Fengshui5.0*, *rowIroning* and *Domino* respectively. Correspondingly we are 3.6×, 2.8× and 15× faster. On the ISPD05 benchmarks, we achieve 8.1% and 9.1% more wirelength reduction compared to *Fengshui5.0* and *rowIroning* respectively. Correspondingly we are 3.1× and 2.3× faster.

The rest of the paper is organized as follows: Section II provides an overview of our detailed placement algorithm. Section III describes the techniques used in our detailed placer. Experimental results and discussions are presented in Section IV followed by our conclusions in Section V.

II. Overview

Our detailed placer works on a legalized placement. The placement can be a legalized row-based standard cell placement or a legalized placement with all macros fixed. For standard cell placement, the placeable segments are the rows specified in the placement region. For the placement with macros, the whole placement region is divided into placeable segments based on the macros and placement blockages. In both cases, the detailed placer only works on the standard cells in the placeable segments to improve the wirelength.

The detailed placer consists of four key techniques: Global Swap, Vertical Swap, Local Re-ordering and Single-Segment Clustering. Global Swap is the technique that gives us the most benefit. For any cell i, it tries to identify a good swap pair, so that i after the swap would be in the position that gives the best wirelength when all other cells are fixed. Because the target position can be close to or far from the current position of i, this technique moves a cell globally to reduce the wirelength. The Vertical Swap tries to swap a cell i with another nearby cell in the segment above or below so as to move i towards its best position. Although this technique is similar to Global Swap, it is more local and faster. It tries to fix some local problems in the vertical direction. In the horizontal direction, we employ a Local Re-ordering technique to find a better order for consecutive standard cells within segments. Finally, a Single-segment Clustering technique is developed to optimally place the standard cells within a segment while cells in all other segments are fixed. A near-optimal implementation based on this technique has the time complexity linear to the number of cells in a segment.

```
**Detailed Placement Algorithm**

Perform Single-Segment Clustering
**Repeat**
    Perform Global Swap
    Perform Vertical Swap
    Perform Local Re-ordering
**Until** no significant improvement in wirelength
**Repeat**
    Perform Single-Segment Clustering
**Until** no significant improvement in wirelength
```

Fig. 1. Detailed Placement Flow.

The flow of our detailed placement algorithm is summarized in Figure 1. We first apply the Single-Segment Clustering technique to obtain a relatively good starting solution for the main steps of the algorithm. In the main loop, Global Swap, Vertical Swap and Local Re-ordering are employed to reduce the wirelength until there is no significant improvement. Finally, we re-apply the clustering to get better positions for the cells within the segments without changing their order.

III. Detailed Placement Techniques

In this section, we describe the techniques used in our detailed placer.

A. Global Swap

The basic idea behind Global Swap is to find the "optimal region" for a cell i in the placement region and swap i with a cell j or a space s in the "optimal region". We define the "optimal region" and describe the method to find it in Section III-A.1. In Section III-A.2 we discuss the penalty charged for any overlap created during a swap. Finally, in Section III-A.3 we describe swapping based on the "optimal region" and the penalty for overlap.

1) Optimal Region: Given all other cells in the circuit are fixed, the "optimal region" for a cell i is defined as the region to place i where the wirelength is optimal. This region is determined based on the median idea of [10].

For any cell i, we traverse all the nets connecting to it (noted as N_i) and find their bounding boxes. Here, cell i is excluded from the nets when computing their bounding boxes. For each net $p \in N_i$, we find its bounding box ($x_l[p]$, $x_r[p]$, $y_l[p]$, $y_u[p]$ - the left, right, lower and upper boundaries). From [10], the optimal position for i is given by (x_{opt}, y_{opt}), where x_{opt} and y_{opt} are the medians of the x series ($x_l[1]$, $x_r[1]$, $x_l[2]$, $x_r[2]$, ...) and y series ($y_l[1]$, $y_u[1]$, $y_l[2]$, $y_u[2]$, ...) of bounding boxes. In general, the optimal position is a region rather than a point as the total number of elements in the x and y series are even. This region is the "optimal region" for cell i. In some cases, the "optimal region" can degrade to a point or a line

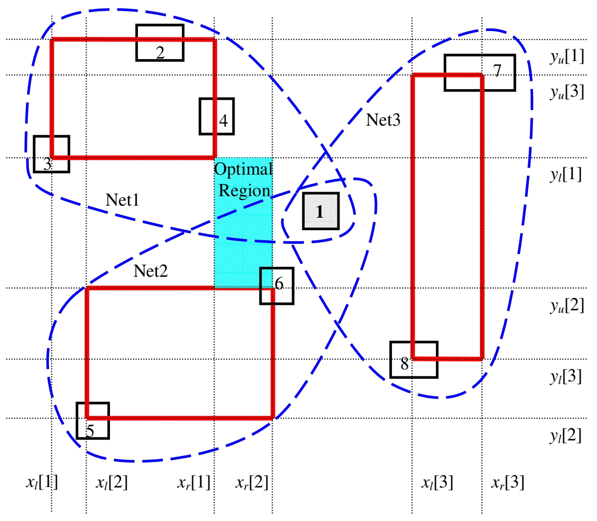


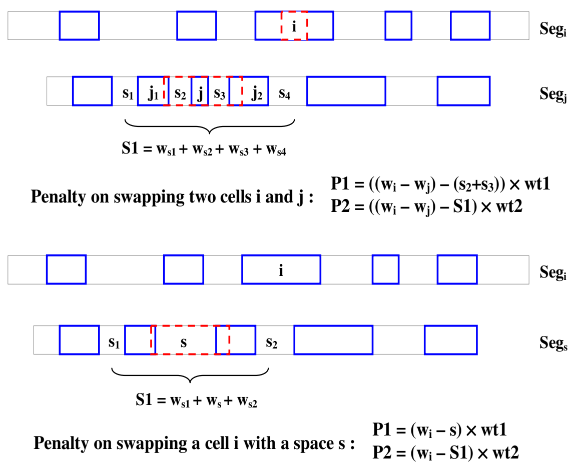
Fig. 2. Optimal Region.

when the two medians of the x and/or the y series carry the same value. Figure 2 shows the optimal region for cell 1. There are three nets connecting to cell 1 ($Net1$, $Net2$ and $Net3$). The nets are denoted by closed dashed lines: $Net1$ includes cells 1, 2, 3 and 4; $Net2$ includes cells 1, 5 and 6; $Net3$ includes cells 1, 7 and 8. The bold boundary boxes are the bounding boxes for the nets excluding cell 1. The light lines are the grids constructed by the x series ($x_l[1]$, $x_r[1]$, $x_l[2]$, $x_r[2]$, $x_l[3]$, $x_r[3]$) and y series ($y_l[1]$, $y_u[1]$, $y_l[2]$, $y_u[2]$, $y_l[3]$, $y_u[3]$). The shadowed region is the optimal region for cell 1.

2) Penalty on Overlap: For a cell i, although we find its optimal region, it may not be possible to move it into the optimal region. The reason being that since the detailed placer transforms one legalized placement to another, it is not allowed to have any overlap among cells. Therefore, we need to consider the effect of any resulting overlap among cells when swapping or moving cell i. If a swap causes an overlap, a consequent legalization has to be done to resolve it. Therefore, we need to have a method to model the overlap and consider it when we try to make a swap. We now discuss the method to add a penalty on a swap when it creates an overlap.

If we swap two cells that are not of the same size, the space at the smaller cell may not be enough to hold the bigger cell. Also, If we swap a cell with a space, the space may be smaller than the cell. Both cases may lead to an overlap after swapping. To resolve this overlap, the cells in the segment need to be shifted. We introduce a penalty on this shifting effect. In addition, if the total width of the cells in a segment after swapping is greater than the segment width, we just neglect the swap.

For swapping two cells, if there is no overlap after the swap, no penalty is applied; otherwise, a penalty is charged. For swapping a cell with a space, if the space is equal to or bigger than the cell size, no penalty is applied. Otherwise, a penalty is charged. In order to characterize the penalty more accurately, we have two types of penalties: $P1$ and $P2$. $P1$ is the penalty on shifting the closest two cells to resolve overlap.



i
Seg_i
s_1
j_1
s_2
j
s_3
j_2
s_4
Seg_j
$S1 = w_{s1} + w_{s2} + w_{s3} + w_{s4}$
Penalty on swapping two cells i and j :
$P1 = ((w_i - w_j) - (s_2+s_3)) \times wt1$
$P2 = ((w_i - w_j) - S1) \times wt2$
i
Seg_i
s_1
s
s_2
Seg_s
$S1 = w_{s1} + w_s + w_{s2}$
Penalty on swapping a cell i with a space s :
$P1 = (w_i - s) \times wt1$
$P2 = (w_i - S1) \times wt2$

Fig. 3. Penalty for swapping two cells with different sizes and swapping a cell with a space.

$P2$ is the penalty on shifting cells other than the closest two cells. Figure 3 illustrates an example to compute $P1$ and $P2$. The bold boxes are the cells and the light boxes are segments. The dotted lines show the positions after swap for the cells swapped. Consider the case we swap cell i (width w_i) in segment seg_i with another cell j (width w_j) in segment seg_j that is in the optimal region of i. Assume the size of i is larger than j. The two cells left and right to j are j_1 and j_2. The two closest spaces left to j are s_1 and s_2, and the two closest spaces right to j are s_3 and s_4. The total width of spaces s_1, s_2, s_3, s_4 is S1. $P1$ is the wirelength increase caused by shifting j_1 and j_2. If $S1 \geq (w_i - w_j)$, the total shift of j_1 and j_2 to resolve overlap is $(w_i - w_j) - (s_2 + s_3)$. We make $P1$ proportional to this shift. If $S1 \leq (w_i - w_j)$, only shifting j_1 and j_2 cannot resolve the overlap and we need to shift more cells in seg_j. $P2$ is the penalty of shifting cells other than j_1 and j_2 in seg_j. In this case $P2$ is proportional to the shift on cells other than j_1 and j_2, which is $(w_i - w_j) - S1$. Hence, we set $P1$ and $P2$ as follows:

$$\begin{aligned} P1 &= ((w_i - w_j) - (s_2 + s_3)) \times wt1 \\ P2 &= ((w_i - w_j) - S1) \times wt2 \end{aligned} \tag{1}$$

where $wt1$ and $wt2$ are the two weights on the shift. For the case where we swap i with a space s, the way to get the penalty is similar to that for swapping two cells. The only difference is that the width difference is $w_i - 0 = w_i$ and $S1$ is the sum of the widths of s, the closest space left to s and the closest space right to s.

Since the shifts in $P1$ and $P2$ have the dimension of length, the two weights $wt1$ and $wt2$ are just constants with no dimension. Because we do not want to disturb the original placement too much, large overlap is discouraged by setting $wt2$ much higher than $wt1$.

3) Global Swap Based on Optimal Region: Based on the optimal region and the penalty on overlap, we develop a Global Swap technique to swap each cell with a cell or space in its optimal region. Since there could be several cells and spaces in the optimal region, we have many choices. We use a term "benefit" B as a measure for selecting the cell or space in the optimal region. The "benefit" for a swap has two components: one is the the difference between the total wirelength before and after the swap, the other is the penalty charged on the created overlap. If the wirelength before and after the swap are W_1 and W_2, respectively, the "benefit" can be obtained by equation (2).

$$B = (W_1 - W_2) - P1 - P2 \tag{2}$$

If $B > 0$, it means that we will benefit from the swap. Otherwise, the resulting placement is worse than original. Of course, the "benefit" we compute is not accurate because the real wirelength change due to resolving the overlap is hard to measure. We only use a simple penalty on shifting cells to model this wirelength change. Based on the "benefit", we do the swapping as follows. For each standard cell i, we find its optimal region and try to swap it with every cell j and space s in the optimal region of i. We measure the "benefit" for each swap and pick the j or s with the best "benefit" to perform the swap. If the best "benefit" has a value less than zero we do not make a swap as it would increase the wirelength.

In this technique, we look at the optimal region for a cell to find a good target position. The optimal region can be close to or far from the current position. Hence, compared to the traditional window-based branch-and-bound methods, our Global Swap technique has a more global view when repairing the positions of cells. In Table I, we show the distribution of the cells according to the distance of the cells from their respective optimal regions before and after 1 iteration of Global Swap for the circuit ibm01. The unit of the distance is the standard row height. Distance 0 means the cell is in its optimal region. It is clear that our technique is very effective in moving cells towards their optimal region.

TABLE I
DISTRIBUTION OF CELLS BASED ON THE DISTANCE FROM THEIR OPTIMAL REGIONS BEFORE AND AFTER 1 ITERATION OF GLOBAL SWAP

Distance	0	(0,1]	(1,2]	(2,3]	(3,4]	> 4
before	30.0%	36.8%	18.0%	6.4%	3.4%	5.4%
after	33.0%	39.3%	17.1%	5.3%	2.2%	3.1%

In the actual implementation, to save runtime, for a selected cell, we do not pick the cell with the best "benefit" in its optimal region. Instead, we pick the first "good" cell that can give us certain "benefit". Another issue is that after swapping two cells with different sizes, the placement is no longer legal. Overlaps are created around the bigger cell and spaces are created around the smaller cell. We need to re-legalize the segments containing the two cells. However, legalization after every swap will be very time consuming. In the implementation, we legalize the whole placement after all the segments has been traversed. Of course, we will lose some accuracy on the positions of cells, but experiments show that this inaccuracy does not affect the final wirelength significantly.

B. *Vertical Swap*

In the Global Swap technique, for a cell i, we may not find a good candidate cell or space in its optimal region to swap with it. There could be two reasons for this. First, the size of i is large and the optimal region of i is congested. Hence, the segments that span the optimal region cannot hold i. Second, in order to hold i, many cells have to be shifted to legalize the placement which introduces a high penalty.

To increase the possibility for a good swap and reduce the vertical wirelength locally, we have a Vertical Swap technique very similar to the Global Swap. The idea of Vertical Swap is to move a cell vertically toward its optimal region. This technique is not as greedy as Global Swap. Every time it only moves a cell up or down by one row. For a cell i, if the optimal region is above / below the current position, a few nearby cells above / below i are considered to be candidates. We use the same penalty as in Global Swap to estimate the effect of overlap and pick the best candidate to swap with i. We observe that if we interleave the Vertical Swap with Global Swap, the wirelength decrease is faster than only applying Global Swap. We believe that this is because the Vertical Swap is not very greedy and has more flexibility in moving the cells. At the same time, it may increase the possibility for Global Swap. In addition, this technique is much faster than Global Swap because for each cell, the number of candidate cells considered for swap are much less than in Global Swap.

C. *Local Re-ordering*

With Vertical Swap fixing local vertical errors, we need a technique to fix local horizontal errors. Although Global Swap can also fix horizontal errors, it is quite expensive to use it to fix local problems. Therefore, we propose a very fast Local Re-ordering technique to handle this problem. For any n consecutive cells within a segment, we try all possible left-right ordering of cells and pick the order giving the best wirelength. In this technique, we also need to decide the position of the cells in each order. To speed-up the technique, we consider the cells as a group and make the left boundary of the group as the left boundary of the first cell in the original order and the right boundary of the group as the right boundary of the last cell in the original order. Then for each order, we keep the left and right boundaries of the group and evenly distribute the cells inside the group. Since we have the Single-Segment Clustering technique to take care of the cell positions, we do not pay much attention to the exact positions of the cells during Local Re-ordering.

In our detailed placer, we set $n = 3$. The reason is that $n = 2$ means pairwise swapping and it is too constrained. But if we choose $n = 4$, it will be 4 times slower and the improvement is not so significant. Compared to the conventional window-based technique, Local Re-ordering has a 3-cell window in one row and is very local. But since we have the Global Swap technique, it is only used to efficiently fix local errors.

D. Single-Segment Clustering

After the main loop of the detailed placer we fix the segments and the ordering within the segments for all the standard cells. We now want to further reduce the wirelength by moving the cells inside the segments. For a legalized placement, if we fix the order of the cells in one segment and the positions of the cells in all other segments, the problem becomes a fixed-order single segment problem described below.

Fixed-Order Single Segment Placement Problem: Given a segment S in the placement region with n standard cells $C1, C2, ..., Cn$, whose left-to-right order is fixed (C_i is left to C_j if $i < j$). All cells not in S are fixed. Find a non-overlapping placement for the segment S so that the total half-perimeter wirelength is minimized.

This problem is basically the same as the Single-Row Problem in [17]. In [17], the authors proposed a dynamic programming algorithm to solve the problem optimally. In the following part, we describe a more efficient algorithm that can also solve the problem optimally.

First, we define some terms used in our algorithm. A *cluster* is a standard cell or a group of standard cells abutted together (retaining the original order of standard cells). *Clustering* is the operation to abut two clusters to form a new cluster (the width of the new cluster is the sum of the widths of the original clusters). The wirelength function of x-coordinate of a cluster is a convex piecewise linear function W(x) when all other objects are fixed. The slopes for the linear pieces are $..., -3, -2, -1, 0, 1, 2, 3, ...$ The slope 0 part is the optimal region in x-direction for the cluster. The points where the function changes slope are called *bounds*. These bounds are the left and right boundaries of the bounding boxes for the nets connecting to the cluster. *Optimal Region Center* of a cluster is the middle point of the optimal region in x-direction when all the objects (standard cells and macro blocks) not part of the cluster are fixed.

In order to find the optimal region for a cluster C in segment S, we need to fix the positions for all the other objects. But the standard cells in S are not fixed. Therefore, if C has connections to any standard cells in S, the bounds for C cannot be determined. However, since we fix the order of the standard cells in S, we know the left-right orders between the cells. We use this information to get the bounds so that the optimality of the solution will not be affected. The method to get the bounds is as follows. When computing the bounding box for any net N connecting to C, if N is connecting to a standard cell C' in S, we will assume C' at the end of the segment S, i.e., if C' is left to C, we assume C' is at the left end of segment S; otherwise, C' is at the right end of segment S. Although we are not using the real position for C', we will not affect the optimality of the position of C because the left-right order of C and C' has to be maintained. The main idea of the algorithm is to put every cluster at its *Optimal Region Center*. If there is overlap between two clusters, we perform clustering and form a new cluster. The new cluster will not be broken at any later stage. Then we put the new cluster at its *Optimal Region Center*. We iteratively perform clustering until all the cells are put at *Optimal Region Center* without any overlap. If any optimal region boundary is out of the segment range, we will assign it at the closest boundary. In this way, no cell will be put out of the segment. The pseudo-code of the Single-Segment Clustering Algorithm is given in Figure 4.

```
Single-Segment Clustering Algorithm

num_old_cluster ← n
Initialize old_cluster[i] as standard cell Ci, i=1, 2, ..., num_old_cluster.
do
  Find the bounds list and the Optimal Region Center Xic for Ki,
      and set X(old_cluster[i]) = Xic
  newcount ← 1 // the count for the number of new clusters
  new_cluster[1] ← old_cluster[1] // initialize the first new cluster
  j ← 1
  while(j < num_old_cluster)
    do
      if new_cluster[newcount] and old_cluster[j+1] has overlap
        Cluster new_cluster[newcount] and old_cluster[j+1] to form the
            new new_cluster[newcount]
        Merge the bounds list for new_cluster[newcount] and old_cluster[j+1]
            to get the new bounds list for new_cluster[newcount]
        Find the Optimal Region Center Xc for new_cluster[newcount]
            based on the new bounds list
        X(new_cluster[newcount]) ← Xc
      else
        newcount ← newcount + 1 //begin a new cluster new_cluster[newcount+1]
        j ← j+1

  num_old_cluster ← newcount
  old_cluster[i] ← new_cluster[i] (i=1, ..., newcount)
until no overlap among old_cluster[i], ( i=1, ..., num_old_cluster )
Assign the Ci ( i=1, 2, ..., n) to the positions according to the positions of the
old_cluster[j] ( j=1, 2, ..., num_old_cluster) they belong to
```

Fig. 4. Single-Segment Clustering Algorithm.

Theorem 1 The Single-Segment Clustering Algorithm finds the optimal solution for the *Fixed-Order Single Segment Placement Problem.*
Due to the page limit, we only give the sketch of the proof.

Proof: It is not hard to see that if the clusters are formed correctly, then the solution obtained by our algorithm is optimal. To show that we will not form wrong clusters, assume on the contrary that the clustering in the optimal solution is different from our solution.

Consider a gap in the optimal solution surrounded by a pair of cells a and b that are in the same cluster in our solution. Suppose a and b are clustered together when we merge clusters A and B in some step of our algorithm. See Fig. 5 for an illustration. Without loss of generality, we can assume there is no gap within cluster A and within cluster B in the optimal solution. Otherwise, we can consider the gap within cluster A or cluster B instead. Since we merge cluster A and cluster B together at some point, A and B cannot be at the optimal region at the same time if their order is not changed. For any solution, either A wants to move left or B wants to move right (or both) to reduce the wirelength. We can always generate a better wirelength than the optimal solution by moving either A or B towards the gap without creating any overlap. This is a contradiction. Thus, our solution should be optimal. ■

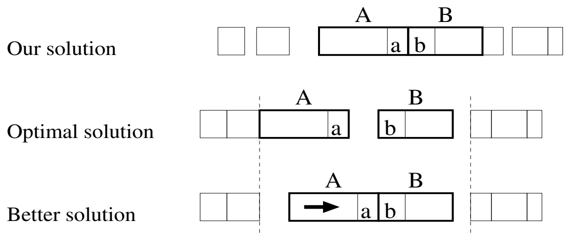


Fig. 5. Proof of optimality of Single-Segment Clustering Algorithm.

We now analyze the complexity of the algorithm. There are n cells in total, and the maximum number of clustering is $n-1$. In the clustering operation, every step needs constant time except merging the two bounds lists. The merge takes linear time to the number of bounds m. The complexity of the algorithm is $O(nm)$. However, in practice, it can be much better. In our implementation, we are not keeping all the bounds for the clusters. Instead, we only keep a constant number of bounds for every cluster. Therefore, the merge also takes constant time. The total complexity of the algorithm is $O(n)$. Of course, it will compromise the optimality, but experiments show that even using a small constant will not degrade the solution appreciably. In implementation, the constant we choose is 16. Table II shows different results when using different constants on the 8^{th} segment of ibm01. It shows that even if a small constant is used, the result can be very close to optimal. Moreover, the segment we choose here is one that has a lot of room to reduce the wirelength. For the 1^{st} to 7^{th} segment in ibm01, just using 8 bounds can achieve the optimal solution.

TABLE II

THE RELATIONSHIP BETWEEN # BOUNDS AND WIRELENGTH DECREASE ON THE 8TH SEGMENT OF IBM01

#bounds	4	8	12	16	20	opt
WL dec	13600	14060	14210	14377	14425	14425

Although this algorithm can give the optimal solution for a segment, we still need to run it iteratively as it is only optimal when all cells not in the current segment are fixed. Since we change the cell positions segment by segment, we need to run several iterations to find good positions for the cells.

IV. Experimental Results

We consider the ISPD04 IBM Standard-Cell Benchmark suite [1], [22] and the ISPD05 Benchmark suite [19] for our experiments. The placement tools considered are our new detailed placer, post-processing in *Fengshui 5.0* [26], *rowIroning* in the *Capo9.1* package [4], and *Domino* [8]. For post-processing in *Fengshui 5.0*, we use the default control string used in the complete flow of *Fengshui 5.0* which is, -reorder "r,4,4:r,4,2:r,4,2:r,4,1:r,4,1:r,4,1". For *rowIroning*, we use the default options on "-ironPasses -ironWindow -ironOverlap -ironTwoDim" used in the complete *Capo9.1* flow.

We run *mPL5* and *Capo9.1* to get the global placements for both IBM and ISPD05 benchmarks. Since we have to disable both the legalizer and detailed placer in *mPL5*, the global placements created by *mPL5* are not legalized. We therefore use the Placement Utilities in the *Capo9.1* package to legalize the *mPL5* global placements. For *Capo9.1*, we disable the greedy swapping and *rowIroning* in the overall flow to get the legalized global placements. All the results are generated on a Linux machine with Intel Pentium 4, 3.00GHz CPU and 2GB memory.

The half-perimeter wirelength and runtime results on IBM benchmarks for *Fengshui5.0*, *rowIroning*, *Domino* and our detailed placer are reported in Tables III and IV. Table III gives the results for different detailed placers on the global placements generated by mPL5+Legalizer. On average our detailed placer gives 19.05% better wirelength with a 3.62× speed-up over *Fengshui5.0*. Compared with *rowIroning*, we are 13.22% better in wirelength and 2.79× faster. Compared with *Domino*, we can achieve 0.54% better wirelength and are around 15× faster. In addition, on average, we can reduce the wirelength of the legalized placement by nearly 30%. This shows that there is a lot of room for the detailed placer to improve the global placement solution. From Table IV, for the *Capo9.1* global placements, our detailed placer is 1.17% better than *Fengshui 5.0* in wirelength with a 4.48× speed-up. We are also 1.91% better than *rowIroning* in wirelength and 5.66× faster. Compared with *Domino*, we are 0.55% better in wirelength and 13.45× faster.

Tables V and VI show the comparison results on the recent ISPD05 benchmarks. This benchmark set has fixed/movable macros with a large number of cells. For bigblue4 we were unable to obtain the global placement solution of *Capo9.1* as the placer ran out of memory on our machine. Also, we were unable to generate feasible solutions using *Domino* on this set of benchmarks. Hence, only *Fengshui 5.0* and *rowIroning* are used for comparison. Table V gives the results for different detailed placers on the global placements generated by mPL5+Legalizer. On average our detailed placer gives 8.06% better wirelength with a 3.05× speed-up over *Fengshui5.0*. Compared with *rowIroning*, we are 9.12% better in wirelength and 2.29× faster. From Table VI, on the *Capo9.1* global placements, our detailed placer is 2.04% better than *Fengshui 5.0* in wirelength with a 2.81× speed-up. We are also 0.89% better than *rowIroning* in wirelength and 2.18× faster.

From the comparisons made in Tables III–VI, our detailed placer can achieve better solution quality in much less runtime as compared to other detailed placers. For the ISPD05 benchmarks, our detailed placer has lesser speed-up over *Fengshui5.0* and *rowIroning* because it runs for more iterations to reach the stopping criterion, whereas *Fengshui5.0* and *rowIroning* have fixed number of passes to run the algorithms. On the global placements generated by *Capo9.1*, all the detailed placers get lesser improvement than on the global placements generated by mPL5+Legalizer. A possible reason could be that *Capo9.1* has done many local optimizations during partitioning at the lowest level. Therefore, most of the local errors have been fixed. Another interesting observation is that although the wirelengths of the global placements generated by mPL5+Legalizer are much higher than that generated by *Capo9.1*, the final results obtained on the mPL5+Legalizer

TABLE III

Comparison of Detailed Placers on mPL5+Legalizer Global Placement on IBM benchmarks (1 Average over 17 circuits, Fengshui5.0 failed on ibm08, 2 Average over 17 circuits, Domino failed on ibm17)

	mPL+LG	Ours			Fengshui5.0		RowIroning		Domino	
	WL(1e6)	WL(1e6)	Impv	runtime(s)	Impv ,	runtime/Our	Impv ,	runtime/Our	Impv ,	runtime/Our
ibm01	2.423	1.732	-28.49%	6	-14.99%	2.81	-18.15%	3.87	-29.79%	14.33
ibm02	4.745	3.701	-21.99%	14	-9.05%	2.51	-13.98%	2.72	-21.97%	9.42
ibm03	6.624	4.792	-27.67%	14	-12.36%	2.70	-17.27%	3.13	-28.16%	10.02
ibm04	8.696	5.893	-32.23%	20	-12.47%	2.32	-18.52%	2.73	-32.51%	14.11
ibm05	12.074	10.106	-16.30%	23	-7.21%	2.45	-10.75%	2.54	-16.94%	10.47
ibm06	7.390	5.335	-27.81%	15	-12.04%	4.67	-16.59%	4.19	-29.09%	24.24
ibm07	11.576	8.380	-27.61%	26	-11.47%	3.78	-17.05%	3.32	-27.46%	16.78
ibm08	12.714	9.361	-26.37%	77	NA	NA	-16.90%	1.28	-26.49%	7.98
ibm09	15.267	9.648	-36.80%	35	-14.40%	3.48	-20.79%	3.03	-36.45%	20.81
ibm10	26.403	17.665	-33.09%	53	-12.22%	3.23	-18.23%	2.73	-32.62%	23.22
ibm11	22.128	14.411	-34.87%	46	-12.95%	4.07	-19.71%	3.13	-34.73%	19.44
ibm12	32.378	22.803	-29.57%	61	-10.61%	2.97	-16.34%	2.53	-28.83%	21.12
ibm13	27.249	17.050	-37.43%	62	-13.31%	4.21	-19.69%	2.94	-36.50%	12.77
ibm14	47.110	32.006	-32.06%	117	-11.34%	5.04	-18.23%	2.56	-31.86%	14.86
ibm15	60.133	39.474	-34.35%	146	-11.21%	5.60	-17.27%	2.50	-34.58%	12.42
ibm16	69.489	43.892	-36.84%	199	-12.13%	4.40	-19.54%	2.05	-36.75%	13.62
ibm17	93.186	62.078	-33.38%	191	-10.91%	4.88	-17.04%	2.28	NA	NA
ibm18	67.687	41.759	-38.30%	484	-11.86%	2.50	-21.11%	0.92	-39.69%	7.03
			-30.84%		**-11.79%**[1]	**3.62**[1]	**-17.62%**	**2.79**	**-30.30%**[2]	**14.86**[2]

TABLE IV

Comparison of Detailed Placers on Capo9.1 Global Placement on IBM benchmarks (1 Average over 17 circuits, Fengshui5.0 failed on ibm08)

	CAPO	Ours			Fengshui5.0		RowIroning		Domino	
	WL(1e6)	WL(1e6)	Impv	runtime(s)	Impv ,	runtime/Our	Impv ,	runtime/Our	Impv ,	runtime/Our
ibm01	1.840	1.788	-2.83%	4	-2.44%	4.74	-0.78%	10.79	-2.68%	15.84
ibm02	3.850	3.715	-3.50%	7	-2.35%	5.11	-1.72%	9.02	-4.64%	15.91
ibm03	5.165	4.977	-3.64%	9	-2.19%	4.88	-1.48%	8.83	-2.84%	11.82
ibm04	6.151	5.938	-3.47%	17	-1.89%	2.98	-1.30%	5.54	-3.57%	7.85
ibm05	10.110	9.822	-2.84%	14	-1.07%	4.47	-0.66%	7.04	-2.85%	11.60
ibm06	5.628	5.415	-3.78%	19	-2.36%	3.89	-1.47%	5.60	-3.69%	19.92
ibm07	9.468	9.208	-2.74%	23	-2.01%	4.85	-1.16%	6.60	-2.18%	18.31
ibm08	9.933	9.582	-3.53%	70	NA	NA	-0.98%	2.36	-2.83%	5.87
ibm09	10.483	10.192	-2.77%	23	-2.31%	6.28	-1.48%	8.18	-1.40%	23.52
ibm10	19.271	18.723	-2.84%	50	-1.64%	3.84	-0.95%	4.95	-1.54%	18.53
ibm11	15.540	15.121	-2.69%	33	-1.92%	6.39	-1.27%	7.37	-1.35%	23.69
ibm12	24.833	24.055	-3.14%	90	-1.44%	2.28	-0.93%	2.97	-2.04%	8.87
ibm13	18.561	18.005	-2.99%	44	-2.07%	6.40	-1.31%	6.90	-1.68%	10.95
ibm14	34.573	33.655	-2.66%	131	-1.62%	4.90	-0.91%	3.85	-2.13%	9.59
ibm15	42.702	41.556	-2.68%	149	-1.63%	5.48	-1.05%	4.10	-2.68%	9.58
ibm16	49.597	48.173	-2.87%	192	-1.55%	4.81	-0.85%	3.53	-2.19%	9.25
ibm17	68.990	67.251	-2.52%	167	-1.37%	2.46	-0.81%	2.37	-1.63%	12.71
ibm18	45.020	43.750	-2.82%	218	-1.66%	2.38	-0.94%	1.84	-2.57%	8.34
			-3.02%		**-1.85%**[1]	**4.48**[1]	**-1.11%**	**5.66**	**-2.47%**	**13.45**

TABLE V

Comparison of Detailed Placers on mPL5+Legalizer Global Placement on ISPD05 benchmarks

	mPL+LG	Our			Fengshui5.0		RowIroning	
	WL(1e8)	WL(1e8)	Impv	runtime(s)	Impv	runtime/Our	Impv	runtime/Our
adaptec1	0.925	0.864	-6.53%	96	-3.63%	3.52	-2.85%	3.95
adaptec2	1.139	1.036	-9.05%	177	-4.99%	2.36	-3.81%	2.56
adaptec3	3.206	2.506	-21.84%	427	-6.83%	2.02	-6.65%	1.99
adaptec4	3.040	2.279	-25.02%	481	-9.45%	2.01	-7.71%	1.66
bigblue1	1.221	1.106	-9.38%	152	-6.45%	3.30	-4.25%	3.43
bigblue2	2.183	1.925	-11.84%	620	-6.93%	2.95	-4.33%	1.55
bigblue3	5.024	4.038	-19.63%	1473	-7.90%	2.99	-8.68%	1.43
bigblue4	10.535	9.230	-12.39%	2273	-5.05%	5.23	-4.46%	1.73
			-14.46%		**-6.40%**	**3.05**	**-5.34%**	**2.29**

TABLE VI

COMPARISON OF DETAILED PLACERS ON CAPO9.1 GLOBAL PLACEMENT ON ISPD05 BENCHMARKS

	CAPO	Our			Fengshui5.0		RowIroning	
	WL(1e8)	WL(1e8)	Impv	runtime	Impv	runtime/Our	Impv	runtime/Our
adaptec1	0.918	0.906	-1.26%	112	0.56%	2.89	-0.78%	3.44
adaptec2	1.027	1.010	-1.61%	171	0.80%	2.30	-0.80%	2.57
adaptec3	2.538	2.509	-1.16%	399	0.37%	2.93	-0.62%	1.97
adaptec4	2.654	2.637	-0.65%	410	0.51%	3.22	-0.60%	2.06
bigblue1	1.167	1.135	-2.72%	214	0.65%	2.27	-0.88%	2.43
bigblue2	1.813	1.783	-1.68%	985	0.07%	1.95	-1.01%	0.94
bigblue3	4.444	4.270	-3.92%	1051	-1.70%	4.11	-2.11%	1.84
			-1.86%		**0.18%**	**2.81**	**-0.97%**	**2.18**

global placements are better than that obtained on Capo global placements. The reason may be because the legalizer disturbs the global placements of *mPL5* by a significant amount, but most of these errors can be fixed by the detailed placer.

V. CONCLUSIONS

In this paper, we present an efficient and effective detailed placement algorithm. It consists of a Global Swap technique to swap cells based on their optimal regions, a Vertical Swap technique to fix local errors vertically, a fast Local Re-ordering technique to repair the local order horizontally, and a Single-Segment Clustering technique that can optimally place the cells in a segment when cells in all other segments are fixed. We also give the flow to apply these four techniques. Experiments on two sets of benchmarks show that our detailed placer can achieve better solution quality in much shorter time compared to *Fengshui 5.0*, *rowIroning* and *Domino*.

The algorithm presented in this paper is a wirelength-driven placement algorithm. Other objectives such as timing-driven placement and congestion-driven placement should be considered in the physical synthesis flow. Our future work will focus on efficient algorithms considering these objectives.

ACKNOWLEDGMENT

The authors would like to thank Gi-Joon Nam and Charles Alpert of IBM for providing the ISPD05 benchmarks; Igor Markov and Jarrod Roy for the Capo placement tool; Jason Cong and Kenton Sze for the mPL placement tool; Frank Johannes and Bernd Obermeier for the Domino placer.

REFERENCES

[1] ISPD04 IBM Standard Cell Benchmarks with Pads. *http://www.public.iastate.edu/~ nataraj/ISPD04_Bench.html*.

[2] S. N. Adya, and I. L. Markov. Consistent Placement of Macro-Blocks using Floorplanning and Standard-Cell Placement. In *Proc. Intl. Symp. on Physical Design*, pp. 12-17, 2002.

[3] A. Agnihotri et al. Fractional Cut: Improved Recursive Bisection Placement. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 307, 2003.

[4] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Can recursive bisection produce routable placements? In *Proc. ACM/IEEE Design Automation Conf.*, pp. 477-482, 2000.

[5] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Optimal Partitioners and End-Case Placers For Standard-Cell Layout. In *IEEE Trans. on Computer-Aided Design, vol. 19*, pp. 1304-13, 2000.

[6] T. Chan, J. Cong, T. Kong, and J. Shinnerl. Multilevel optimization for large-scale circuit placement. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 171-176, 2000.

[7] T. Chan, J. Cong, K. Sze. Multilevel generalized force-directed method for circuit placement. In *Proc. Intl. Symp. on Physical Design*, pp. 185-192, 2005.

[8] K. Doll, F. M. Johannes, and K. J. Antreich. Iterative Placement Improvement by Network Flow Methods. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 13(10):1189-1200, 1994.

[9] H. Eisenmann and F. Johannes. Generic global placement and floorplanning. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 269-274, 1998.

[10] S. Goto. An Efficient Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Layout. In *IEEE Transitions on Circuits and Systems, Vol. CAS-28, No. 1*, pp. 12-18, 1981.

[11] B. Hu and M. Marek-Sadowska. FAR: Fixed-points addition and relaxation based placement. In *Proc. Intl. Symp. on Physical Design*, pp. 161-166, 2002.

[12] S.-W. Hur and J. Lillis. Relaxation and Clustering in a Local Search Framework: Application to Linear Placement. In *Proc. ACM/IEEE Design Automation Conf.*, pp.. 360-366, 1999.

[13] S.-W. Hur and J. Lillis. Mongrel: Hybrid Techniques for Standard Cell Placement. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 165-170, 2000.

[14] S.-W. Hur, et al. Force directed Mongrel with physical net constraints. In *Proc. ACM/IEEE Design Automation Conf.*, pp.. 214-219, 2003.

[15] A. B. Kahng, I. L. Markov and S. Reda. On Legalization of RowBased Placements. In *Proc. Great Lakes Symp. on VLSI*, pp. 214-219, 2004.

[16] A. B. Kahng and Q. Wang. Implementation and extensibility of an analytical placer. In *Proc. Intl. Symp. on Physical Design*, pp. 18-25, 2004.

[17] A. B. Kahng, P. Tucker and A. Zelikovsky. Optimization of Linear Placements for Wirelength Minimization with Free Sites. In *in Asia and South Pacific Design Autom. Conf.*, pp. 241-244, 1999.

[18] J. Kleinhans, G. Sigl, F. Johannes, and K. Antreich. Gordian: VLSI placement by quadratic programming and slicing optimization. *IEEE Trans. Computer-Aided Design*, 10(3):356-365, 1991.

[19] Gi-Joon Nam, Charles J. Alpert, Paul Villarrubia, Bruce Winter and Mehmet Yildiz. The ISPD2005 placement contest and benchmark suite. In *Proc. Intl. Symp. on Physical Design*, pp. 216-220, 2005.

[20] C. Sechen and A. L. S. Vincentelli. Timberwolf 3.2: A new standard cell placement and global routing package. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 432-439, 1986.

[21] R. Varadarajan. Convergence of placement technology in physical synthesis: Is placement really a point tool? In *Proc. Intl. Symp. on Physical Design*, page 7, 2003.

[22] N. Viswanathan, and Chris C. N. Chu. FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model. In *Proc. Intl. Symp. on Physical Design*, pp. 26-33, 2004.

[23] N. Viswanathan, and Chris C. N. Chu. FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model. *IEEE Trans. Computer-Aided Design*, 24(5):722-733, 2005.

[24] M. Wang, X. Yang, and M. Sarrafzadeh. Dragon2000: Standard-cell placement tool for large industry circuits. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 260-263, 2000.

[25] Z. Xiu, J. D. Ma, S. M. Fowler, and R. A. Rutenbar. Large-Scale Placement by Grid-Warping. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 351-356, 2004.

[26] M. C. Yildiz and P. H. Madden. Global objectives for standard cell placement. In *Proc. 11th Great Lakes Symposium on VLSI*, pp. 68-72, 2001.