Fine Granularity Clustering for Large Scale Placement Problems

Bo Hu
Department of Electrical & Computer Engineering
Univ. of California, Santa Barbara, CA 93106, USA
1-805-893-5678
hb@ece.ucsb.edu

Malgorzata Marek-Sadowska
Department of Electrical & Computer Engineering
Univ. of California, Santa Barbara, CA 93106, USA
1-805-893-2721
mms@ece.ucsb.edu

Abstract

In this paper we present a linear-time Fine Granularity Clustering (FGC) algorithm to reduce the size of large scale placement problems. FGC absorbs as many nets as possible into Fine Clusters. The absorbed nets are expected to be short in any good placement; therefore the clustering process does not affect the quality of results. We compare FGC with a connectivity-based clustering algorithm proposed in [1] and simulated-annealing-based algorithm in TimberWolf [2], both of which also reduce the number of external nets between clusters. The experimental results show that our algorithm achieves better net absorption than the previous approaches while using much less CPU time for large scale problems. With our FGC algorithm, we propose a Fast Placer Implementation (FPI) framework, which combines our FGC-based size reduction with traditional placement techniques to handle large-scale placement problems. We compared FPI placement results with a public-domain fast standard cell placer Capo[4] on large scale benchmarks. The results show that FPI can reduce CPU time for large scale placement by a factor of 3~5x while obtaining placement results of comparable or better quality.

Categories and Subject Descriptors

J.6 Computer-Aided Engineering - Computer-aided Design (CAD)

General Terms

Algorithms.

Keywords

Placement, Clustering.

1. Introduction

The rapid advance of modern VLSI technology makes possible a billion-transistor integration in a single chip. To handle such design complexity, a hierarchical approach, which has already been well accepted in tool development of Electric Design Automation (EDA) industry, is deemed essential, especially for the success of future nanometer-scale chip design. It is a tradition that standard cell placement exploits the merits of hierarchical approach for better quality and fast CPU time. But as for the future billions-of-transistors chip, it is still a tremendous challenge for a modern standard cell placer to deliver a good-quality placement in terms of performance, timing, power, and congestion in a reasonable time, especially when the time-to-market requirement is stringent.

Due to the deep sub-micron effects like crosstalk, IR drop, etc., the timing is not known until the physical design is finished. The reliance on physical data to evaluate whether the design meets higher level specification requires that the physical design tool be able to provide a good solution as fast as possible. Especially because of the uncertainty caused by those deep sub-micron effects, the chip design may iterate several times between logic synthesis and physical design. These iterations also require efficient physical layout tool implementation, particularly in the case of large designs.

Among various techniques to improve placement quality and decrease the CPU time, clustering is one of the most frequently researched, having attracted academic and industrial attention consistently for years. A number of clustering techniques [1], [2], [5], [6], [7], [8], [9], [10], [11], [12], [14], [15] etc. have been proposed in the past decades. Most of these techniques share common characteristics - they proceed bottom-up and are connectivity-oriented. It is also true that most of the previous works apply clustering techniques mainly to improve partitioning.

In [15], the authors presented a clustering technique which absorbs small-fanout nets into clusters in order to minimize the inter-cluster communication in hierarchical FPGAs. Their results show remarkably improved routability. For ASIC designs, Timberwolf [2] in its hierarchical placement mode, also absorbs into the clusters as many small-fanout nets as possible. It applies the reasoning that the bounding boxes (or wire lengths) of small-fanout nets are much easier to reduce than those of large-fanout nets. The nets are modeled as trees. Each net of degree *k* introduces *k - 1* edges whose weights are set to *1/(k-1)*. If there are *m* nodes of a net *e* in one cluster *c*, then the weight contributed by the net *e* to the cluster *c* is *(m-1)/(k-1)*. In [2] the authors proposed to maximize the total weight of all clusters, given that the weight of each cluster is the sum of the contributions from all nets incident to this cluster. In [1], the authors devised a greedy connectivity-based algorithm to merge strongly connected nodes. They used the formula in EQ1 to compute the connectivity between nodes *i* and *j*:

$$c_{ij} = \frac{bandwidth_{ij}}{A_i \cdot A_j \cdot (fanout_i - bandwidth_{ij}) \cdot (fanout_j - bandwidth_{ij})} \quad \text{(EQ1)}$$

Each net of degree *k* connecting *i* and *j* contributes *1/(k-1)* to $bandwidth_{ij}$. Similarly, each net of degree *k* incident to node *i* contributes *1/(k-1)* to $fanout_i$. A_i and A_j are sizes of nodes *i* and *j,* respectively. It can be seen that this formula also captures the small-fanout net absorption because those nets contribute a lot to $bandwidth_{ij}$ by the weighting scheme *1/(k-1)*. Based on connectivity computation, [1] visited all the nodes in a random order and picked the neighbor with maximum connectivity for each un-clustered node.

Motivated by these works which choose net absorption as a way to either optimize wire length or reduce inter-cluster communication, we focus on net absorption problem in the context of size reduction for large-scale placement problems. We hope to substantially reduce the time cost for large-scale placement problems while maintaining or improving the placement quality. To achieve this, we propose to extract Fine Cluster (small cluster containing only several nodes) netlist such that we need only to apply a global placement optimization on it without compromising the solution quality.

In this paper, we present a linear-time Fine Granularity Clustering (FGC) algorithm to reduce the size of large-scale placement problems. Fine Cluster typically contains 2~6 standard cells of average size and has a strict upper and a lower size bound. One pass of our algorithm consists of two phases. In phase I, we adapt the traditional Fidducia-Matheysys (FM) heuristic (FM) to partition a netlist composed of thousands of clusters. In phase II, we apply the Primitive Cluster Movement (PCM) to escape from the local minimum achieved by phase I. Primitive Clusters are pre-characterized to help our net absorption objective, and are finer clusters including only 2 or 3 cells. Unlike traditional clusters in multi-level partitioner, Primitive Clusters are dynamically recognized in each PCM pass, and different Primitive Clusters can overlap each other. We apply several heuristics to expand the set of Primitive Clusters. As for net absorption, we compare FGC results with simulated annealing based technique in Timberwolf [2] and the connectivity-based greedy algorithm in [1] using large scale standard cell benchmarks. The comparisons show that our technique achieves a better net absorption while consuming much less CPU time. With our FGC algorithm, we propose a Fast Placer Implementation (FPI) framework, which combines our FGC-based size reduction with traditional placement techniques to handle efficiently large-scale placement problems. We embed a public-domain standard cell placer Capo[4] into our FPI framework and compare FPI placement results with the original Capo on large-scale benchmarks. The results show that FPI framework can reduce the CPU time for such problems by a factor 3~5x while obtaining placement results of comparable or better quality. Our examples indicate that a proper placer framework can dramatically boost the performance of an existing placer.

The paper is organized as follows. In section 2 we introduce the terminology. In section 3 we formulate the FGC problem. In section 4, we present the two phases of the FGC algorithm: the Adapted FM and the Primitive Cluster Movement. Experimental results are given in section 5, followed by conclusions in section 6.

2. Terminology

In this section, we first briefly review the terminology used extensively in the literature on clustering, and then we introduce the terms used throughout this paper.

A hypergraph *G(V, E)* is a graph with a node set *V* and a hyperedge set *E*. Each hyperedge *e* in *E* is a subset of nodes in *V*. We denote this subset as *N(e)*. A node *v* is incident to a hyperedge *e* if *v* is in *N(e)*. Similarly, a net *e* is said to be incident to a node *v* if *N(e)* contains *v*. The degree of a node, denoted by *D(v)*, is the number of hyperedges incident to *v*. The degree of a hyperedge *e*, denoted by *D(e)*, is the number of nodes incident to *e*. Each hyperedge is assigned a weight, denoted by *w(e)*. In addition, each node is associated with an area cost, denoted by *A(v)*.

A cluster *C* is defined as a set of nodes. A hyperedge *e* is absorbed into a cluster *C* if all the nodes in *N(e)* are included in *C*. A clustering solution *S* is a set of clusters, such that each node in *V* appears in one and only one cluster in *S*. The hyperedge which is absorbed into a cluster is called an internal hyperedge with respect to *S*; otherwise, it's called an external hyperedge.

A special hypergraph *G(V, E)* whose each hyperedge connects only two nodes is called a Simple Hypergraph. We denote this type of hypergraph as *SG(V, E)*. In a Simple Hypergraph, a hyperedge is called a connection. The nodes incident to a connection *c* are terminals with respect to *c*. As in a regular hypergraph, the degree of a node *v* is the number of connections incident to *v*.

There is a one-to-one correspondence between a hypergraph and a standard cell netlist. The cell in the netlist corresponds to the node and the net corresponds to the hyperedge in the hyper-graph. In this work, we use the terminology of cell and node, and net and hyperedge, without distinguishing one from the other.

3. Fine Granularity Clustering (FGC) Problem Formulation

In this section, we first discuss the cost function we use in the FGC algorithm, and then give the formulation of the FGC problem.

In this work, we model the hyperedges with degrees larger than 2 as cliques. In this way, a given regular hypergraph can be reduced to a simple hypergraph by replacing each hyperedge with a clique of connections. The weight of each connection is given by EQ2:

$$w(c) = \frac{w(e)}{(D(e)-1)D(e)} \qquad \text{(EQ2)}$$

In the equation above, *D(e)* is the degree of the hyperedge *e*, and *w(e)* is the weight that can be used to control the absorption priority. We choose the clique model instead of a tree model of Timberwolf [2] for the following two reasons. First, the clique model can distinguish more topologies and favors the "nearly" complete absorptions. As shown in Figure 1, suppose the 4 black circles are nodes incident to a hyperedge of degree 4. The tree model in Timberwolf has the same cost for the two topologies in the figure, while the clique model favors the second one over the first. It is obvious that the second topology has a better chance to absorb this hyperedge completely. Second, by using the clique model, a regular hyper-graph can be reduced to a simple hyper-graph, which makes our FGC algorithm much more efficient.

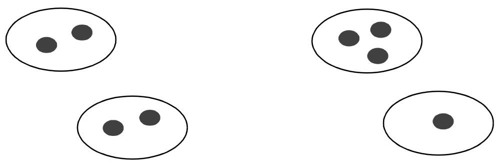

Fig. 1: comparison between tree model and clique model

Before discussing the FGC cost function, we define our terminology.

Definition 1: A Fine Cluster is a small cluster with upper and lower size bounds (U, L). In general, (U, L) are defined such that the Fine Cluster contains only several nodes of an average size in the hyper-graph. We denote a Fine Cluster *f* with the bound constraints (U, L) as *f*(U,L), and a set of Fine Clusters with bound constraints (U, L) as *F(U, L)*. A clustering solution *S*, which consists of only Fine Clusters, is called the Fine Cluster Solution (*FCS*). Like previous clustering algorithms, Fine Clusters in *FCS* cannot overlap with each other.

Definition 2: The cost of each Fine Cluster *f,* denoted by *C(f)*, is a summation of the weights *w(c)* for all connections which are completely absorbed into *f* as stated in EQ3.

$$C(f) = \sum_{c \in f} w(c) \qquad \text{(EQ3)}$$

Definition 3: The cost of a Fine Cluster Solution *FCS*, denoted by *C(FCS)*, is the sum of the costs of all Fine Clusters in *FCS* as expressed by EQ4.

$$C(FCS) = \sum_{f} C(f) \qquad \text{(EQ4)}$$

Based on the definitions of Fine Clusters and their costs, we formulate the Fine Granularity Clustering (FGC) problem.

FGC formulation: Given a simple hypergraph *SG(V, E)*, find a Fine Cluster Solution *FCS*, which consists of a set of Fine Clusters *F(U,L)* and *C(FCS)* is maximized.

By maximizing *C(FGC)*, we can push as many heavy weight connections as possible into Fine Clusters. If the original hypergraph consists of only hyperedges of degree 2, maximizing EQ4 is equivalent to maximizing the number of internal hyper-edges. Our FGC formulation is similar to that of [2] except that (1) we choose a clique model instead of a tree model for the multi-fanout nets; (2) FGC targets Fine Clusters, which are on the average much smaller than those in [2].

There are several reasons that we do not make a Fine Cluster larger:

(1) Our objective is to reduce the placement problem size. Fine Clusters actually behave more like hard constraints once we make a decision to create them. After clustering, the placer searches only a global solution space of the clustered netlist. When the global placement is completed, the refinement for Fine Clusters is limited to a local region. So it is not desirable to form bigger clusters because the global solution quality might be compromised. Traditionally, in multi-level clustering for partitioning purpose, bad clustering decisions are more tolerable because they can, to some extent, be recovered at the cost of CPU time as partitioning proceeds. In contrast, including nodes into a Fine Cluster implies that we are going to place these nodes in close proximity. In this way we reduce the placement problem size and thus boost the placer performance. To achieve the same or even better placement quality, Fine Clusters have to be carefully generated.

(2) By using Fine Clusters, the design cost such as total wire length can be estimated more accurately. Since Fine Clusters contain only several nodes of average size, we are able to get a reasonable estimate even when we assume that all the nodes in the cluster overlap with each other.

(3) Since Fine Clusters consist of only a few nodes of average size, a good quality clustering solution can be possibly achieved by exploring only the local structure of the netlist.

4. Fine Granularity Clustering Algorithm

In this section, we describe the Fine Granularity Clustering algorithm (FGC). In sub-section 4.1 we explain the algorithm in general terms. In sub-section 4.2 the overall algorithm is described. Details of the algorithm are explained in the subsequent sub-sections.

4.1 FGC Algorithm Overview

The main loop of our FGC algorithm consists of two phases. The phase I is an adoption of the classical FM [3] gain update strategy for thousands of clusters. Since the gain is usually a real number, we have to discretize it in order to build a gain bucket. We devise an efficient data structure to make a gain update very quickly. In phase II, we perform Primitive Cluster Movement (PCM). Primitive Clusters are pre-characterized before the PCM algorithm begins and can be exchanged between Fine Clusters. Figure 2 gives the pseudo code of FGC.

```
FGC()
Input: Hypergraph SG(V, E), Constraints (U, L)
Output: a Fine Cluster Solution FCS
 {  N = 0;
    PreCharacterizePrimitiveClusters(SG(V, E));
    InitializeFineClusters(SG(V, E));
    while(improvement & N < max_iterations) {
        PhaseI:AdaptedFM();
        PhaseII:PrimitiveClusterMovement();
        N++;
    }
   Output solution FCS;
 }
```

Fig. 2: The FGC algorithm

4.2 Initial Cluster Generation

We use a greedy algorithm to generate the initial set of Fine Clusters. Initially, each un-clustered node forms a trivial one-node cluster. We maintain an array of such trivial clusters. Then we visit trivial clusters in a random order. We select one trivial cluster and check to see whether it has been already included into some Fine Cluster. If so, we continue to visit the next trivial cluster; otherwise, we make the current trivial cluster a seed of a new Fine Cluster and start attracting the un-clustered neighbors into the newly created Fine Cluster, until the size constraint (U + L)/2 is reached. The neighbors are attracted in a greedy fashion such that the one with the maximum connection to the Fine Cluster is selected first.

This process is order-dependent and has a very limited grasp of the local structure. For example, (1) when a Fine Cluster is created, it has no ability to judge whether it is worthwhile to attract a new node into itself. (2) Once an un-clustered node is attracted into some Fine Cluster, it is not allowed to move again.

4.3 FGC algorithm Phase I: the Adapted Fiduccia-Mattheyses Heuristic (AFM)

In this section, we describe the adaptation of the classical FM heuristic[3] for our FGC problem.

For each node in a Fine Cluster, we maintain an array of move destinations with the corresponding gains. Move destimations of a node are those Fine Clusters which have direct connections with this node. The gain is a difference of cost function if we move the node to a destination. In general, the gain is not an integer number. We have to discretize the gains so that a gain bucket can be built. A gain bucket is a set of gain entries. Each entry represents a range of gain (lg, ug). Given a gain upper bound UG, a lower bound LG and the number of entries in the bucket N, we build the gain bucket by evenly dividing the full gain range (LG, UG), and we index the gain entries from 0 to N - 1. Gain unit is defined as the difference between the upper limit ug and lower limit lg of a gain range. With gain unit, the bucket entry can be obtained in a constant time for any gain within the range (LG, UG).

After we compute the gains for each node, we choose the best move for each node to initialize the gain bucket.

To efficiently update the gain bucket after each move, we devise a data structure as shown in Figure 3. Each node is associated with a

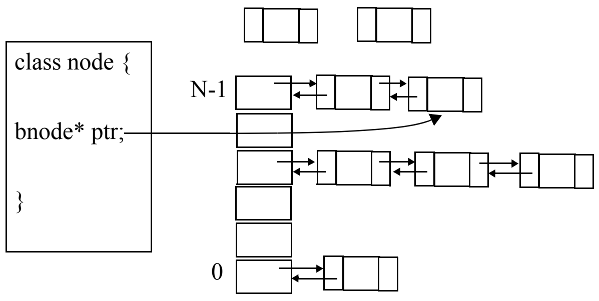

Fig. 3: Data structure in Adapted FM algorithm

bnode data structure which is double-linked in the bucket entries. Double pointers ensure that move insertion and removal will require only a couple of pointer updates. Furthermore, we allocate bnode data structure only once throughout FGC algorithm and thus avoid overhead of memory allocation and release.

If a move is taken, we have to update the gains for all the affected nodes (in this case, all neighbors of the moved node). The neighbors of a node *v* are those nodes directly connected to *v*. We follow the similar update strategy used in FM heuristic. In the following, we briefly describe this strategy with our terminology.

Definition 4: A connection is called Visible when the two terminal nodes are in the different clusters; otherwise, it is Invisible.

Definition 5: Visibility Transition is the change of a connection *c*'s visibility due to a move involving one of *c*'s terminal nodes. If a node *v* is moved out of its current cluster, the connections incident to v may have the following Visibility Transitions:

(1) (I, V): from Invisible to Visible;

(2) (V, V): Visibility doesn't change;

(3) (V, I): from Visible to Invisible.

It is clear that Visibility Transition corresponds exactly to the gain of a move. (I, V) transition occurs when an invisible connection is exposed by the move, thus incurring a negative gain. Similarly, (V, I) transition hides one connection and achieves positive gain. The gain is zero if (V, V) transition occurs. The gain of a move can be computed by looking at the Visibility Transitions of all connections incident to the moved node. Similarly, the update of a gain can be obtained by checking if there are any changes of Visibility Transitions for some connections.

To update a neighbor's *b* gains caused by a move of *v*, it is sufficient to look at Visibility Transition of the connection between *b* and *v*, because it is the only connection which might have visibility change due to the movement of *v*. Figure 4 gives a code segment which computes the Visibility Transition for one affected neighbor of a moved node. In Fig 4, Parent(*b*) and Parent(*v*) are the initial

```
if (Parent(b)!= Parent(v))
  if (DC(b)!= Parent(v))
    VTbefore = (V,V);
  else
    VTbefore = (V, I);
else
  VTbefore = (I, V);

if (Parent(b)!= DC(v))
  if (DC(b)!= DC(v))
    VTafter = (V, V);
  else
    VTafter = (V, I);
else
  VTafter = (I, V);
lookupGainChange(b, DC(b), VTbefore, VTafter);
```

Fig. 4: gain update for a neighbor b after move v to DC(v)

clusters *b* and *v* belong to. DC(*b)* and DC(*v)* denote the destination cluster of *b* and *v.* By looking at the Visibility Transition before moving *v* to DC(*v)*, VTbefore; and Visibility Transition after the move, VTafter; we can compute the gain update for moving *b* from its current cluster Parent(*b)* to DC(*b*).

As shown in Fig. 4, it takes only 4 comparisons to determine the Visibility Transition. The new gain can be found from a table-look-up in a constant time. Since the number of neighbors is bounded for any node in the hypergraph, especially in our case, we ignore all the hyperedges with degrees exceeding a threshold; therefore, the gain update for a move takes constant time O(1).

Each AFM pass visits all the nodes following their positions in the gain bucket. The move with the highest gain is popped out of the gain bucket for a feasibility check. If size constraint is not violated by this move, the move is taken and the gains are updated for all the affected neighbors. The pass ends when the bucket is empty. The best solution during the movement is recorded and will be used as the starting point for the next pass. The time for each AFM pass is O(n), where n is a number of nodes in the hypergraph.

4.4 FGC algorithm Phase II - Primitive Cluster Movement (PCM)

Motivated by the successful application of multi-level clustering in traditional partitioning algorithms, we introduce a higher level representation of the initial hypergraph. To this end we introduce Primitive Clusters defined as follows:

Definition 6: Primitive Cluster (PC) is the finest cluster which consists of only 2 or 3 nodes in the original hypergraph. A Primitive Cluster consisting of nodes a and b is denoted by pc{a, b}.

From the definition, it is clear that a Primitive Cluster is even smaller than a Fine Cluster. In fact, a Primitive Cluster can be viewed as another type of movable unit other than the nodes in the graph. These additional types of node exchange among the clusters increase the solution space and allow to obtain a better quality clustering result. Compared to the original nodes in the hypergraph, Primitive Cluster is powerful in that it makes it possible to move more than one node simultaneously, possibly helping the clustering to escape from a local minimum achieved by node-only movement.

It is worthwhile to show the differences between our Primitive Cluster Movement and the clustering in traditional multi-level partitioning where clusters are exchanged between partitions.

First, the Primitive Clusters are dynamically collected in each PCM pass whenever they are available for movement. Since we interleave the PCM and the AFM phases, two consecutive PCM passes could have a completely different set of Primitive Clusters. This rich set of Primitive Clusters enables us to explore a larger solution space. In contrast, in traditional partitioning, clustering tree is usually generated in the first place and remains unchanged as the partitioning proceeds.

Second, Primitive Clusters do not need to be disjoint; instead, different Primitive Clusters can overlap with each other. For example, Primitive Cluster pc{a, b} and pc{b, c} can exist simultaneously. As shown in Fig 5a, suppose we first decide to move the Primitive Cluster pc{a, b} from the Fine Cluster A to B according to cost computation. After the nodes a and b settle down in B, we may realize that b and c form another Primitive Cluster pc{b, c}, and then we may continue to move pc{b, c} to the Fine Cluster C as indicated in Fig 5b, if the movement is justified by the cost. In other words, allowing overlapping among Primitive Clusters enables us to enjoy more flexibility of moving nodes around and thus to achieve a better solution quality. In contrast, at each level of a traditional multi-level clustering tree, all clusters are disjoint, and no sharing among them is allowed.

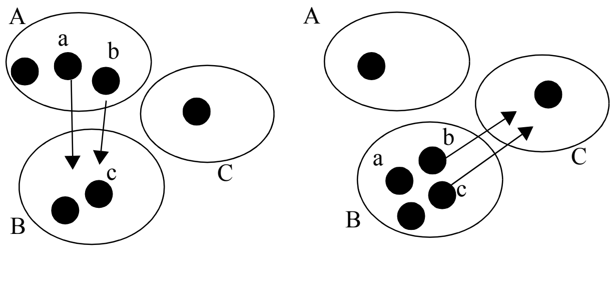

Fig. 5a: move pc{a,b} to B Fig. 5b: move pc{b,c} to C

Third, the Primitive Cluster is an abstract coarsening level of an initial hypergraph. Because it is an abstract level, we avoid the cost overhead of building the coarser level, as is done in traditional clustering hierarchy generation.

The Primitive Clusters are pre-characterized based on the initial regular hypergraph before reduction. We employ the following heuristics to create good Primitive Clusters.

(1) Heavy connection: it is straightforward to create Primitive Clusters from those connections with large weight. Such connections are called Heavy Connections. Hiding heavy connections inside clusters leads directly to the maximization of our cost function in EQ4.

(2) Hyperedge of degree 2: by putting two nodes of a degree 2 hyperedge into a Primitive Cluster, we instantly absorb one hyperedge.

(3) Hyperedge of degree 3: from a hyperedge e of degree 3, we can create 3 2-node Primitive Clusters by enumerating any two-terminal nodes of e and a three-node Primitive Cluster by including all its three terminal nodes into a cluster.

Primitive Cluster Movement (PCM) phase is interleaved with phase I (Adapted FM). Each PCM can have several passes. During each PCM pass, we randomly pick a pre-characterized Primitive Cluster and check to see whether it is included in some Fine Cluster. If so, the best destination Fine Cluster is determined for the selected Primitive. The best destination Fine Cluster are selected from those clusters which have direct connections with the Primitive. If there is a positive gain associated with the best destination cluster, the Primitive Cluster Movement will be taken. The approach is very greedy, but since it is interleaved with AFM, in most cases, it can help in escaping from the local minimum achieved by the previous AFM phase. In PCM phase, the number of passes is usually small. In the experiments, we observed that 3 ~ 5 passes per PCM phase are enough to achieve good results.

4.5 Complexity Analysis

The movement for one Primitive Cluster is very fast and needs to go through all connections incident to it only once. Since we ignore the hyperedges with very large degree (>25) (traditionally, total wire length minimization always focus on small-fanout nets), the number of the connections incident to a node or a Primitive Cluster is actually bounded by a small number, espeically when the node is standard cell instance and Primitive Cluster includes only 2 or 3 nodes. Thus the time required for one Primitive Cluster Movement is O(1). In the experiments, the size of the Primitive Cluster set we create is of the order O(n), where n is a number of nodes in the initial hypergraph. Therefore one PCM pass is of O(n) time complexity.

As described in the previous sections, the complexity of the Adapted FM pass is also O(n). Since we limit the number of AFM and PCM passes in FGC algorithm by a constant, the overall FGC is of linear time complexity with respect to the size of a hypergraph or the standard cell circuit.

5. Experiments

We implemented our FGC algorithm in C++ and conducted the experiments on 1Ghz Pentium 4 linux machine with 1gigabyte of memory. For experiments we used the MCNC benchmarks and the IBM placement benchmarks available at [17]. For all benchmarks, we eliminate the channels between standard cell rows and perform the placement in fixed-die mode. The statistics of the circuits are listed in the 2nd and 3rd columns in Table 1. CPU times are given in seconds.

5.1 Net Absorption Comparison with Connectivity based Algorithm and TimberWolf

We first compare the net absorption results of FGC algorithm with the connectivity based [1] and Timberwolf [2] algorithms. We use these 3 algorithms to generate the same number of Fine Clusters with the same lower and upper size bounds. The bounds are set such that each Fine Cluster can have 2~6 cells of average size. For each benchmark, after clustering has been performed, we collect the number of the nets which were totally absorbed into the Fine Clusters. Since our Fine Clusters are very small, the majority of the absorbed nets are of degree 2 and 3.

In table 1, we denote connectivity based algorithm and Timberwolf as CON and TW respectively. The net absorption results are normalized with respect to that of CON and listed in columns 4, 5 and 6. It can be seen that FGC and TW can absorb many more small-fanout nets than the greedy algorithm CON. We believe it is because CON makes hard decisions in the clustering process, and once two nodes are combined, they are never to be separated. In contrast, FGC and TW both maximize a global cost function and

bench	#nodes	#nets	#Net Absorption			clustering CPU(s)			total wire length (m)		
			CON	TW	FGC	CON	TW	FGC	CON	TW	FGC
biomed	6417	5742	2695	3571	3619	1.37	22	4.2	3.15	2.71	2.68
industry2	12142	13419	4045	5234	5818	3.5	53.1	9.1	12.2	12.1	11.0
avqsmall	21854	22124	9099	10028	11087	10.4	397	21.8	4.94	4.73	4.56
avqlarge	25114	25384	10838	12410	13255	11.2	500	22.4	5.52	5.20	4.89
ibm01	12282	11507	3962	4814	5303	3.1	43	10.1	67.4	62.5	60.6
ibm02	19321	18429	5211	8216	8333	6.2	115	20.1	184	164	169
ibm03	22207	21621	8083	10395	10536	6.5	200	20	12.1	10.5	10.3
ibm04	26633	26163	8208	10985	11428	8.2	257	22	14.6	14.1	13.8
ibm05	29347	28446	11219	13363	14511	13.2	310	47	41.2	40.1	38.7
ibm06	32185	33354	10551	14853	15373	15.7	315	43	17.6	16.4	16.3
ibm07	45135	44394	16395	19498	20297	18	496	50	423	400	392
			100%	**125%**	**132%**	**4.5%**	**100%**	**11.7%**	**100%**	**92.8%**	**90.1%**

TABLE 1. Net Absorption and total wire length comparison between [1], [2], and FGC algorithm

thus explore a much larger solution space. Compared to TW, FGC achieves better net absorption results in much less CPU time. One reason is that, although simulated annealing technique used in TW is able to find a global optimum solution, it requires substantial computation time. We also observed that FGC is particularly effective to handle small clusters like Fine Clusters. For clusters of this size, it is possible to rely only on local structure to achieve good net absorption.

Clustering CPU times are given in columns 7, 8, and 9, normalized with respect to TW, in Table 1. As analyzed before, FGC is of linear time complexity with respect to the size of a benchmark, and it is much faster than TW. In general, FGC only takes a tenth of the time consumed by TW under the current annealing schedule. We have also tried to improve the solution quality of TW by spending more time on iterations, but the gain is small even when the CPU time doubles.

The results indicate that FGC algorithm is very powerful in handling Fine Clusters and can absorb a substantial number of small-fanout nets into clusters, thus the inter-cluster communications are dramatically reduced.

5.2 Placement Evaluation

To show the effect of net absorption on placement, we used the public-domain standard cell placer Capo[4] and performed a global placement on Fine Clustered netlist generated by CON, TW and FGC. The global placement was conducted in fixed-die mode and followed by a simple overlap removal procedure that detail-placed cells inside the Fine Clusters. In the last three columns of Table 1 we list the placement results for all three algorithms.

Generally, FGC achieves a better total wire length result than CON and TW, because it hides more small-fanout nets inside the Fine Clusters. This behavior corresponds to that of a good placer. CON gives the worst result on net absorption due to its greedy nature. Similarly, the placement quality in terms of wire length by CON is worse than both FGC and TW. The reason is that, as described in the earlier section, we treat Fine Clusters more like hard constraints than we would in a traditional clustering process. Wrong decisions about these hard constraints will compromise the global placer's search for a good solution. As observed in [16], the lower level routing layers like M1 and M2 are usually not congested. Thus, absorbing more small-fanout nets into Fine Clusters is likely not to cause congestion problem at M1 and M2 layers. At the same time, since the number of inter-cluster (global) nets has been reduced, it is expected that the congestion at top routing layers will be improved.

5.3 Placement Improvement by Fast Placer Implementation (FPI)

In this experiment, we investigate the feasibility of net absorption-based size reduction for large-scale placement problems. By feasibility we mean if the quality of placement can be maintained while CPU time is being dramatically reduced. Especially, we propose the following hierarchical framework of a Fast Placer Implementation (FPI) as shown in figure 5.

In figure 5, our Fast Placer Implementation (FPI) consists of 3 stages. In Stage 1, we reduce the size of initial large scale netlist by the Fine Granularity Clustering (FGC) procedure. The motivation is that time-consuming global optimization might not need to go all the way down to the initial netlist. Based on the local structure of the netlist and the behavior of a good placer, it is possible to apply global optimization only on a Fine Cluster netlist. In stage 2, global placement optimization can be applied on the reduced netlist. Any existing global placement method like that of [4], [18] can be used at this stage. Since placement is not a linear-time procedure, size reduction by only 2x at the previous stage will produce more than 2x global optimization speedup. After stage 2, we de-cluster Fine Clusters and apply local refinement. It can be seen that Stage 1 is crucial in FPI framework because, if many bad decisions are made at this stage, either they cannot be recovered by local refinement or extensive effort is required, which might overwhelm the speedup achieved at Stage 2.

To experiment with our FPI, we choose FGC at Stage 1. For Stage 2, we use public-domain fixed-die placer Capo[4]. Capo placement is based on recursive partitioning and falls into the category of polynomial-time placers. Thus it is expected that by embedding Capo into our FPI framework, the time complexity for large scale placement can be greatly reduced. At the last stage, we apply a fast low-temperature annealing procedure and placement legalization to remove overlapping. Stage 2 of the FPI, that is the Fine Cluster Placement, still dominates the placement time for large-scale netlists.

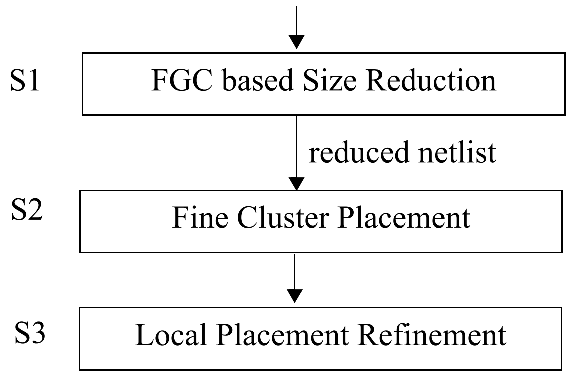

Fig. 5: Fast Placer Implementation (FPI) framework

In the experiment, we compare the total wire lengths and CPU time between Capo[4] and our FPI framework. We use Capo fast-mode by setting branching factor to be 1. Several large-scale benchmarks from IBM-place suits are selected to evaluate the performance of FPI framework. The number of cells for those benchmarks ranges from 50,977 to 182,137. Table 2 gives the CPU time and total wire length (Bounding Box) experimental results from Capo and FPI. Wire lengths are given in meters, and CPU times are reported in seconds. The last row in Table 2 normalizes the data with respect to Capo.

From table 2, it can be seen that CPU time is reduced by a factor of 3~5x while total wire lengths are comparable to or slightly better than the results of Capo. Furthermore, as the sizes of the benchmarks increase, more speedup is achieved. It is actually not surprising because of the polynomial time-complexity of the placement. These results suggest that it is possible to dramatically speed up the placement of standard cell netlists by applying the time-consuming global placement optimization only on Fine Cluster netlists. By carefully generating such Fine Cluster netlists, the placement quality can be maintained or may be even better. The results also indicate that the net absorption is a good objective for Fine Cluster netlist generation.

	total WL (m)		**CPU(s)**	
bench	**Capo**	**FPI**	**Capo**	**FPI**
ibm08	396.7	390.9	2653	1079
ibm09	335.7	328.6	3098	921
ibm10	643.8	637	5083	1420
ibm11	49.7	49.9	5180	1522
ibm12	57.4	57.6	5614	1895
ibm13	40.6	40.2	6757	1792
ibm14	98.7	98.8	26922	5769
ibm15	112.2	109.2	30881	7183
ibm16	139.2	136.7	38779	8223
	100%	**98%**	**100%**	**24%**

TABLE 2. Comparison with Capo[4]

6. Conclusions

In this paper, we have explored the problem of size reduction for large-scale placement. We present a linear time net-absorption-based algorithm FGC to create a Fine Cluster netlist. With FGC, we propose a Fast Placer Implementation (FPI) framework. In FPI, we apply a time-consuming global placement optimization only to a Fine Cluster netlist. By embedding the existing standard cell placer into our FPI framework, we find that 3~5x placement speedup can be achieved by the FPI while the solution is of comparable or even better quality.

7. Acknowledgment

This work was supported by MARCO/DARPA Gigascale Silicon Research Center (GSRC).

Reference

[1] S. Hauck and G. Borriello, “An evaluation of bipartitioning techniques”, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol 16, No. 8, 1997.

[2] W. Sun and C. Sechen, “Efficient and effective placement for very large circuits”, Proc. IEEE Intl. Conf. Computer-Aided Design, pp. 170-177, 1993.

[3] C. M. Fiduccia and R. M. Mattheyses, “A linear time heuristic for improved network partitions”, Proc. Design Automation Conf, pp. 241-247, 1982.

[4] A. E. Caldwell, A. B. Kahng, I. L. Markov, “Can recursive bisection alone produce routable placements”, Design Automation Conference, 2000, pp. 260-263.

[5] J. Garbers, H. J. Promel, and A. Steger, “Finding clusters in VLSI circuits”, in Proc. Int. Conf. Computer-Aided Design, 1990, pp. 520-523.

[6] J. Cong, L. Hagen and A. B. Kahng, “Random walks for circuit clustering”, Proc. IEEE Intl. ASIC Conf. 1991, pp. 14.2.1-14.2.4.

[7] C. J. Alpert and A. B. Kahng. “A general framework for vertex ordering, with application to netlist clustering”, Proc. IEEE Intl. Conf. Computer-Aided Design, 1994, pp.63-67.

[8] D. J. Huang and A. B. Kahng, “When clusters meet partitions: new density-based methods for circuit decomposition”, In Proc. European Design and Test Conf., pp. 60-64, 1995

[9] J. Cong and M.L. Smith, “A parallel bottom-up clustering algorithm with applications to Circuit Partitioning in VLSI design”, Proc. ACM/IEEE Design Automation Conf. 1993.

[10]Y. C. Wei and C. K. Cheng, “Ratio cut partitioning for hierarchical designs”, IEEE trans. on Computer-Aided Design, pp 911-921, 1992.

[11]D. M. Schuler and E. G. Ulrich, “Clustering and linear placement”, In Proc. Design Automation Conf. 1972.

[12] L. Hagen and A. B. Kahng, “A new approach to effective circuit clustering”, Proc. IEEE Intl. Conf. on Computer-Aided Design, 1992, pp. 422-427.

[13]C. J. Alpert and A. B. Kahng, “Recent directions in netlist partitioning: a survey”, Integration, the VLSI Journal, pp. 1-81, 1995.

[14]J. Cong and S. K. Lim, “Edge separability based circuit clustering with application to circuit partitioning”, Proc. ASP-DAC, 2000, pp. 429-434.

[15]A. Singh, G. Parthasarathy, M. Marek-Sadowska, “Efficient circuit clustering and placement for area and power reduction in FPGAs”, In Proc. Intl. Symp. on Field Programmable Gate Arrays, 2002, pp. 59-66.

[16]C. C. Chang, J. Cong and Z. Pan, “Physical hierarchy generation with routing congestion control”, In Proc. Intl. Symp. on Physical Design, 2002, pp. 36-41.

[17]http://gigascale.org/bookshelf/

[18]J.M.Kleinhans, G.Sigl, F.M.Johannes and K.J.Antreich, “GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization”, IEEE Trans. CAD, vol.10, no.3, Mar1991, pp.356-365.