A Semi-Persistent Clustering Technique for VLSI Circuit Placement

Charles Alpert†, Andrew Kahng‡, Gi-Joon Nam†, Sherief Reda‡ and Paul Villarrubia†

†IBM Corp. 11400 Burnet Road, Austin, Texas, 78758

‡Department of CSE, UCSD, La Jolla, CA, 92093

{alpert, gnam, pgvillar}@us.ibm.com, {sreda, abk}@cs.ucsd.edu

ABSTRACT

Placement is a critical component of today's physical synthesis flow with tremendous impact on the final performance of VLSI designs. However, it accounts for a significant portion of the overall physical synthesis runtime. With complexity and netlist size of today's VLSI design growing rapidly, clustering for placement can provide an attractive solution to manage affordable placement runtime. Such clustering, however, has to be carefully devised to avoid any adverse impact on the final placement solution quality. In this paper we present a new bottom-up clustering technique, called *best-choice*, targeted for large-scale placement problems. Our best-choice clustering technique operates directly on a circuit hypergraph and repeatedly clusters the globally *best* pair of objects. Clustering score manipulation using a priority-queue data structure enables us to identify the best pair of objects whenever clustering is performed. To improve the runtime of priority-queue-based best-choice clustering, we propose a lazy-update technique for faster updates of clustering score with almost no loss of solution quality. We also discuss a number of effective methods for clustering score calculation, balancing cluster sizes, and handling of fixed blocks. The effectiveness of our best-choice clustering methodology is demonstrated by extensive comparisons against other standard clustering techniques such as Edge-Coarsening [12] and First-Choice [13]. All clustering methods are implemented within an industrial placer CPLACE [1] and tested on several industrial benchmarks in a *semi-persistent clustering* context.

Categories and Subject Descriptors

B.7.2 [Hardware]: INTEGRATED CIRCUITS – *Design Aids*; J.6 [Computer Applications]: COMPUTER-AIDED ENGINEERING – *Computer-Aided Design*.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

VLSI placement, Hypergraph clustering, Physical design

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISPD'05, April 3-6, 2005, San Francisco, California, USA.
Copyright ACM 1-59593-021-3/05/0004…$5.00.

1. INTRODUCTION

The task of VLSI placement is to assign exact locations to various circuit components within chip area. It typically involves optimizing a number of objectives such as wirelength, timing and power. The solution of placement has significant impact on the final performance of design, thus being considered as one of most critical processes in physical layout synthesis. However, the placement itself is an extremely computational intensive process which accounts for a significant portion of the overall physical synthesis runtime. With today's multi-million component designs, one iteration of physical synthesis flow can easily take a few days. This kind of turn-around time represents a serious obstacle to the productive development cycle in today's competitive market. The objective of this work is to speed up today's placement process via an effective clustering technique, particularly on large-scale designs, with almost no loss of solution quality.

A circuit netlist (or hypergraph) is composed of a set of objects and a set of hyperedges. An object represents a circuit component while a hyperedge represents electrical connection among objects. To speed up the placement, we reduce the netlist size by repetitively clustering objects together into larger artificial objects and adjusting the circuit netlist accordingly. This simplified circuit netlist is then used to drive the subsequent placement process. Clustering can effectively reduce the placement problem size allowing for faster placement turn-around time. The result of clustering, however, can directly affect placement solutions so that an intelligent clustering strategy is absolutely necessary to produce acceptable placement results.

In this paper we present a new bottom-up clustering algorithm called *best-choice*, which is targeted towards large scale reduction of a given circuit netlist. Such reduction is achieved by repeatedly identifying and clustering the *best* pair of objects among all object pairs, until the required target netlist size is attained. In essence, our clustering algorithm always selects the globally best clustering choice possible with a given clustering objective function. To rapidly identify the sequence of best-choice pairs of objects, an efficient priority-queue-based clustering score manipulation algorithm is proposed. Furthermore, additional technique called *lazy-updating* is proposed to dramatically reduce the runtime of best-choice clustering. The idea of lazy-updating is to defer clustering score updates as late as possible with almost no loss of clustering quality. We implement our methodology within a leading industrial placement tool CPLACE [1] and demonstrate its effectiveness on a number of large-scale industrial designs.

The rest of the paper is organized as follows. In Section 2 we present a brief overview of related works on clustering in placement and the motivation for this work. In Section 3, the best-choice clustering algorithm is described in addition to

lazy-update speed-up technique. Also, we discuss additional techniques to control cluster sizes leading to more balanced clustering. Our experimental results are presented in Section 4, and finally conclusions and future work are presented in Section 5.

2. PREVIOUS WORK AND MOTIVATION

Circuit clustering is an attractive solution to manage runtime and quality of placement results on large-scale VLSI designs. Naturally it has a long history of research activities [14, 4, 9, 15, 3, 2, 12, 13, 10, 6, 8, 11]. In terms of the interactions between clustering and placement, the prior work can be classified into two categories. The first category of clustering in VLSI placement uses *transient clustering* as part of the core placement algorithm [5, 16, 17]. In these approaches, the act of clustering and unclustering is generally part of the internal placement algorithm iterations. For example, in MLP (Multi-Level Partitioning [12])-based placers, a cluster hierarchy is first generated followed by a sequence of partitioning and unclustering. Partitioning result of prior clustered netlist is projected to the next level by unclustering, which becomes the seed for the subsequent partitioning. Typically, several partitioning attempts are executed, thereby providing for multiple clustering and unclustering operations as part of the inner loop of the placement algorithm. Furthermore, concepts such as V-cycling [12] have been introduced where multiple clusterings occur at each level of the hierarchy for further optimization.

The second category of clustering in VLSI placement involves *persistent clusters*. In this case, the cluster hierarchy is generated at the beginning of the placement in order to reduce the size of the problem. Then, the coarsened netlist is presented to the placer [11]. Usually, the clustered objects will be dissolved at or near the end of the placement process, with a "clean-up" operation applied to the uncoarsened netlist to improve the results. In some cases, these approaches take the opportunity to uncluster and/or recreate clusters at strategic points in the placement flow [7]. In these methods, however, it can be argued that the clustering algorithm itself is not part of the core placement algorithm, but rather a preprocessing step which produces a smaller/simpler netlist structure for the placement algorithm. For instance, in relatively time consuming simulated annealing placement [15], significant runtime speed-up can be achieved with persistent clustering.

Semi-Persistent clustering falls into the second category. Persistent clustering offers significant runtime improvements at the expense of the quality of the final placement solution. This problem is particularly magnified as more clustering operation is performed. Another problem associated with persistent-clustering is the control of physical cluster sizes. During the placement flow, the size of clustered objects may be too large relative to the decision granularity, which results in the degradation of final placement solution quality. Therefore, the goal of our semi-persistent clustering is to address these two deficiencies. First, we seek to generate high quality clustering solution so that any potential loss of final placement solution quality is minimized (or prevented). Secondly, we take advantage of the hierarchical nature of clustering so that clustered objects are dissolved slowly during the placement flow. In this way, during the early stage of the placement algorithm, a global optimization process is performed on highly coarsened netlist while local optimization/refinement can be executed on almost flattened netlist at later stage.

We now review some of the relevant literature on clustering. In Edge-Coarsening (EC) [12, 2], objects are visited in a random order, and for each object u, only a set of unmatched adjacent objects (i.e., objects that have never been visited or clustered before) is considered. Among these objects the one with the largest weight is matched to u. In EC, a hyperedge of k pins is assigned a weight of $1/(k-1)$. Karypis and Kumar [13] modified the EC scheme and proposed the First-Choice (FC) clustering approach. In FC, similar to EC, objects are visited in a random order. But for each object u, all objects that are adjacent to u, regardless of their matching status, are considered. Again, the object with the largest weight is matched to u. Thus, a clustered object with multiple layers of clustering hierarchy can be formed. To limit the cluster size, FC stops clustering when the coarsened netlist reaches a certain threshold.

In another approach, Cong and Lim [10] transform a given hypergraph into a graph by decomposing every k-pin hyperedge into a clique, with an edge weight $1/(k-1)$. Then, they (i) rank edges according to a connectivity-based metric using a priority-queue data structure, (ii) cluster two objects with highest ranking edge if their sizes do not exceed a certain size limit, and (iii) update circuit netlist and priority-queue structure accordingly. We note that decomposing a hyperedge into a clique can cause discrepancy in edge weights once any two objects of a k-pin hyperedge are clustered. This discrepancy leads to incorrect edge weights as demonstrated by the following example.

Example 1: Assume that two objects $v_1 \in e$ and $v_2 \in e$ were clustered, where e is a k-pin hyperedge. In Karypis and Kumar's scheme [13], the clustering score of any other objects in e becomes to $1/(k-1-1)$, while in Cong and Lim's scheme [10], the clustering score stays same as $1/(k-1)$. This edge weight discrepancy occurs because the transformation of a hyperedge to a clique of edges is performed only once before clustering starts[1].

Chan *el.* [6] uses a connectivity-based approach similar to [10]. The difference is that the area of a clustered object is included in the objective function to produce more balanced clustering. The inclusion of cluster size in an objective function is originally proposed in [14]. Another recent approach [11] proposes fine-grain clustering particularly targeted for improving runtime in mincut-based placement. The approach decomposes hyperedges into cliques, and uses a connectivity-based net weighting scheme similar to [14]. A priority queue is used to rank all the edges according to the calculated net weights. Clustering proceeds in an iterative fashion. At each iteration, clustering is allowed only if both target objects have never been visited before during the same iteration. A cluster is typically limited to few (2-3) objects; thus, the name fine-grain clustering.

We think that the general drawbacks of previous approaches are:

- Hypergraph to graph transformation [15, 6, 10, 11] leads to discrepancy in edge weights and increases the size of required priority-queue.
- Pass-based clustering methods (i.e., clustering iterations) [11, 12, 2] that disallow an object to be revisited during

[1] Note however that the clustered object of $\{v_1, v_2\}$ will have a score of $2/(k-1)$ to other objects on the same hyperedge e [10].

Input: Flat Netlist
Output: Clustered Netlist

1. Until *target object number* is reached:
   2. Find *closest pair* of objects
   3. Cluster them
   4. Update netlist

Figure 1. Bottom-up best-choice clustering.

the same iteration lead to suboptimal choices because an object might be prevented from getting clustered to its best neighbor.

- Non-priority-queue based implementations [13] lead to suboptimal clustering choices due to the lack of a global picture of clustering sequences.

Given this brief overview of related work, we next describe our clustering method.

3. BEST-CHOICE BOTTOM-UP CLUSTERING FOR PLACEMENT

A high-level outline of the bottom-up best-choice clustering is given in Figure 1. The key idea of best-choice clustering is to identify the globally best pair of objects to cluster by managing a priority-queue data structure with the clustering score as a key. Priority-queue management naturally provides an ideal clustering sequence and it is always guaranteed that two objects with the best clustering score will be clustered.

The degree of clustering is controlled by *target clustering ratio* α. The *target number of objects* is determined by the original number of objects divided by the target clustering ratio α. The clustering operation is simply repeated until the overall number of objects becomes the calculated target number of objects. For example, a target clustering ratio of $\alpha = 10$ indicates that the clustered netlist will have one tenth the number of movable object in the original netlist.

The challenges associated with best-choice clustering are as follows:

- Using an efficient and effective clustering score function which leads to higher quality placement solutions.
- Accurately handling hyperedges.
- Efficient netlist and priority-queue data structure updating after each clustering is performed.
- Controlling clustered object size for more balanced clustering.
- Handling fixed blocks and associated movable objects around these fixed blocks.

We address these challenges as follows.

3.1 Best-choice Clustering Score Function

The weight w_e of a hyperedge e is defined as $1/|e|$. Thus, the weight is inversely proportional to the number of objects that are incident to the hyperedge. Given two objects u and v, the *clustering score d(u, v)* between u and v is defined as:

$$d(u, v) = \sum_e w_e / [a(u) + a(v)] \quad (1)$$

where *a(u)* and *a(v)* are the areas of u and v respectively. The clustering score of two objects is directly proportional to the total sum of edge weights between them, and inversely proportional to the sum of their areas. Suppose N_u is the set of neighboring objects to a given object u. We define the *closest object* to u, denoted *c(u)*, as the neighbor object with the largest clustering score to u, i.e.,

$$c(u) = v \text{ such that } d(u, v) = max\ d(u, z) \text{ for } \forall z \in N_u \quad (2)$$

Input: Flat Netlist
Output: Clustered Netlist

Phase I. Priority-queue PQ Initialization:
1. For each object u:
2. Find *closest object* v, and its associated clustering score d
3. Insert tuple (u, v, d) into PQ with d as key

Phase II. Clustering:
1. While *target object number* is not reached and top tuple's score $d > 0$:
   2. Pick top tuple (u, v, d) of PQ
   3. Cluster u and v into new object u'
   4. Update netlist
   5. Find *closest object* v' to u' with its clustering score d'
   6. Insert tuple (u', v', d') into PQ with d' as key
   7. Update clustering scores of all neighbors of u'

Figure 2. Best-choice clustering algorithm.

In order to identify the globally closest pair of objects with the best clustering score, a priority-queue based implementation is proposed as given in Figure 2. The best-choice algorithm is composed of two phases. In phase I, for each object u in the netlist, the closest object v and its associated clustering score d are calculated. Then, the tuple (u, v, d) is inserted to the priority-queue with d as key. For each object u, only one tuple with the closest object v is inserted. This vertex-oriented priority queue allows for more efficient data structure managements than edge-based methods. Phase I is a simply priority queue *PQ* initialization step.

In the second phase, the top tuple (u, v, d) in *PQ* is picked up (Step 2), and the pair of objects (u, v) are clustered creating a new object u' (Step 3). The netlist is updated (Step 4), the closest object v' of the new object u' and its associated clustering score d' are calculated, and a new tuple (u', v', d') is inserted to *PQ* (Steps 5-6). Since clustering changes the netlist connectivity, some of previously calculated clustering scores might be rendered invalid. Thus, the clustering scores of the neighbors of the new object u', (equivalently all neighbors of u and v) need to be re-calculated (Step 7), and *PQ* is adjusted accordingly. The following example illustrates clustering score calculation and updating.

Example 2: Assume the input netlist with 5 objects $\{A, B, C, D, E, F\}$ and 8 hyper-edges $\{A, B\}$, $\{A, D\}$, $\{A, E\}$, $\{A, F\}$, $\{A, C\}$, another $\{A, C\}$, $\{B, C\}$ and $\{A, C, F\}$ as in Figure 3 (a). By calculating the clustering score of A to its neighbors, we find that *d(C, B) = 1/2*, *d(A,B) = 1/2*, *d(A, C) = 4/3*, *d(A, D) = 1/2*, *d(A,E) = 1/2*, and *d(A, F) = 5/6*. *d(A, C)* has the highest score, and C is declared as the closest object to A. If we assume that *d(A, C)* is the highest score in the priority queue, A will be clustered with C and the circuit netlist will be updated as shown in Figure 3 (b). With a new object AC introduced, corresponding cluster scores will be *d(AC, F)= 1*, *d(AC, E) = 1/2*, *d(AC, D) = 1/2*, and *d(AC, B) = 1*.

We can summarize the main advantages of our best-choice clustering methodology as follows.

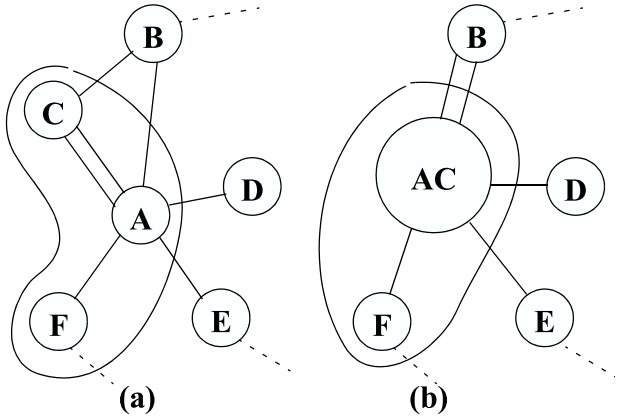

Figure 3. Clustering a pair of objects *A* and *C*.

- Clustering will always be performed on the overall best choice.
- Direct hyperedge handling without converting hyperedges into clique or star models [10, 11].
- Object-based priority queue manipulation. Thus, the size of priority queue is equal to the number of objects (rather than edges [10]) in netlist by recording only the closest neighbor per object, leading to more efficient priority queue management.

As will be demonstrated in Section 4, the best-choice scheme can produce high quality clustering results for subsequent placement. However, the overall clustering runtime is not yet competitive to other faster clustering algorithms such as edge-coarsening (EC) [12] or first-choice (FC) [13]. To improve the runtime of best-choice, we propose to update the priority queue in a *lazy* fashion.

3.2 Speed-up Technique for Best-Choice Clustering: Lazy-Update

Analyzing the runtime characteristic of the best-choice clustering algorithm of Figure 2, we find that (Step 7) is the most time consuming task. To update the score priority queue after each clustering, each neighbor object of a newly created object needs to be visited to find its *new* closest object and its clustering score. The closest object of a given target object u can only be found by visiting all the neighbor objects of u. Therefore, updating the clustering scores after a clustering operation (Step 7) typically involves two levels of netlist exploration.

Statistical analysis of clustering score priority queue management, however, reveals the following facts:

1. An object in the priority queue might be updated a number of times before making to the top (if ever). Effectively, all the updates but the last one are useless since only the final update determines the final location within the priority queue.
2. In 96% of clustering score updates, a new score decreases, i.e., most of time, objects are moving downward the priority queue rather than popping up.

Motivated by these observations, we propose *Lazy-Update* technique which delays updates of clustering scores as late as possible, thus reducing the actual number of score update operations on the priority queue. More specifically, lazy updating

Input: Flat Netlist
Output: Clustered Netlist

Phase II. Clustering:
1. While *target object number* is not reached and top tuple's score $d > 0$:
2. Pick top tuple (u, v, d) of PQ
3. If u is marked as invalid, re-calculate *closest object* v' and score d' and insert tuple (u, v', d') to PQ
4. else
5. Cluster u and v into new object u'
6. Update netlist
7. Find *closest object* v' to u' with its clustering score d'
8. Insert tuple (u', v', d') into PQ with d' as key
9. Mark all neighbors of u' as invalid

Figure 4. Lazy-Update speed-up technique for best-choice clustering.

waits until an object reaches the top of the priority queue and only then updates the object's score if necessary. The modification to the clustering phase (Phase II) is shown in Figure 4. In Step 9 of the modified algorithm, we only *mark* neighbor objects as invalid instead of re-calculating their scores. When an object is picked up from the top of priority queue, we check whether it is marked or not. If it is marked (invalid), its new closest object and its score are re-calculated and re-inserted into the priority queue (Step 3); otherwise (valid), it is clustered with its pre-calculated closest object. In the experimental section, we demonstrate that Lazy-Update technique can dramatically reduce the clustering runtime at almost no adverse impact on clustering quality.

3.3 Cluster Size Growth Control

The presence of the area function in the denominator of EQ (1) provides an indirect way to automatically control the sizes of clustered objects, potentially leading to more balanced clustering results. Without such an area control, gigantic clustered objects might be formed by absorbing small objects and/or clusters around it. In this section, two classes of cluster size control methods are discussed; *indirect* vs. *direct* control.

3.3.1 Indirect Size Control

The cluster size is controlled automatically via a clustering score function as in EQ (1) which is inversely proportional to the size of cluster object. A more generic form of this approach will be as follows. Given a target object u and its neighbor v, a clustering score between u and v is defined as:

$$d(u,v) = \sum_e w_e / [a(u) + a(v)]^k \qquad (3)$$

where $k \geq 1$. k can be either fixed number or it can be dynamically adjusted by setting it to $k = \lceil (a(u) + a(v)) / \mu \rceil$, where μ is the average cell area multiplied by the clustering ratio α. μ represents the expected average size of clustered objects. Another possibility is to use the total number of pins instead of object area, because in general, the number of pins in a cluster is well-correlated with its cluster size.

3.3.2 Direct Size Control

The clustering algorithm can take a more direct approach by imposing a bound constraint on the size of clusters. Given two objects u and v, two methods are proposed:

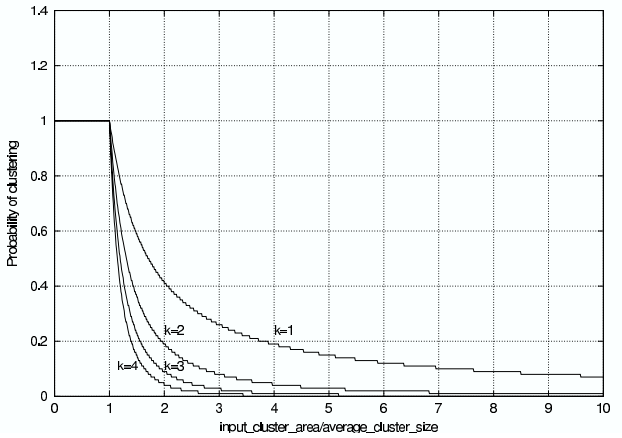

Figure 5. Probabilistic cluster size control curves.

- *Hard Bound:* If the total area $a(u)+a(v) \le k \cdot \mu$, then accept clustering, else reject it.
- *Soft Bound:* If the total area $a(u)+a(v) \le k \cdot \mu$, then accept clustering, else accept it with probability equal to $2^{(\mu/(a(u)+a(v)))^k} - 1$ where $k \ge 1$.

With hard bound, an upper bound on the cluster size is strictly enforced while with soft bound, the upper bound is slightly relaxed, where the probability of clustering two objects declines as the sum of their areas increases. The parameter k controls the amount of relaxation. The plot of Figure 5 demonstrates the probability of clustering two objects for various values of k. The x-axis is $(a(u)+a(v))/\mu$ (we assume here that $\mu = 1$, and the y-axis shows the corresponding probability of clustering occurrence. The cluster bounds, whether hard or soft, can be incorporated in two ways during the calculation of the closest objects:

- Method A: Pick the closest object among all neighbors and check if the chosen object satisfies the area constraints.
- Method B: Pick the closest object only from the set of neighbor objects that satisfy the are constraints.

Basically, method A ensures to choose the object with the highest clustering score despite that it might get rejected due to the area constraint violation later, while method B ensures that the chosen object meets the area constraints, despite that its clustering score might not be the highest among all neighbor objects. Empirically, we have found that using method A produces better results than method B. The empirical comparison of different methods of cluster size control will be given in Section 4.

3.4 Handling Fixed Blocks during Clustering

The presence of fixed blocks in netlist might alter how clustering is performed. We observe that sometimes, particularly when significant degree of clustering is performed, movable cells directly connected to fixed blocks are being clustered to objects located far away from them. This might cause adverse impact on timing results as well as placement wirelengths. Ideally, movable objects around fixed blocks need to be placed around those directly connected fixed blocks after placement, regardless of the degree of clustering performed. To our knowledge, there has been no prior clustering work that explicitly considers fixed blocks and their neighbor movable objects. We have attempted the following techniques to address this issue:

- Ignoring all nets connected to fixed objects since such nets cannot be eliminated by clustering.
- Ignoring all pins connected to fixed objects thus altering the weight of nets connecting movable objects and fixed blocks.
- Incorporating fixed blocks during clustering, only to remove them from the clusters after the clustering is done and before placement process starts.
- Chaining all movable cells attached to a fixed by a set of additional artificial nets to control their affinity to fixed blocks during placement.

However, none of above techniques made distinguishable improvements to the final placement results, and we leave this topic as future work to further explore effective handling of fixed blocks during clustering.

Bench	Cells	Blks	IOs	Nets	Density	Util
AL	270163	4235	14100	292425	44.74%	22.54%
BL	276194	14461	17380	327102	69.37%	20.37%
CL	351056	26713	6360	395918	82.32%	37.59%
DL	425610	14665	17960	465927	56.25%	34.91%
EL	457516	3460	7094	478842	72.12%	51.90%
FL	880410	53481	23255	1010392	74.77%	48.19%
AD	389226	0	35944	401463	87.65%	83.55%
BD	285085	0	13286	309050	87.76%	85.87%
CD	56436	2	1968	57595	57.43%	57.32%

Table 1. Benchmark characteristics.

4. EXPERIMENTAL RESULTS

We implement the proposed clustering methodology within the industrial placement tool CPLACE [1]. The placer uses a quadratic solver in a top-down partitioning framework, where clustering is executed at the beginning as a pre-processing step to condense the netlist. The main placement algorithm is then invoked on the clustered netlist. Clustered objects get dissolved when their sizes exceed a pre-determined percentage of their assigned bin's size. Typically, most of the unclustering typically occurs towards the end of global placement and before the start of detailed placement.

The effectiveness of our clustering approach is tested on a number of large-scale industrial benchmarks ranging from 250K to one million objects. The benchmarks' detailed characteristics are given in Table 1. Column "Cells" gives the number of movable cells in the design, column "Blks" gives the number of fixed blocks, column "IOs" gives the total number of IO-related objects, column "Nets" gives the total number of nets, column "Density" gives the design density defined as total object area divided by total placement are, and column "Util" gives the design utilization (or density) where it is defined as the area sum of only movable cells divided by the available free space. We have two sets of benchmarks. The

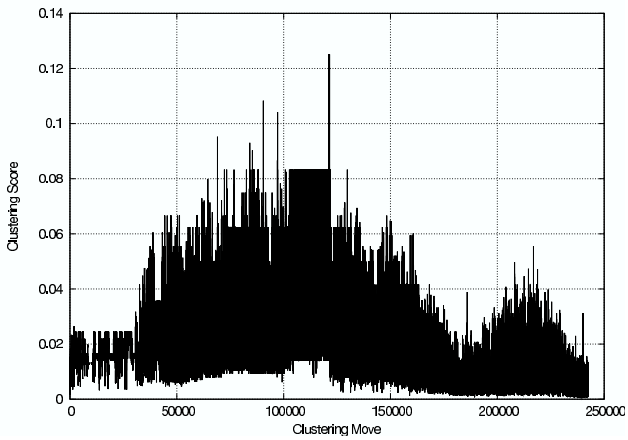

Figure 6. Edge-Coarsening clustering score plot. Total clustering score = 5301.05. Clustering runtime = 9.23 sec.

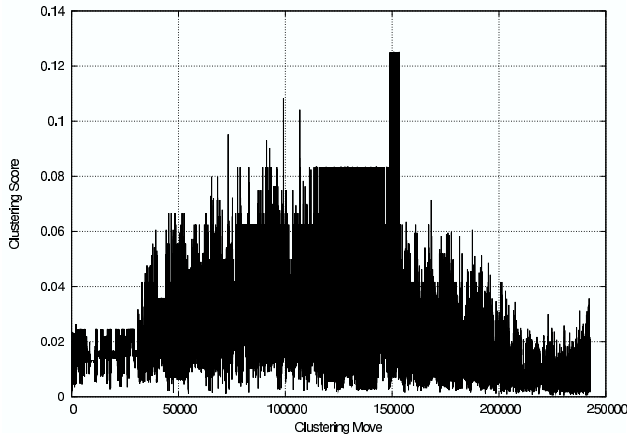

Figure 7. First-Choice clustering score plot. Total clustering score = 5612.83. Clustering runtime = 9.03 sec.

benchmarks whose name ends with 'L' are low utilization designs whereas those ending with 'D' have high design utilization.

To speed up the placement process significantly, a rather high clustering ratio $\alpha = 10$ is used reducing the number of objects in benchmark by an order of magnitude. In all experiments, a workstation with 4 Intel Xeon 2.40 GHz CPUs, 512KB cache and 6GB of memory is used.

In the first series of experiments, we demonstrate that with the best-choice approach, placement produces better solutions than with other standard clustering approaches. For comparison purposes, we implement two approaches:

- Edge-Coarsening (EC) [12]: The objects are visited in a random order. For each object u, all unmatched neighbor objects, (i.e., objects that have never been visited or clustered before), are considered. Among those, the one with the highest weight score is matched to u.
- First-Choice (FC) [13]: This is an improvement over EC. Again, objects are visited in a random order, but each visited object is clustered to its closest neighbor whether it has been visited before or not. To limit the cluster size, clustering stops when the hypergraph after clustering reaches a certain threshold.

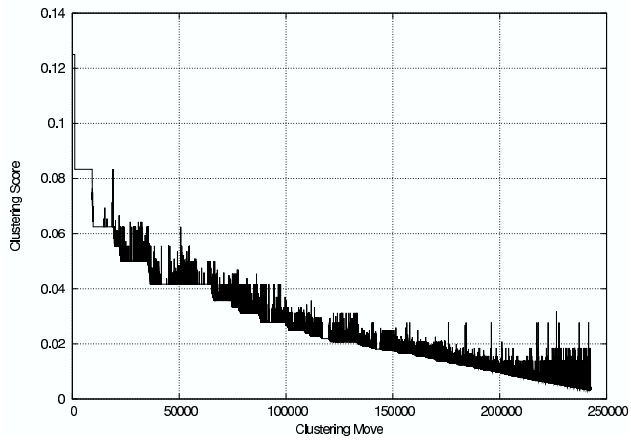

Figure 8. Best-Choice clustering score plot. Total clustering score = 6671.53. Clustering runtime = 97.35 sec.

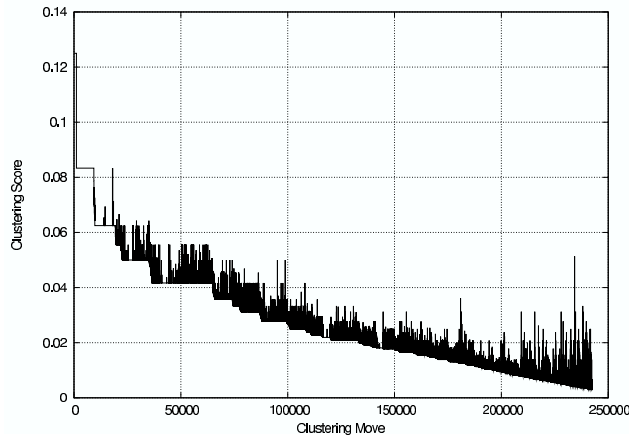

Figure 9. BC clustering with Lazy Update score plot. Total clustering score=6658.23. Clustering runtime=49.84 sec.

We first construct *clustering score plots* for a typical benchmark, AD, as shown in Figure 6, 7, 8 and 9, where the x-axis gives the sequence of clustering operations, and the y-axis gives the corresponding clustering score. For example, a 10 on the x-axis represent the 10th clustering execution. We also compute the total clustering score value of each clustering method. From the figures, the total clustering score of edge-coarsening, first-choice and best-choice are 5301, 5612 and 6671, respectively. Clearly, best-choice achieves the highest total clustering score among these three methods. Also, the figure shows that using lazy update reduces the clustering runtime by around 50% with almost no loss in the total clustering score -a negligible drop from 6671 to 6658.

We now investigate how different clustering algorithms affect the final placement results. Table 2 presents the final placement wirelength of the EC (Edge-Coarsening), FC (First-Choice) and BC (Best-Choice) clustering algorithms on the benchmarks of Table 1. All results are normalized with respect to EC. We compare BC and BC+Lazy Update versus FC and EC. In the table, "CPU" shows clustering CPU times with respect to that of EC. "WL(%)" presents the percentage improvement in Half-Perimeter wirelength (HPWL) over EC's HPWL. We make the following observations:

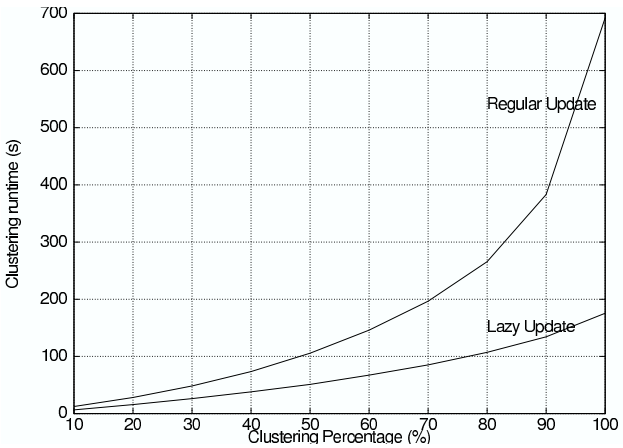

Figure 10. Runtime breakdown of best-choice clustering with/without lazy update on benchmark FL.

- Best-Choice clustering dominates other standard clusterings on all benchmarks with an average improvement of 4.3%.
- Lazy Update significantly improves Best-Choice clustering runtime for all benchmarks with an average runtime reduction of 57% without almost no impact to quality of results, only a 0.11% change in HPWL.

Figure 10 shows the runtime breakdown of best-choice clustering on the largest benchmark FL. From the plot, we immediately notice that the runtime reduction with Lazy-Update increases as cluster progresses, i.e., as more clusterings are performed, Lazy-Update becomes more effective.

In the second experiment, we examine how cluster size control affects the quality of placement results. We have found that cluster size control is particularly critical to dense designs with no or few fixed blocks (AD, BD, and CD). For sparse benchmarks, the ones ending with "L", no distinguishable impact has been found with different size control methods. Three additional area control methods are implemented and compared to the standard area control method with EQ (1). In Table 3, "Automatic" refers to the method using EQ (3) with $k = 2$. "Hard" and "Soft" refer to the bounding methods of Section 3.3.2. Both hard and soft bounds are executed with $k = 3$. All size control methods are implemented within our best-choice clustering framework. The quality of solution is measured by the final placement wirelength. We also report the maximum and average cluster sizes after clustering is done. From the table, we observe that careful control of the cluster size can improve the placement wirelength by up to 13%. The results also indicate that probabilistic control of cluster size, "Soft", produces the best results. We think that soft probabilistic control occasionally provide a way to form a high quality but slightly larger clusters leading to better results.

The comparisons between flat (i.e., no clustering) and semi-consistent clustering CPLACE runs are demonstrated in Table 4. The CPLACE run with best-choice-lazy-update clustering speeds up the overall placement on the average by x2.1. More interestingly, with best-choice clustering, CPLACE was

Bench	EC		FC		BC		BC+LazyUpdate	
	WL(%)	CPU	WL(%)	CPU	WL(%)	CPU	WL(%)	CPU
AL	0.00	1.00	-0.43%	x0.98	3.19%	x9.77	2.10%	x4.02
BL	0.00	1.00	3.30%	x0.94	6.99%	x8.69	6.28%	x3.61
CL	0.00	1.00	2.14%	x0.96	3.43%	x4.99	4.23%	x3.29
DL	0.00	1.00	0.44%	x1.03	2.22%	x9.76	2.07%	x3.84
EL	0.00	1.00	2.37%	x0.98	5.72%	x4.92	6.27%	x3.37
FL	0.00	1.00	-1.14%	x1.04	4.33%	x14.48	4.25%	x4.50
Avg.	**0.00**	**1.00**	**1.11%**	**x0.98**	**4.31%**	**x8.76**	**4.20%**	**x3.77**

Table 2. Clustering results. All results are normalized with respect to Edge-Coarsening (EC). Column "CPU" presents the clustering runtime in comparison to EC. Column "WL" shows the improvement in HPWL over EC.

Bench	Stnd			Automatic			Hard			Soft		
	Max	Avg	WL(%)	Max	Avg	WL(%)	Max	Avg	WL(%)	Max	Avg	WL(%)
AD	14823	171.4	0.00	1140	160.4	-0.88%	364	169.9	0.47%	1668	169.6	0.56%
BD	28600	150.0	0.00	1140	114.6	3.71%	405	147.9	5.89%	1520	147.9	4.86%
CD	9060	113.5	0.00	610	109.8	30.05%	280	116.1	29.11%	1075	114.9	34.16%
Avg.	**-**	**-**	**0.00**	**-**	**-**	**10.96%**	**-**	**-**	**11.82%**	**-**	**-**	**13.19%**

Table 3. Impact of cluster-size control on the total HPWL for dense designs. Column "Automatic" controls cluster sizes via the exponent to the area term in the clustering score function. Here, the exponent is 2 in contrast to 1 in "Stnd". Column "Hard" controls cluster sizes by imposing a hard upper bound. Column "Soft" controls cluster sizes with probability. Column "Max" and "Avg" are the max and the average cluster size, "WL(%)" is the improvement in HPWL over "Stnd" area control method.

Bench	WL(%)	CPU	CL-CPU
AL	2.09%	x0.40	1.17%
BL	-4.28%	x0.52	1.35%
CL	3.27%	x0.51	1.14%
DL	0.87%	x0.45	1.35%
EL	1.59%	x0.33	1.10%
FL	1.41%	x0.46	1.68%
AD	8.23%	x0.50	0.98%
BD	-0.34%	x0.47	0.94%
CD	-0.36%	x0.69	0.51%
Avg.	**1.39%**	**x0.48**	**1.14%**

Table 4. Comparison between flat (no clustering) and best-choice+Lazy Update clustering CPLACE runs.

able to produce 1.39% better final wirelengths on average. Column "CL-CPU" shows the portion of clustering CPU time in overall CPLACE run. Although best-choice-lazy-update clustering takes 3.77x more CPU time than EC as shown in Table 2, it takes only 1.14% of overall CPU time.

5. CONCLUSIONS AND FUTURE WORK

We summarize our contributions as follows.

- Globally optimal clustering choices are always identified and clustered.
- Clustering operates directly on a given hypergraph without any edge transformations that can lead to inaccuracies in computing clustering scores.
- Object-based priority queue score manipulation with a priority queue size being equal to the number of objects in netlist. This contrasts to existing edge-based method [10]. Thus, the best-choice approach is likely to be faster.
- Faster best-choice clustering implementation based on the idea of Lazy-Update that defers updating clustering scores as late as possible.
- Exploration of cluster size control methods to produce more balanced clustering, resulting in higher quality placement solutions.

The proposed methods are implemented in a leading industrial top-down quadratic placement engine CPLACE and their effectiveness are demonstrated though comparisons against leading clustering algorithms, such as edge-coarsening (EC) and first-choice (FC), on a number of large-scale industrial benchmarks. Our results show that placement runtime has been sped up by a factor of 2.2 with consistent wirelength improvements, on average 4.20% over EC and FC. For the future work, we believe that higher quality of placement solutions is achievable by better handling of fixed blocks during clustering.

6. REFERENCES

[1] C. J. Alpert, G.-J. Nam, and P. G. Villarrubia, "Effective Free Space Management for Cut-based Placement", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 22(10), pp. 1343-1353, 2003.

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

[3] C. J. Alpert and A. B. Kahng, "A General Framework for Vertex Orderings with Application to Netlist Clustering," in *Proc. ACM/IEEE Design Automation Conference,* 1994, pp. 63-67.

[4] T. N. Bui, "Improving the Performance of Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms,", in *Design Automation Conference,* 1989, pp.775-778.

[5] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?", in *Proc. ACM/IEEE Design Automation Conference,* 2000, pp. 477-482.

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

[7] C.-C. Chang, J. Cong, D. Pan, and X. Yuan, "Multilevel Global Placement with Congestion Control," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 22(4), pp. 395-409, 2003.

[8] C.-C. Chang, J. Cong, D. Pan, and X. Yuan, "Physical Hierarchy Generation with Routing Congestion Control," in *Proc. ACM/IEEE International Symposium on Physical Design,* 2002, pp. 36-41.

[9] J. Cong, L. Hagen, and A. B. Kahng, "Random Walks for Circuit Clustering," in *IEEE International Conference on ASIC,* 1991, pp. 14.2.1-14.2.4.

[10] J. Cong and S. K. Lim, "Edge Separability-Based Circuit Clustering with Application to Multilevel Circuit Partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 23(3), pp. 346-357, 2004.

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

[12] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Application in VLSI Domain," in *Proc. ACM/IEEE Design Automation Conference,* 1997, pp. 526-529.

[13] G. Karypis and V. Kumar, "Multilevel *k*-way Hypergraph Partitioning," in *Proc. ACM/IEEE Design Automation Conference,* 1999, pp. 343-348.

[14] D. M. Schuler and E. G. Ulrich, "Clustering and Linear Placement," in *Proc. ACM/IEEE Design Automation Conference,* 1972, pp. 50-56.

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

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

[17] M. Yildiz and P. Madden, "Global Objectives for Standard-Cell Placement," in *Proc. IEEE Great Lakes Symposium on VLSI,* 2001, pp. 68-72.